Stephen Jarvis · Steven Wright
Simon Hammond (Eds.)

LNCS 10724

# High Performance Computing Systems

## Performance Modeling, Benchmarking, and Simulation

8th International Workshop, PMBS 2017
Denver, CO, USA, November 13, 2017
Proceedings

EXTRAS ONLINE

# Lecture Notes in Computer Science 10724

*Commenced Publication in 1973*
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen


Editorial Board

David Hutchison
*Lancaster University, Lancaster, UK*
Takeo Kanade
*Carnegie Mellon University, Pittsburgh, PA, USA*
Josef Kittler
*University of Surrey, Guildford, UK*
Jon M. Kleinberg
*Cornell University, Ithaca, NY, USA*
Friedemann Mattern
*ETH Zurich, Zurich, Switzerland*
John C. Mitchell
*Stanford University, Stanford, CA, USA*
Moni Naor
*Weizmann Institute of Science, Rehovot, Israel*
C. Pandu Rangan
*Indian Institute of Technology, Madras, India*
Bernhard Steffen
*TU Dortmund University, Dortmund, Germany*
Demetri Terzopoulos
*University of California, Los Angeles, CA, USA*
Doug Tygar
*University of California, Berkeley, CA, USA*
Gerhard Weikum
*Max Planck Institute for Informatics, Saarbrücken, Germany*

More information about this series at http://www.springer.com/series/7407

Stephen Jarvis · Steven Wright
Simon Hammond (Eds.)

# High Performance Computing Systems

## Performance Modeling, Benchmarking, and Simulation

8th International Workshop, PMBS 2017
Denver, CO, USA, November 13, 2017
Proceedings

Springer

*Editors*
Stephen Jarvis
University of Warwick
Coventry
UK

Simon Hammond
Sandia National Laboratories
Albuquerque, NM
USA

Steven Wright
University of Warwick
Coventry
UK

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-72970-1 ISBN 978-3-319-72971-8 (eBook)
https://doi.org/10.1007/978-3-319-72971-8

Library of Congress Control Number: 2017962895

LNCS Sublibrary: SL1 – Theoretical Computer Science and General Issues

© Springer International Publishing AG 2018
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

# Special Issue on the 8th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS 2017)

This volume contains the 13 papers that were presented at the 8th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems (PMBS 2017), which was held as part of the 29th ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 2017) at the Colorado Convention Centre in Denver between 12–17 November 2017. SC offers a vibrant technical program, which includes technical papers, tutorials in advanced areas, Birds of a Feather sessions (BoFs), panel debates, a doctoral showcase, and a number of technical workshops in specialist areas (of which PMBS is one). The focus of PMBS is comparing high performance computing systems through performance modeling, benchmarking, or the use of tools such as simulators. Contributions are sought in areas including: performance modeling and analysis of applications and high performance computing systems; novel techniques and tools for performance evaluation and prediction; advanced simulation techniques and tools; micro-benchmarking, application benchmarking, and tracing; performance-driven code optimization and scalability analysis; verification and validation of performance models; benchmarking and performance analysis of novel hardware; performance concerns in software/hardware co-design; tuning and auto-tuning of HPC applications and algorithms; benchmark suites; performance visualization; real-world case studies; studies of novel hardware such as Intel's Knights Landing platform and NVIDIA Pascal GPUs.

The 8th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS 2017) was held on November 13 as part of the 29th ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 2017) at the Colorado Convention Center in Denver during November 12–17, 2017.

The SC conference is the premier international forum for high performance computing, networking, storage, and analysis. The conference is unique in that it hosts a wide range of international participants from academia, national laboratories, and industry; this year's conference attracted over 13,000 attendees and featured over 350 exhibitors in the industry's largest HPC technology fair.

This year's conference was themed "HPC Connects," encouraging academia and industry to come together to inspire new collaborations between different fields of science, with the goal of bringing about an impact on society and the changing nature of our world.

SC offers a vibrant technical program, which includes technical papers, tutorials in advanced areas, Birds of a Feather sessions (BoFs), panel debates, a doctoral showcase, and a number of technical workshops in specialist areas (of which PMBS is one).

The focus of the PMBS 2017 workshop was comparing high performance computing systems through performance modeling, benchmarking, or the use of tools such as simulators. We were particularly interested in receiving research papers that reported on the ability to measure and make trade-offs in hardware/software co-design to improve sustained application performance. We were also keen to capture the assessment of future systems, for example, through work that ensured continued application scalability through peta- and exa-scale systems.

Like SC 2017, the aim of the PMBS 2017 workshop was to bring together researchers from industry, national labs, and academia, who are concerned with the qualitative and quantitative evaluation and modeling of high performance computing systems. Authors were invited to submit novel research in all areas of performance modeling, benchmarking, and simulation, and we welcomed research that combined novel theory and practice. We also expressed an interest in submissions that included analysis of power consumption and reliability, and were receptive to performance modeling research that made use of analytical methods as well as those based on tracing tools and simulators.

Technical submissions were encouraged in areas including: performance modeling and analysis of applications and high performance computing systems; novel techniques and tools for performance evaluation and prediction; advanced simulation techniques and tools; micro-benchmarking, application benchmarking, and tracing; performance-driven code optimization and scalability analysis; verification and validation of performance models; benchmarking and performance analysis of novel hardware; performance concerns in software/hardware co-design; tuning and auto-tuning of HPC applications and algorithms; benchmark suites; performance visualization; real-world case studies; and studies of novel hardware such as the Intel's Knights Landing platform and NVIDIA Pascal GPUs.

## PMBS 2017

We received a good number of submissions for this year's workshop. This meant that we were able to be selective in those papers that were chosen; the acceptance rate for papers was approximately 35%. The resulting papers show worldwide programs of research committed to understanding application and architecture performance to enable exascale computational science.

The workshop included contributions from Argonne National Laboratory, Brookhaven National Laboratory, Clemson University, École Normale Supérieure de Lyon, Edinburgh Parallel Computing Centre, ENS Lyon, Florida State University, Hewlett Packard Labs, Inria, Lawrence Berkley National Laboratory, Los Alamos National Laboratory, New Mexico State University, NVIDIA Corporation, Pacific Northwest National Laboratory, Pazmany Peter Catholic University, Universidade de Lisboa, University of Basel, University of Bristol, University at Buffalo, University of Cambridge, University of Chicago, University of Florida, University of Tennessee, University of Udine, University of Warwick, and Vanderbilt University.

Several of the papers are concerned with "Performance Evaluation and Analysis" (see Section A). The paper by Nathan Tallent et al. discusses the performance differences between PCIe- and NVLink-connected GPU devices on deep learning workloads. They demonstrate the performance advantage of NVLink over PCIe- connected GPUs. Balogh et al. provide a comprehensive survey of parallelization approaches, languages and compilers for unstructured mesh algorithms on GPU architectures. In particular, they show improvements in performance for CUDA codes when using the Clang compiler over NVIDIA's own nvcc. Guillaume Aupy and colleagues exploit the periodic nature of I/O in HPC applications to develop efficient scheduling strategies. Using their scheduling strategy they demonstrate a 32% increase in throughput on the Mira system. Finally, Romero et al. document their porting of the PWscf code to multi-core and GPU systems decreasing time-to-solution by 2–3×.

Section B of the proceedings collates papers concerned with "Performance Modeling and Simulation." Nicolas Denoyelle et al. present the cache-aware roofline model (CARM) and validate the model on a Xeon Phi Knights Landing platform. Similarly, Chennupati et al. document a scalable memory model to enable CPU performance prediction. Mollah et al. examine universal globally adaptive load-balanced routing algorithms on the Dragonfly topology. Their performance model is able to accurately predict the aggregate throughput for Dragonfly networks. Cavelan et al. apply algorithm-based focused recovery (ABFR) to N-body computations. They compare this approach with the classic checkpoint/restart strategy and show significant gains over the latter. Zhang et al. propose a multi-fidelity surrogate modeling approach, using a combination of low-fidelity models (mini-applications) and a small number of high fidelity models (production applications) to enable faster application/architecture co-design cycles. They demonstrate an improvement over using either low-fidelity models or high-fidelity models alone. Finally, Simakov and colleagues document their development of a simulator of the Slurm resource manager. Their simulation is able to use historical logs to simulate different scheduling algorithms to identify potential optimizations in the scheduler.

The final section of the proceedings, Section C, contains the three short papers presented at PMBS. The paper by Yoga et al. discusses their extension to the Gen-Z communication protocol in the structural simulation toolkit, enabling source-code attribution tagging in network packets. Tyler Allen and colleagues at the Lawrence Berkley National Laboratory, conduct a performance and energy survey for NERSC workloads on Intel KNL and Haswell architectures. The final paper in this volume, by Turner and McIntosh-Smith, presents a survey of application memory usage on the ARCHER national supercomputer.

The PMBS 2017 workshop was extremely well attended and we thank the participants for the lively discussion and positive feedback received throughout the workshop. We hope to be able to repeat this success in future years.

The SC conference series is sponsored by the IEEE Computer Society and the ACM (Association for Computing Machinery). We are extremely grateful for the support we received from the SC 2017 Steering Committee, and in particular from Almadena Chtchelkanova and Luiz DeRose, the workshop chair and vice chair.

The PMBS 2017 workshop was only possible thanks to significant input from AWE in the UK, and from Sandia National Laboratories and the Lawrence Livermore

National Laboratory in the USA. We acknowledge the support of the AWE Technical Outreach Program (project CDK0724).

We are also grateful to LNCS for their support, and to Alfred Hofmann and Anna Kramer for assisting with the production of this issue.

November 2017

Stephen A. Jarvis
Steven A. Wright
Simon D. Hammond

# Organization

## Workshop Chairs

| | |
|---|---|
| Stephen Jarvis | University of Warwick, UK |
| Steven Wright | University of Warwick, UK |
| Simon Hammond | Sandia National Laboratories (NM), USA |

## Workshop Technical Program Committee

| | |
|---|---|
| Reid Atcheson | Numerical Algorithms Group Ltd., UK |
| Pavan Balaji | Argonne National Laboratory, USA |
| Prasanna Balaprakash | Argonne National Laboratory, USA |
| David Beckingsale | Lawrence Livermore National Laboratory, USA |
| Abhinav Bhatele | Lawrence Livermore National Laboratory, USA |
| Robert Bird | Los Alamos National Laboratory, USA |
| Richard Bunt | ARM Ltd., UK |
| Cristopher Carothers | Rensselaer Polytechnic Institute, USA |
| Patrick Carribault | CEA, France |
| Aurélien Cavelan | University of Basel, Switzerland |
| Raphaël Couturier | L'université Bourgogne, Franche-Comté, France |
| Todd Gamblin | Lawrence Livermore National Laboratory, USA |
| Wayne Gaudin | NVIDIA, UK |
| Paddy Gillies | European Centre for Medium-Range Weather Forecasts, UK |
| Jeff Hammond | Intel Corporation, USA |
| Andreas Hansson | ARM Ltd., UK |
| Andy Herdman | UK Atomic Weapons Establishment, UK |
| Thomas Ilsche | Technische Universität Dresden, Germany |
| Nikhil Jain | Lawrence Livermore National Laboratory, USA |
| Guido Juckeland | Helmholtz-Zentrum Dresden-Rossendorf, Germany |
| Michael Klemm | Intel Corporation, Germany |
| Andrew Mallinson | Intel Corporation, UK |
| Satheesh Maheswaran | UK Atomic Weapons Establishment, UK |
| Simon McIntosh-Smith | Bristol University, UK |
| Branden Moore | Sandia National Laboratores (NM), USA |
| Misbah Mubarak | Argonne National Laboratory, USA |
| Gihan Mudalige | University of Warwick, UK |
| Elmar Peise | AICES, RWTH Aachen, Germany |
| John Pennycook | Intel Corporation, USA |
| Karthik Raman | Intel Corporation, USA |
| István Reguly | Pázmány Péter Catholic University, Hungary |
| Jose Cano Reyes | University of Edinburgh, UK |

| | |
|---|---|
| Yves Robert | ENS Lyon, France |
| Stephen Roberts | ARM Ltd., UK |
| Arun Rodrigues | Sandia National Laboratories (NM), USA |
| Fabio Schifano | Università di Ferrara, Italy |
| Andrey Semin | Intel Corporation, Germany |
| Govind Sreekar Shenoy | University of Edinburgh, UK |
| Thomas Steinke | Zuse Institute Berlin, Germany |
| Peter Strazdins | Australian National University, Australia |
| Christian Trott | Sandia National Laboratories (NM), USA |
| Alejandro Valero | University of Zaragoza, Spain |
| Yunquan Zhang | Chinese Academy of Sciences, China |

# Contents

**Performance Evaluation and Analysis**

Evaluating On-Node GPU Interconnects for Deep Learning Workloads . . . . . 3
*Nathan R. Tallent, Nitin A. Gawande, Charles Siegel, Abhinav Vishnu, and Adolfy Hoisie*

Comparison of Parallelisation Approaches, Languages, and Compilers for Unstructured Mesh Algorithms on GPUs . . . . . . . . . . . . . . . . . . . . . . . . 22
*G. D. Balogh, I. Z. Reguly, and G. R. Mudalige*

Periodic I/O Scheduling for Super-Computers. . . . . . . . . . . . . . . . . . . . . . . . 44
*Guillaume Aupy, Ana Gainaru, and Valentin Le Fèvre*

A Performance Study of Quantum ESPRESSO's PWscf Code on Multi-core and GPU Systems . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
*Joshua Romero, Everett Phillips, Gregory Ruetsch, Massimiliano Fatica, Filippo Spiga, and Paolo Giannozzi*

**Performance Modeling and Simulation**

Modeling Large Compute Nodes with Heterogeneous Memories with Cache-Aware Roofline Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
*Nicolas Denoyelle, Brice Goglin, Aleksandar Ilic, Emmanuel Jeannot, and Leonel Sousa*

A Scalable Analytical Memory Model for CPU Performance Prediction. . . . . 114
*Gopinath Chennupati, Nandakishore Santhi, Robert Bird, Sunil Thulasidasan, Abdel-Hameed A. Badawy, Satyajayant Misra, and Stephan Eidenbenz*

Modeling UGAL on the Dragonfly Topology . . . . . . . . . . . . . . . . . . . . . . . 136
*Md Atiqul Mollah, Peyman Faizian, Md Shafayat Rahman, Xin Yuan, Scott Pakin, and Michael Lang*

Resilient N-Body Tree Computations with Algorithm-Based Focused Recovery: Model and Performance Analysis . . . . . . . . . . . . . . . . 158
*Aurélien Cavelan, Aiman Fang, Andrew A. Chien, and Yves Robert*

Multi-fidelity Surrogate Modeling for Application/Architecture Co-design . . . 179
*Yiming Zhang, Aravind Neelakantan, Nalini Kumar, Chanyoung Park, Raphael T. Haftka, Nam H. Kim, and Herman Lam*

A Slurm Simulator: Implementation and Parametric Analysis . . . . . . . . . . . . 197
*Nikolay A. Simakov, Martins D. Innus, Matthew D. Jones, Robert L. DeLeon, Joseph P. White, Steven M. Gallo, Abani K. Patra, and Thomas R. Furlani*

**Short Papers**

Path-Synchronous Performance Monitoring in HPC Interconnection Networks with Source-Code Attribution . . . . . . . . . . . . . . . . 221
*Adarsh Yoga and Milind Chabbi*

Performance and Energy Usage of Workloads on KNL and Haswell Architectures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 236
*Tyler Allen, Christopher S. Daley, Douglas Doerfler, Brian Austin, and Nicholas J. Wright*

A Survey of Application Memory Usage on a National Supercomputer: An Analysis of Memory Requirements on ARCHER. . . . . . . . . . . . . . . . . . . 250
*Andy Turner and Simon McIntosh-Smith*

**Author Index** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 261

# Performance Evaluation and Analysis

# Evaluating On-Node GPU Interconnects for Deep Learning Workloads

Nathan R. Tallent[1(✉)], Nitin A. Gawande[1], Charles Siegel[1], Abhinav Vishnu[1], and Adolfy Hoisie[2]

[1] Pacific Northwest National Laboratory, Richland, WA, USA
{nathan.tallent,nitin.gawande,charles.siegel,abhinav.vishnu}@pnnl.gov
[2] Brookhaven National Laboratory, Upton, NY, USA
ahoisie@bnl.gov

**Abstract.** Scaling deep learning workloads across multiple GPUs on a single node has become increasingly important in data analytics. A key question is how well a PCIe-based GPU interconnect can perform relative to a custom high-performance interconnect such as NVIDIA's NVLink. This paper evaluates two such on-node interconnects for eight NVIDIA Pascal P100 GPUs: (a) the NVIDIA DGX-1's NVLink 1.0 'hybrid cube mesh'; and (b) the Cirrascale GX8's two-level PCIe tree using dual SR3615 switch risers. To show the effects of a range of neural network workloads, we define a parameterized version of the popular ResNet architecture. We define a workload intensity metric that characterizes the expected computation/communication ratio; we also locate AlexNet and GoogLeNet within that space. As expected, the DGX-1 typically has superior performance. However, the GX8 is very competitive on all ResNet workloads. With 8 GPUs, the GX8 can outperform the DGX-1 on all-to-all reductions by 10% for medium-sized payloads; and in rare cases, the GX8 slightly outperforms on ResNet.

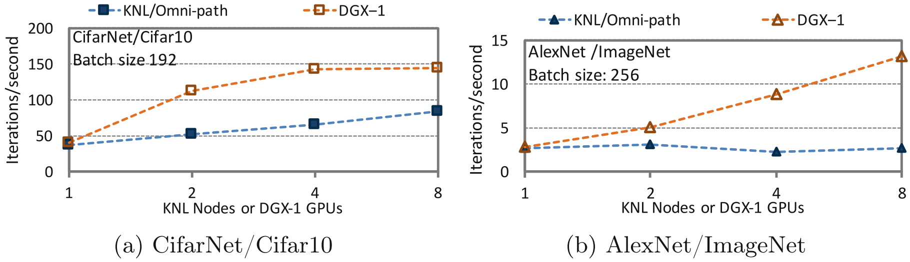
**Keywords:** GPU interconnects · NVIDIA DGX-1 · NVIDIA NVLink
Cirrascale SR3615 switch riser · Convolutional neural networks

## 1 Introduction

Scaling deep learning workloads across multiple GPUs has become increasingly important in data analytics. For example, strong scaling can reduce the training time of neural networks. Moreover to train deep networks on large data sets, it may be necessary to harness multiple GPU memories.

The inter-GPU network can dictate performance when scaling deep learning workloads across multiple GPUs. Figure 1 shows that scaling some workloads is impossible without a high-performance interconnect [1]. The figure shows strong scaling behavior of two well known workloads — CifarNet/Cifar10 and AlexNet/ImageNet — on an NVIDIA DGX-1 [2] and an Intel Knights Landing [3] (KNL) cluster. The DGX-1 uses an NVLink-based GPU interconnect. The KNL cluster interconnects KNL processors (1 per node) using Intel's Omni-Path. For each workload, the single-KNL/GPU performance is *very similar* — despite

© Springer International Publishing AG 2018
S. Jarvis et al. (Eds.): PMBS 2017, LNCS 10724, pp. 3–21, 2018.
https://doi.org/10.1007/978-3-319-72971-8_1




(a) CifarNet/Cifar10 (b) AlexNet/ImageNet

**Fig. 1.** Performance scaling of (a) CifarNet/Cifar10 and (b) AlexNet/ImageNet on an NVIDIA DGX-1 and an Intel KNL/Omni-Path cluster.

the GPU's higher peak floating point rate. However, scaling behavior is quite different. Although both workloads perform better over NVLink than Omni-Path, the qualitative scaling trends are different. With NVLink, the AlexNet workload (Fig. 1b) scales better than the CifarNet one (Fig. 1a). With Omni-Path, the qualitative scaling performance is *inverted*: scaling is better with CifarNet than AlexNet. The reason is that AlexNet's much larger all-to-all reduction operations (allreduce) place a much higher stress on interconnect bandwidth. Omni-Path, designed as a cluster interconnect, has a per-node (uni-directional) bandwidth of 12.5 GB/s whereas the DGX-1's NVLink supports up to 80 GB/s per GPU.

Because GPU interconnect performance can be a bottleneck when scaling deep learning workloads, some computing vendors are creating products to enable scalable GPU computing on a single densely populated node. A key question is how well a PCIe-based GPU interconnect can perform relative to a custom high-performance interconnect such as NVIDIA's NVLink. Unfortunately, it is difficult for data scientists to quantify the potential of these different products. In particular, Fig. 1 shows that a high-performance interconnect *may not* be critical to scaling. The interconnect's importance depends significantly on a workload's characteristics, including total work and effective communication to computation ratio.

This paper evaluates two recent GPU interconnects (Sect. 2) for eight NVIDIA Pascal P100 GPUs on a single node: (a) the NVIDIA DGX-1's 'hybrid cube mesh' based on NVLink 1.0; and (b) the Cirrascale GX8's [4] two-level PCIe tree using two Cirrascale SR3615 switch risers.

We evaluate the two interconnects on a parameterized neural network workload (Sect. 3). The performance scaling of a parameterized neural network space has not been well studied. Other performance evaluations select specific networks — for example AlexNet [5] and GoogLeNet [6] — that have been designed for classifier performance, not workload evaluation. We define a parameterized variant of the popular ResNet [7] with controllable computational and communication intensities. With our parameterized ResNet, we show the effects of different neural network topologies and batch sizes on a workload's communication/computation ratio and scaling behavior. We define a workload intensity

metric to characterize space of workload intensities and locate AlexNet and GoogLeNet within that space.

Our findings (Sect. 4) are as follows. The workload intensity metric in helpful in explaining scaling behavior. Given that the DGX-1's NVLink interconnect has more links and higher per-link bandwidth than the GX8's PCIe bus, it is not surprising that the DGX-1 typically has superior performance. However, we find that the GX8 is very competitive for all ResNet-style workloads; in rare cases, the GX8 slightly outperforms. Surprisingly, with 8 GPUs, the GX8 can outperform the DGX-1 on an allreduce benchmark by as much as 10% on payloads between 0.5–6 MB. In contrast, with 4 GPUs the DGX-1 allreduces outperform the GX8 by 40%. The reason is that with 8 GPUs, the PCIe network saturates more quickly with respect to payload size. The DGX-1 has a distinct scaling advantage for the communication-intensive AlexNet where we hypothesize that load imbalance enables its NVLink interconnect to perform closer to the 4 GPU bandwidths than 8, resulting in a 36% DGX-1 advantage.

## 2 Multi-GPU Computing Systems

This section describes the NVIDIA DGX-1 (Pascal) [2] and the Cirrascale GX8 (NVIDIA Pascal) [4] computing systems and then explains the test configuration. To isolate the interconnects, we configured the systems as closely as possible except for GPU interconnect.

Each system has a very similar host processor configuration. Both systems have a dual-processor host based on Intel Xeon processors. For the DGX-1, each processor is an Intel Xeon E5-2698v4; for the GX8, it is an E5-2697v4. The DGX-1's Xeon has 20 cores, two threads enabled per core, running at 2.2/3.6 GHz; and a 50 MB L3 cache, a 256 KB L2 cache shared between two cores, and 64 KB L1 cache per core. The GX8's Xeon has 18 cores with 2 thread/core, running at 2.3/3.6 GHz; L3 45 MB. In both cases, host memory is 512 GB DDR4-2133. Both systems use PCIe 3.0.

All important workload activities (e.g., neural network training) occurs on the GPUs. The primary work the host CPU performs is reading the initial training data set into memory and transferring it to the GPUs. Both systems read the large training inputs files from a local SSD whose throughput is sufficient to overlap training and reading.

### 2.1 NVIDIA P100 Pascal

Both systems have eight NVIDIA Tesla P100 (Pascal) GPUs. To isolate the interconnects, we configured the systems with the closest possible GPUs: Tesla P100-SXM2 and P100-PCIE-16GB. The DGX-1 has the former and the Cirrascale the latter. The only P100 available with NVLink support is the P100-SXM2; and because of NVLink support it uses a different form factor (SXM2). The P100-PCIE-16GB is the 'highest bin' P100 available with the PCIe 3.0 $\times$ 16 interface. The only differences between the two P100s — besides NVLink and form factor — are SM clock speed (1328 vs. 1189 MHz) and TDP (300 vs. 250 W).

Pascal GPUs are fabricated with a 16 nm process. Each GPU has 3584 CUDA cores divided into 56 streaming multiprocessors (SM), where each SM has 64 CUDA cores. The P100-PCIE-16GB has a peak FP performance of 9.3 Teraflops single precision (4.67 Teraflops double). Due to the higher clock rate, the P100-SXM2 has a peak FP performance of 10.6 Teraflops single precision (5.3 Teraflops double). Each GPU has 16 GB high-bandwidth global memory (HBM2), a 4096-bit memory bus operating at 715 MHz (split into 8 memory controllers), and 4 MB L2 cache.

**Normalizing GPU Performance.** Given the different GPUs, it is necessary to distinguish the performance effects of the varying GPU clocks from the different interconnects. One possibility is normalizing or scaling GPU performance post facto. This approach is difficult with fixed clocks; and more difficult with dynamically boosted clocks. Rather than attempting this approach, we power-capped both GPUs. The obvious approach is to cap both GPU variants at the nominal frequency of the P100-PCIE-16GB, 1189 MHz. To present results as close to the P100-SXM2 as possible, we found the maximum sustained frequency of the P100-PCIE-16GB for a representative workload. That is, we empirically identified the maximum frequency for the P100-PCIE-16GB to execute without throttling. Based on this study, we capped both GPUs at 1227 MHz, which closes the gap by 27%. With this experimental setup, we expect the performance of each GPU to be identical. The GPU performance is still sufficiently high to highlight the scaling effects of each interconnect.

### 2.2 NVIDIA DGX-1 and NVLink 1.0

Figure 2 shows the DGX-1's intra-node interconnect topology [2]. Each GPU's SXM2 interface, in contrast to the more conventional PCIe interface, connects directly to the NVLink interconnect. The NVLink interconnect enables intra-node GPU communication. Each GPU has 4 NVLink lanes arranged in a 'hybrid cube mesh' topology. The hybrid cube mesh has two directly connected groups of 4 along with 3D hypercube links between the groups. The topology ensures that a GPU is no more than two hops away from another GPU.

Each of the 4 NVLink lanes supports 20 GB/s in both directions. Thus, the total NVLink uni-directional bandwidth of a GPU is 80 GB/s. Each GPU also connects via a PLX-switch to a PCIe 3.0 $\times$ 16 bus with maximum bandwidth of 16 GB/s (uni-directional). This PLX switch serves as a connecting point between GPUs and CPUs, and a potential InfiniBand network.

### 2.3 Cirrascale GX8 and SR3615 Switch

The Cirrascale GX8 [4] system supports direct communication between 8 GPUs using two Cirrascale SR3615 switch risers [8]. Communication occurs over the PCIe bus, enabling a single memory address space.

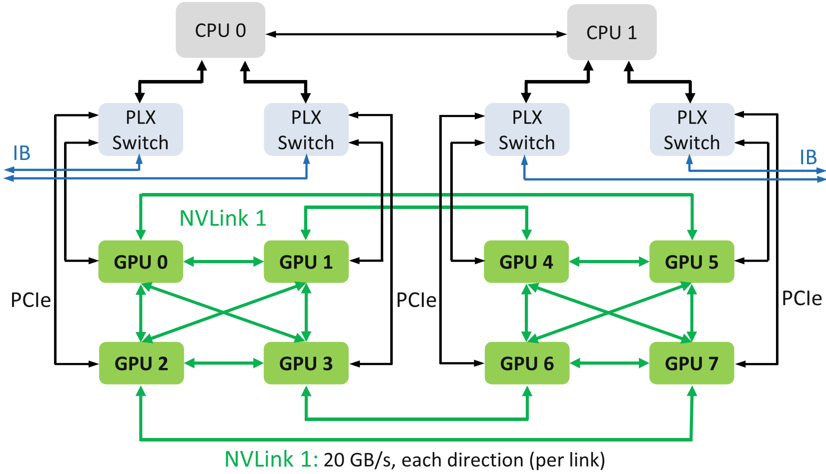



**Fig. 2.** Inter-GPU network on NVIDIA DGX-1.

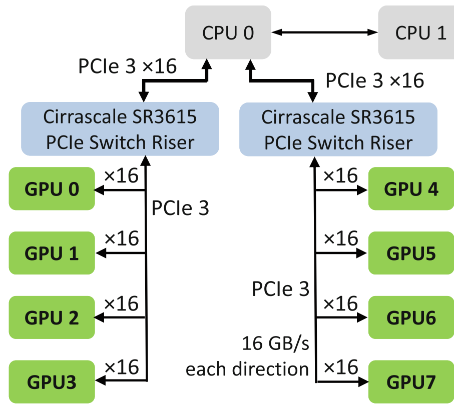



**Fig. 3.** Inter-GPU network on Cirrascale GX8.

Figure 3 shows the GX8's inter-GPU network. To enable communication over a single PCIe bus (and hence single memory address space), the GX8 uses a tree topology rooted at only *one* of the host CPUs [9]. The two-level tree is rooted at one host's on-die PCIe controller, a.k.a. the root complex, supporting PCIe $3.0\times40$. Attached to that host CPU are two SR3615 switch risers. Each SR3615's upstream is PCIe $3.0 \times 16$ (16 GB/s uni-directional). Two risers consume 32/40 lanes of the root complex. Communication between the SR3615s occurs via the root complex using the standard PCIe bus.

Four P100s are attached to each SR3615 switch riser. Each GPU (P100-PCIE-16GB) has a PCIe $3.0 \times 16$ interface. Thus, each switch riser's input is 64 PCIe

lanes of GPU; and 16 out. As a result there is a peak uni-directional 16 GB/s (PCIe 3.0 × 16) between any two GPUs.

Because of the SR3615 switch, communication paths do not all need to traverse the root complex. A pair of GPUs attached to different risers traverse two switches and the PCIe root complex. However, a pair of GPUs attached to the same switch require no intermediate paths.

### 2.4 Inter-GPU Communication

For inter-GPU (peer-to-peer) communication, we use a combination of CUDA 8.0 and the NVIDIA Collective Communications Library (NCCL). CUDA 8.0 includes support for GPUDirect, or GPU-to-GPU direct memory access (DMA). NCCL [10,11] is a library for inter-GPU collective communication and synchronization. NCCL's collective algorithms are based on topology-aware rings and optimized for throughput [12,13]. NCCL is interconnect-aware and thus the same collective call uses, as appropriate, the NVLink or PCIe interconnect. Available collectives include allgather, allreduce, and broadcast.

To achieve high throughput on large payloads, NCCL's algorithms are pipelined based on small 4–16 KB chunks and GPUDirect peer-to-peer direct access. With large payloads, pipelining hides the linear latency term of the ring resulting in transfer bandwidths approaching link bandwidth [14]. However, for small messages, the ring latency is exposed.

## 3 Workloads

In this paper, we develop a systematic approach for characterizing and specifying neural network workloads. To explore the effects of different neural network topologies and batch sizes on scaling behavior, we define a parameterized variant of the popular ResNet [7] with controllable computational and communication intensities. We complement our study with results from the well known AlexNet [5] and GoogLeNet [6]. The subsections below describe each CNN architecture. After each network is described, we characterize the space of workload intensities and locate AlexNet and GoogLeNet within that space.

Each distinct neural-network training workload executes in the following manner. First, a given neural network architecture is replicated on each GPU. Then, the neural network is trained, processing an image dataset sequentially in batches or *iterations*. For each batch, images are divided among available GPUs for data parallelism. To train, each GPU processes its images resulting in a series of model activations — floating point operations — resulting in distinct values for each GPU's copy of model parameters. At the end of each iteration, allreduce operations ensure each GPU's model has an identical copy of model parameters.

For all workloads, we use the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [15], a well known benchmark for object classification and detection. Specifically, we use ILSVRC2012 which has 1000 object classes and 1.43 M images annotated images, each of size 256 × 256.

### 3.1 AlexNet

AlexNet [5] uses the ImageNet (ILSVRC2012) [15] dataset. Compared to non-deep learning methods, AlexNet has performed well on ILSVRC2012. AlexNet has five convolution layers, three pooling layers, and two fully-connected layers. This CNN architecture requires about 1.4 M activations/image and has 60 M parameters.

### 3.2 GoogLeNet

GoogLeNet [6] is more complex model than AlexNet. GoogLeNet has two convolution layers, two pooling layers, and nine inception layers. Each inception layer consists of six convolution layers and one pooling layer. The concept of inception layer is to cover bigger area of images while maintaining fine resolution for small information on these images. The inception module of GoogLeNet concatenates filters of different sizes into a single new filter. This avoids parameter explosion with the use of inception layers. GoogLeNet performs significantly better than AlexNet for the ImageNet and the recent ILSVRC [15] challenge datasets. This CNN architecture has about 5.5 M parameters. GoogLeNet in relation to AlexNet has (i) more layers; (ii) fewer features per layer, and; (iii) more activations. GoogLeNet has 10.8 M activations per image.

### 3.3 ResNet/$x$

Deep Residual Learning Network (ResNet) [7] introduced the concept of a residual block. Each block consists of two convolution layers along with a connection adding the output of the second block to the input of the first. Residual blocks are designed to allow the training of substantially deeper models than had been trained previously. By adding the input of the block to its output, the residual block learns the *residual* function, and forwards the activations to deeper layers than earlier. One advantage of ResNet is that it can improve accuracy of the model while avoiding parameter explosion. That is, the ResNet blocks increase the depth (and inner layers) of the network instead of its width.

Using residual blocks as a fundamental building block, several ResNet incarnations have been designed by researchers, including ResNet50 and ResNet1000. ResNets of various depths outperform GoogLeNet on the ILSVRC challenge, with a 50 layer network — consisting of a convolutional layer, 48 residual blocks, and a classifier layer — winning in 2015.

To explore the effects of different ResNet networks, we generate several ResNet variants by defining each network's inner layers to be a multiple of a 'ResNet block'. This enables us to explore how neural network topology and training batch size affects its communication/computation ratio and scaling. We define ResNet/$x$ to be a standard ResNet input and output layer but where the inner layers are defined by $x$ replications of the 'ResNet block'. Thus, ResNet/1 is a single convolution layer followed by a residual block and finally a classifier layer. Similarly, ResNet/16 has the same convolution and classifier layers as ResNet/1

but 16 residual blocks. Using this parameterized definition, we can explore the different computation and communication ratios by simply increasing the depth of residual blocks.

Each ResNet block has a certain number of features. As a result, increasing ResNet blocks proportionally increases activations/image and model parameters. More precisely, activations/image as a function of the block replications $x$ is given with the following expression: $1,204,224x + 11,55,113$. Similarly model parameters as a function of replications is given by $46,211x + 74,857$. Thus, our ResNet/$x$ models have the activations/image and parameters shown in Fig. 4.

| $x$ | **Activations** | **Parameters** |
|---|---|---|
| 1 | 2.4M | 121K |
| 2 | 3.6M | 167K |
| 4 | 6.0M | 260K |
| 8 | 10.8M | 445K |
| 16 | 20.4M | 814K |
| 32 | 39.7M | 1,554K |

**Fig. 4.** Activations and parameters for ResNet/$x$.

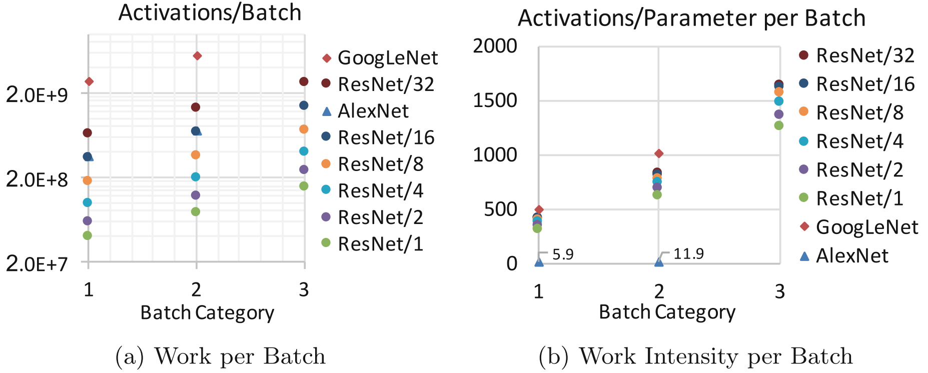
### 3.4 Workload Characterization

Figure 5 overviews the workloads we used in our study. To leverage well-known, verified, and optimized implementations of convolutional neural networks (CNN), we based our experiments on Convolutional Architecture for Fast Feature Embedding (Caffe) framework [16,17], a widely used framework for CNN models. Caffe is a collection of state-of-the-art deep learning algorithms and reference models in a clean and modifiable framework accessible through a open source repository [18].

| **CNN** | **Dataset** | **Caffe Model** | **Batch Sizes** |
|---|---|---|---|
| AlexNet | ImageNet | bvlc_AlexNet | 256, 512 |
| GoogLeNet | ImageNet | bvlc_GoogLeNet | 256, 512 |
| ResNet/$x$ | ImageNet | custom ResNet, $x \in \{1, 2, 4, 8, 16, 32\}$ | 16, 32, 64 |

**Fig. 5.** CNN architecture models and input datasets.

Figure 6 characterizes each workload's batch properties using metrics representing work and work intensity. Figure 6a shows activations per batch, a measure of total GPU work. The horizontal axis refers to the batch categories in Fig. 5. (AlexNet and GoogLeNet each have two categories while ResNet/$x$ has three.)




(a) Work per Batch (b) Work Intensity per Batch

**Fig. 6.** Each workload's (a) work and (b) work intensity (work/communication).

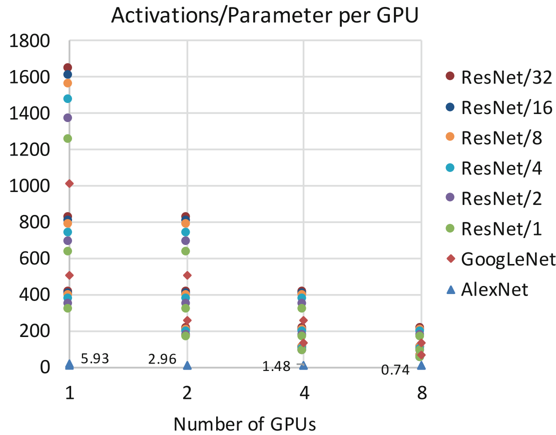



**Fig. 7.** Each workload's intensity (work/communication) during strong scaling.

Observe the large spread of work shown along the vertical axis (independent of the horizontal axis). The points densely cover over two orders of magnitude, specifically between 38 M and 5,500 M activations/batch.

Next we characterize *work intensity*, a measure of the ratio of communication to computation. Figure 6b shows activations per parameter for each batch, a measure of the batch's work intensity. We capture well over two orders of magnitude of intensities, between 6–1650 activations/parameter. Our ResNet/$x$ parameter sweep densely covers the space between 300–1650; and it sandwiches GoogLeNet.

Finally, we characterize each execution's *work intensity*. For each performance experiment, the batch's work is strong-scaled across 1, 2, 4 or 8 GPUs. Figure 7 shows activations per parameter for each GPU, a measure of the communication/computation ratio during execution. We capture well over three orders

of magnitude of intensities, between 1–1650 activations per parameter per GPU. Our ResNet/$x$ parameter sweep densely covers most of the space (between 40–1650); again, it sandwiches GoogLeNet.

# 4 Evaluation

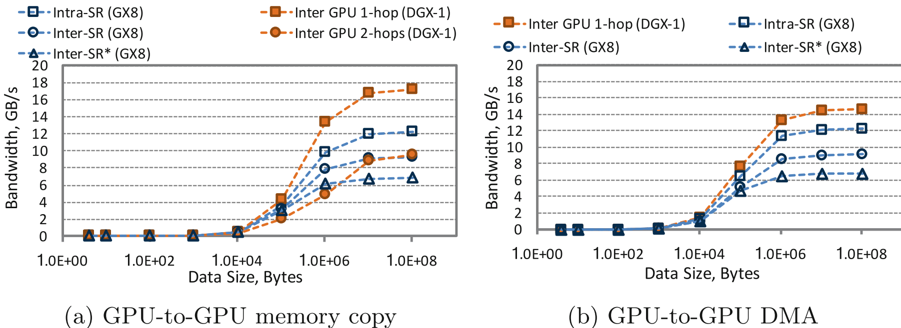
We conduct a performance evaluation using strong scaling to highlight effects of interconnect performance. Strong scaling is often desirable to reduce response time. With strong scaling, the amount of available per-GPU work systematically decreases, increasing the communication to computation ratio. In contrast to strong scaling, weak scaling tends to mask performance effects of weaker interconnects.

We used NVIDIA's optimized Caffe, a fork from BVLC-Caffe [18] optimized for the DGX-1 architecture [19]. For AlexNet and GoogLeNet, we used NVIDIA's provided models. For ResNet/$x$, we defined custom versions. We confirmed that all executions produced semantically meaningful results in that the models were equivalent to a sequentially equivalent execution.

We present our results in four subsections. The first two subsections discuss microbenchmarks for inter-GPU copies and NCCL collectives. We then show scaling results for AlexNet and GoogLeNet. Finally we discuss ResNet/$x$.

## 4.1 Inter-GPU Data Transfer

We used MGBench [20] to collect bandwidths and latencies between pairs of GPUs for GPU-to-GPU memory copy and GPU-to-GPU DMA (direct memory access).




(a) GPU-to-GPU memory copy (b) GPU-to-GPU DMA

**Fig. 8.** Bandwidth of GPU-to-GPU memory copy for DGX-1 and GX8.

Figure 8a shows bandwidths between pairs of GPUs for GPU-to-GPU memory copy. (Units are in power of 2, or GiB.) This unidirectional GPU-to-GPU memory copy is pipelined using CUDA's asynchronous memory-copy primitive.

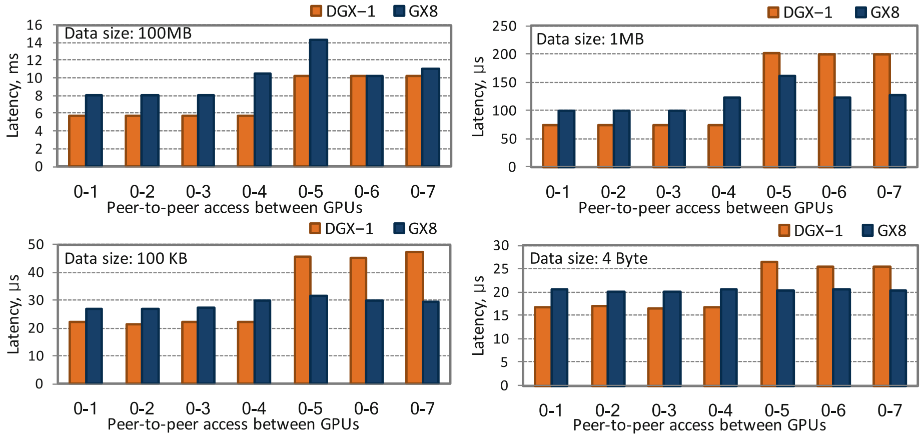



**Fig. 9.** Latency of GPU-to-GPU memory copy for DGX-1 and GX8.

Rather than showing the full matrix for all pairs, we group the results by value clusters, where each group has an insignificant spread.

Figure 9 shows latencies of GPU-to-GPU memory copy highlighted at four different data sizes. A horizontal axis label of $x$-$y$ means GPU $x$ sent data to GPU $y$. Although the figure shows data with GPU 0 as source, we validated that using other GPUs as source produced qualitatively similar results.

For both figures, the DGX-1 results are typically clustered in two groups, one representing a single NVLink hop and the other representing two NVLink hops. The *one-hop* data corresponds to communication within a fully-connected 4-GPU cluster; achieved bandwidth is about 85% (17.2 GB/s) of the 20 GB/s per-link peak. The *two-hop* data corresponds to communication between 4-GPU clusters; achieved bandwidth is about 50% (9.6 GB/s) of the peak.

The GX8 results are clustered in three groups. The groups are clearly seen in the latency plots (Fig. 9) for payload sizes 1 MB and above. The first two groups, *Intra-SR* and *Inter-SR*, correspond to communication within and between an SR3615 switch riser (SR), respectively. These groups are analogous to DGX-1 groups in that each SR forms a fully connected 4-GPU cluster. The Intra-SR achieved bandwidth is about 75% (12.2 GB/s) of peak (16 GB/s). The Inter-SR group includes GPUs 4, 6 and 7; achieved bandwidth is about 60% (9.6 GB/s) of peak. The third *Inter-SR** group captures the anomaly of sending data from GPU 0 to 5. It turns out that the second logical PCIe slot (GPU5) has longer physical signal paths between some elements on the ASIC which can lead to delays in dequeuing PCIe packets [21]. The circumstances in which these delays occur are narrow and more likely to originate within a microbenchmark than a real world application. For example, we do not observe the behavior in collective benchmarks.

Interestingly, the GX8 can have better bandwidth and latencies between GPUs that are in different 4-GPU-clusters. Compare Fig. 8a's Inter-SR and

Inter-SR* GX8 curves with the 2-hop DGX-1 curve. The GX8's PCIe bandwidth saturates more quickly with respect to message size, resulting in higher GX8 bandwidth at medium-sized messages. Figure 9 shows the same effect in terms of latency for message sizes 1 MB and below. In contrast, but as is expected, within a fully connected 4-GPU-cluster, NVLink's bandwidth and latency are better than PCIe.

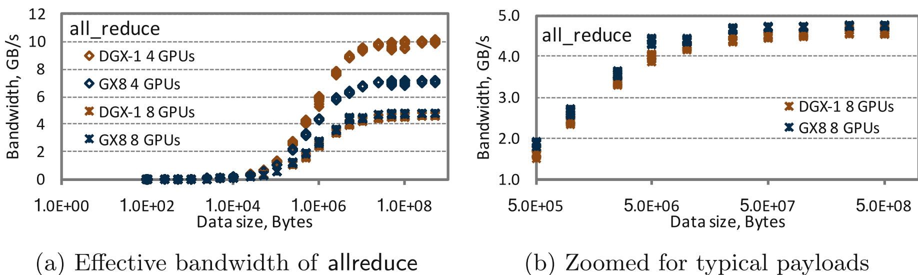
Another interesting comparison is that although NVLink latencies are very predictable, GX8 latencies are dependent on message size. The NVLink latencies fall into two clusters — one-hop and two-hops — independent of data size. In contrast, the GX8 latencies fall into 1 or 3 clusters depending on data size. For very small payloads (4 bytes), the GX8 latencies are flat and independent of hops. This shows the PCIe switching latency — even across multiple hops — is very low. The 1-cluster phenomenon largely holds true at 100 KB. By 1 MB, the GX8 results are clustered into the three groups described above. We hypothesize that the reason that small messages appear to be independent of topology — in contrast to NVLink — is related to PCIe switch buffering that effectively enables pipelining of smaller messages across multiple PCIe switches.

Figure 8b shows GPU-to-GPU bandwidths for DMA (direct memory access). The DMA data closely corresponds to the memory copy data. For DGX-1, we only shows single-hop DMA bandwidth because CUDA 8.0 does not support GPUDirect across 2 NVLink hops. In contrast, DMA is supported over a single PCIe bus.

### 4.2 Inter-GPU Collectives

Figure 10 shows *effective bandwidth* of NCCL allreduce, the key collective used in training. (Bandwidths are in power of 2, or GiB.) These results characterize allreduce performance given perfect GPU load balance. The size of allreduce payloads is the neural network's parameters represented as single precision floating points, or $4 \times \textit{parameters}$ bytes. Thus for ResNet/$x$, payloads range from 0.5–6 MB and for GoogLeNet and AlexNet they are 22 MB and 240 MB, respectively.

We define effective bandwidth to be relative to a *single* GPU's payload, i.e., 1-GPU-payload/runtime. With this metric, the ideal value is relative to the




(a) Effective bandwidth of allreduce (b) Zoomed for typical payloads

**Fig. 10.** Effective bandwidth for NCCL allreduce on DGX-1 and GX8.

bandwidth of one link. For example, in a fully connected 4-GPU NVLink cluster, the ideal allreduce is performed in 1 step where each GPU concurrently utilizes 3 links, yielding an effective bandwidth of 1 link, or 20 GB/s. Similarly, 8-GPU allreduce requires in the best case one more step, meaning the maximum effective bandwidth is halved to 10 GB/s. Three more considerations imply that an allreduce's effective bandwidth should be less than ideal. As already observed, achievable link bandwidth for a direct copy is about 85% of ideal. Further, each GPU must execute the reduction operator. Finally, there is synchronization overhead.

Figure 10a shows that the NCCL allreduce is implemented well; there does not appear to be appreciable optimization headroom. The following observations explain. Within a fully connected 4-GPU-cluster, an allreduce's maximum effective bandwidth on DGX-1 and GX8 is 10 and 7 GB/s, respectively. For both systems, these values are about 60% of achievable bandwidth; they are also higher than *copying* data between inter GPU-clusters. This implies that NCCL's allreduces are effectively using a one-step algorithm. Between fully connected GPU-clusters, an allreduce's maximum effective bandwidth is 4.6 and 4.7 vs GB/s for DGX-1 and GX8, respectively. Given the extra hop, we assume that the maximum achievable bandwidth is half of the single link transfer bandwidth, or 8.6 and 6.1 GB/s. The above effective bandwidths are about 50% and 75% of maximum.

Interestingly, Fig. 10a shows that the relative performance of allreduce depends on number of GPUs. For 4 GPUs, the DGX-1 has a 40% performance advantage for large messages (such as those used for AlexNet). For 8 GPUs, between fully connected GPU clusters, the GX8 has 3% better performance for large messages.

Figure 10b highlights allreduce's effective bandwidth on 8 GPUs for the payloads encountered in our ResNet/$x$ workloads. The figure shows that the performance divergence between 0.5 and 5 MB payloads averages 10% in favor of the GX8. Clearly, as observed in GPU-to-GPU copies (Sect. 4.1), PCIe bandwidth saturates more quickly with respect to payload size. We expect that PCIe switching hardware is part of the explanation.

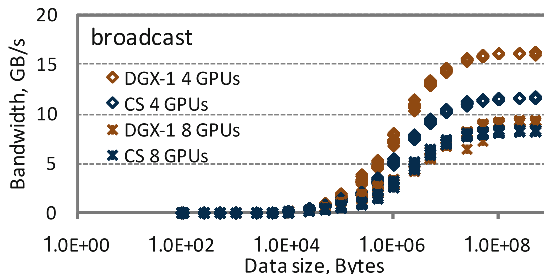
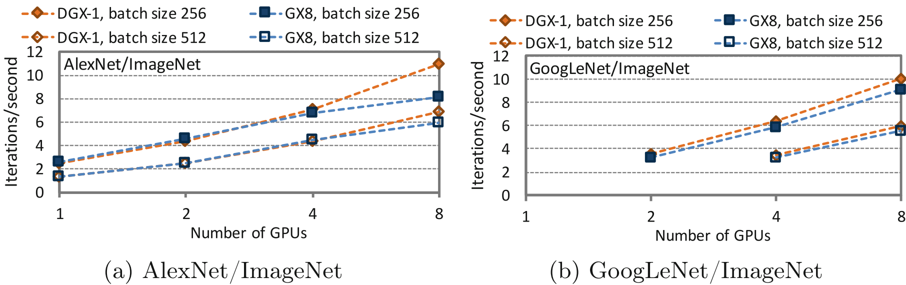



**Fig. 11.** Bandwidth for NCCL broadcast collective on DGX-1 and GX8.

Finally, we observe that performance varies depending on collective. Collectives have two costs: data transmission and synchronization. An allreduce must synchronize with all GPUs using *all-to-all* synchronization. This synchronization attenuates the NVLink's potential advantages. We would therefore expect a single-root collective such as broadcast, where GPUs synchronize only with the root GPU, to have a difference performance profile. Figure 11 shows that on 8 GPUs, the DGX-1 does have slightly higher effective bandwidth.

### 4.3 Strong Scaling of AlexNet and GoogLeNet

Figure 12 shows strong-scaling performance for AlexNet and GoogLeNet training on the DGX-1 and GX8. We collected results using two different batch sizes (256 and 512 images) on 1, 2, 4, and 8 GPUs. Although the batch sizes would be considered large for a single GPU, they are not large when scaling to 8 GPUs. Caffe data-parallelism distributes the images in each batch. Thus, with 256 batch size and 8 GPUs, there are 32 images per GPU.




(a) AlexNet/ImageNet (b) GoogLeNet/ImageNet

**Fig. 12.** Strong-scaling (ImageNet): AlexNet and GoogLeNet on DGX-1 and GX8.

Recall that we power cap GPUs to equalize the slightly different SM frequencies between the P100 SXM2 and PCIe variants. Therefore we expect both systems to have same single-GPU performance. As shown in Fig. 12a, this expectation is true for AlexNet. GoogLeNet's results (Fig. 12b) have data points missing for 1 and 2 GPUs. The reason is limited GPU memory capacity. With 1 and 2 GPUs, there was not enough memory to store both the GoogLeNet activations and the training data.

The most interesting results is that NVLink is far more important for AlexNet scaling than for GoogLeNet: for AlexNet the DGX-1 has a 36% advantage (11.0 vs. 8.1 iterations/s). It is more difficult than it might seem to explain the much higher DGX-1 performance on 8 GPUs. On one hand, as shown in Fig. 7, AlexNet's activations/parameter per GPU is very small: past 4 GPUs, the metric is less than 1, meaning the workload is communication intensive. However, as noted in Sect. 4.2, the GX8 has slightly *higher* performance for 8 GPUs on an allreduce benchmark. Validating the root cause is difficult because of the limited

value of GPU performance tools. Further more, to show the best performance results, we use NVIDIA's (read-only) Docker version of Caffe, which cannot be instrumented.

On GoogLeNet, the benefit of NVLink is comparatively small. As shown by Fig. 6b, GoogLeNet is more compute intensive (in activations/parameter) by almost a factor of 100. Whereas AlexNet's intensities are 5.9 and 11.9 per batch category, GoogLeNet's are 500 and 1004, respectively.

Finally, NVLink becomes less important as batch size increases. This is not surprising as a larger batch size increases the per-GPU computation without changing communication, therefore reducing the importance of the interconnect.

### 4.4 Strong Scaling of ResNet/$x$

Figure 13 shows strong-scaling performance for ResNet/$x$ on the DGX-1 and GX8. We use smaller batch sizes for ResNet than with AlexNet or GoogLeNet. From a learning perspective, ResNet tends to use smaller batch sizes. Because of the deep network, a smaller batch size yields more updates per training epoch, which affects convergence. Also, the activations' memory consumption means

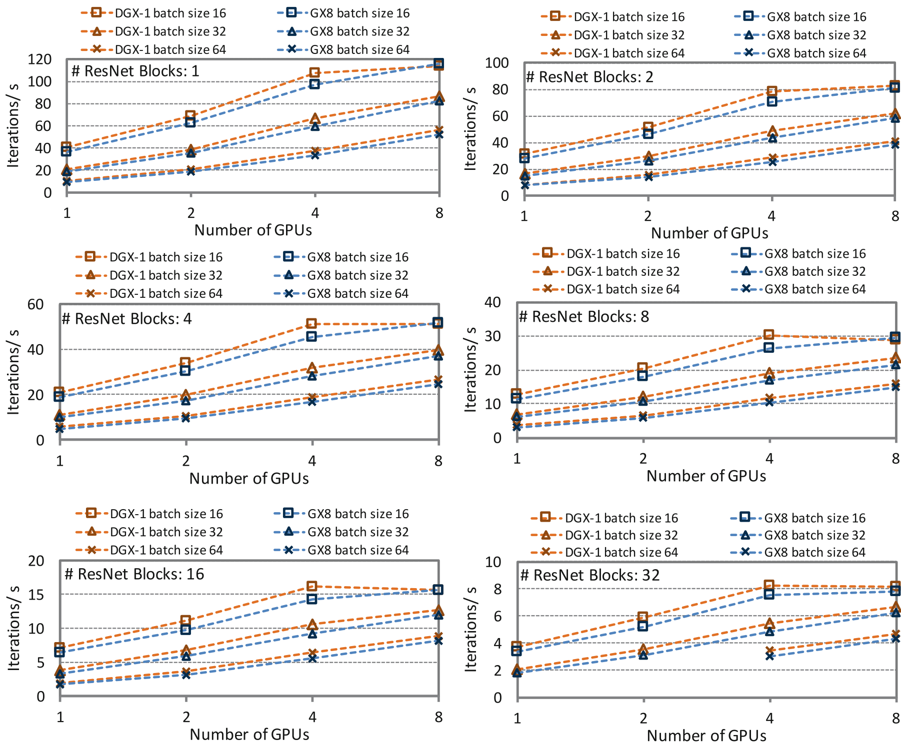



**Fig. 13.** Strong-scaling (ImageNet): ResNet/$x$ on DGX-1 and GX8.

larger network sizes will not fit in GPU memory. Observe that with ResNet/32, batch size 64 would not fit in the memory of either 1 or 2 GPUs.

Furthermore, the smaller batch sizes highlight GPU interconnect effects. Our custom ResNet/$x$ has many fewer model parameters than either AlexNet or GoogLeNet, yielding smaller allreduce payloads and reducing the effects of inter-GPU communication and synchronization. We therefore compensate by using smaller batch sizes. The smaller batch sizes results in less per-GPU work, maintaining pressure on interconnect.

First, we discuss single-GPU performance. As before, we expect both systems to have same single-GPU performance given the power capping to equalize SM frequencies. Curiously, we see single-GPU performance converging as batch size increases. Thus, although the expectation holds true for batch size 64, there is a small divergence for batch sizes 16 and 32. We are not sure how to explain the divergence but have identified two possible factors.

One factor could be that for each batch of images, there is some CPU-based images processing overhead. For ResNet/$x$, this processing includes random horizontal flips, random crops, and subtraction of mean values to center distributions at 0. For smaller batch sizes, there is a potential this overhead can be exposed.

A second factor is host-CPU-based operations such as memory operations (e.g., `cudaMemset` and `cudaFree`) and scattering batch images to GPUs. This operations would occur over each system's PCIe bus. Although both host-to-GPU PCIe busses are PCIe $\times 16$ (16 GB/s), benchmarks show that there are slight differences in performance. For instance, for a host-to-GPU0 scatter, host-GPU0 communication on DGX-1 consistently yields about 4% higher bandwidth (11.1 vs. 10.7 GB/s). Again, this overhead could be exposed for smaller batch sizes.

We next sketch a simple model of our performance expectations. With equivalent GPU performance, each workload has an identical GPU work cost. The expected overall DGX-1 performance advantage is therefore the workload's fraction of communication multiplied by the DGX-1's allreduce performance advantage. Given the DGX-1's slightly better allreduce performance on 4 GPUs for ResNet/$x$ payloads, our model points to better DGX-1 performance for 2 and 4 GPUs. That expectation holds true in general. Given the GX8's better allreduce performance on 8 GPUs, the model suggests the 'knee' that appears in the DGX-1 curves for batch size 16. Recall that on 8 GPUs, the GX8 averages 10% better allreduce performance for the payloads used in ResNet/$x$ (0.5–5 MB); see Sect. 4.2.

Our model does a good job explaining scaling results through 4 GPUs and the DGX-1 'knee' for batch 16. However, for batch sizes 32 and 64 on 8 GPUs, the DGX-1 consistently outperforms the GX8. Clearly, other effects must be taken into account to fully explain the scaling results.

We conclude by observing that similar performance trends hold true for a very large range of workload intensities, or activations/parameter per GPU (Fig. 7). These data show that if one is interested in ResNet-style workloads, the GX8 may be an attractive option if there is enough price differential.

## 5 Related Work

An important aspect of this work is defining ResNet/$x$, a paramerized version of ResNet. To our knowledge, there is no prior study that systematically parameterizes a deep learning workload to explore its space of computational intensities. Other performance evaluations select specific networks that have been designed for classifier performance, not workload evaluation. Even deep learning benchmark suites such as Fathom [22] represent only several points instead of a (discrete) continuum. Conversely, several studies assert general benefits [2,23] of NVLink but do not look at the conditions under which one should or should not expect benefits.

Multi-GPU systems (or nodes) are becoming increasingly important. We have found no study comparing NVLink and PCIe-based GPU interconnects for up to 8 GPUs. Our comparison of the DGX-1/NVLink and GX8/Cirrascale SR3615 is relevant because both systems represent the 'top-tier' of multi-GPU systems but are also generally available.

Shams et al. [24] compare performance of Caffe using AlexNet for up to 4 P100 GPUs with and without NVLink. They show unexpected differences in the performance of AlexNet even with the use of only one GPU. Also, that study does not explain the effect of NVLink and the network topology using microbenchmarks.

Nomura et al. [25] study performance of a multi-GPU system connected by PCIe. They show significant speedup on 4 GPUs for applications of particle motion and advection computation. They showed that data transfer becomes a bottleneck even with relatively low computation.

Ben-Nun et al. [26] present the Groute asynchronous multi-GPU programming model and show nearly 7× speedup for some algorithms on a 8-GPU heterogeneous system. Awan et al. present MVAPICH2-GDR [27], an inter/intra-node multi-GPU collective library. They compare NVIDIA NCCL and MVAPICH2-GDR using microbenchmarks and a DNN training application. Their proposed design of MVAPICH2-GDR showed to have enabled up to 14× to 16.6× improvements as compared to NCCL-based solutions, for intra-/inter-node multi-GPU communication.

## 6 Conclusions

Scaling ML workloads across multiple on-node GPUs is becoming increasingly important. A closely related question is whether PCIe-based interconnects are adequate. We have provided a detailed performance evaluation of two GPU-based interconnects for eight NVIDIA Pascal P100 GPUs: (a) NVIDIA DGX-1 (NVLink 1.0 with 'hybrid cube mesh' topology); and (b) Cirrascale GX8 (two-level PCIe tree using two Cirrascale SR3615 switch risers). To systematically study the scaling effects of different neural networks, we define a parameterized variant of the popular ResNet [7] with controllable model activations and parameters.

To characterize the workload space, we defined a workload intensity metric that captures the expected computation/communication ratio and has good explanatory power. We show that our parameterized ResNet captures a large space of workload intensities.

Our conclusions are as follows. We find that the DGX-1 typically has superior performance. Given that the DGX-1's NVLink interconnect has more links and higher per-link bandwidth than the GX8's PCIe bus, this is not surprising. However, we also find that the GX8 is very competitive for all ResNet-style workloads. In rare cases, the GX8 slightly outperforms. The reason is related to the fact that the GX8's PCIe bandwidth saturates more quickly with respect to payload size. As a result, for medium-sized messages, the GX8 on 8 GPUs can have better memory copy latency and an average of 10% better allreduce performance. Our results shows that if one is interested in ResNet-style workloads, the GX8 may be an attractive option if there is enough price differential.

**Acknowledgments.** The authors thank Matthew Macduff (PNNL) for evaluation assistance. We are grateful for funding support from the U.S. Department of Energy's (DOE) Office of Advanced Scientific Computing Research as part of the "Center for Advanced Technology Evaluation" (CENATE) and "Convergence of Deep Learning and Machine Learning for HPC Simulation and Modeling." Pacific Northwest National Laboratory is operated by Battelle for the DOE under Contract DE-AC05-76RL01830.

## References

1. Gawande, N.A., Landwehr, J.B., Daily, J.A., Tallent, N.R., Vishnu, A., Kerbyson, D.J.: Scaling deep learning workloads: NVIDIA DGX-1/Pascal and Intel Knights Landing. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 399–408, May 2017
2. Foley, D., Danskin, J.: Ultra-performance Pascal GPU and NVLink interconnect. IEEE Micro **37**(2), 7–17 (2017)
3. Sodani, A., Gramunt, R., Corbal, J., Kim, H.S., Vinod, K., Chinthamani, S., Hutsell, S., Agarwal, R., Liu, Y.C.: Knights landing: second-generation Intel Xeon Phi product. IEEE Micro **36**(2), 34–46 (2016)
4. Cirrascale: The GX8 series multi-device peering platform, June 2016. http://www.cirrascale.com/documents/datasheets/Cirrascale_GX8Series_Datasheet_CM080C.pdf
5. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. In: Pereira, F., Burges, C., Bottou, L., Weinberger, K. (eds.) Advances in Neural Information Processing Systems, vol. 25, pp. 1097–1105. Curran Associates, Inc., Red Hook (2012)
6. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going deeper with convolutions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1–9 (2015)
7. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778 (2016)
8. Cirrascale: Cirrascale SR3615 PCIe switch riser, July 2015

9. Cirrascale: Scaling GPU compute performance (2015). http://www.cirrascale.com/documents/whitepapers/Cirrascale_ScalingGPUCompute_WP_M987_REVA.pdf
10. Luehr, N.: Fast multi-GPU collectives with NCCL, April 2016. https://devblogs.nvidia.com/parallelforall/fast-multi-gpu-collectives-nccl/
11. NVIDIA: NCCL: NVIDIA collective communications library, August 2017. https://developer.nvidia.com/nccl
12. Awan, A.A., Hamidouche, K., Venkatesh, A., Panda, D.K.: Efficient large message broadcast using NCCL and CUDA-aware MPI for deep learning. In: Proceedings of the 23rd European MPI Users' Group Meeting, EuroMPI 2016, New York, NY, USA, pp. 15–22. ACM (2016)
13. Jeaugey, S.: Optimized inter-GPU collective operations with NCCL, May 2017. http://on-demand-gtc.gputechconf.com/gtc-quicklink/8Bdyh
14. Patarasuk, P., Yuan, X.: Bandwidth optimal all-reduce algorithms for clusters of workstations. J. Parallel Distrib. Comput. **69**(2), 117–124 (2009)
15. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A.C., Fei-Fei, L.: Imagenet large scale visual recognition challenge. Int. J. Comput. Vis. **115**(3), 211–252 (2015)
16. Berkeley Vision and Learning Center. Berkeley vision and learning center: Caffe (2016). http://caffe.berkeleyvision.org
17. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: Convolutional architecture for fast feature embedding. arXiv preprint arXiv:1408.5093 (2014)
18. Berkeley Vision and Learning Center: Convolutional architecture for fast feature embedding (Caffe) (2016). https://github.com/BVLC/caffe/
19. NVIDIA: Convolutional architecture for fast feature embedding (Caffe) (2017). https://github.com/NVIDIA/caffe
20. Ben-Nun, T.: MGBench: multi-GPU computing benchmark suite, February 2016. https://github.com/tbennun/mgbench
21. Cirrascale: Cirrascale SR3514: Unexpected performance inequality. Technical Brief M901A–092014
22. Adolf, R., Rama, S., Reagen, B., Wei, G.Y., Brooks, D.: Fathom: reference workloads for modern deep learning methods. In: 2016 IEEE International Symposium on Workload Characterization (IISWC), pp. 1–10, September 2016
23. Christensen, C., Fogal, T., Luehr, N., Woolley, C.: Topology-aware image compositing using NVLink. In: 2016 IEEE 6th Symposium on Large Data Analysis and Visualization (LDAV), pp. 93–94, October 2016
24. Shams, S., Platania, R., Lee, K., Park, S.J.: Evaluation of deep learning frameworks over different HPC architectures. In: Proceedings of the IEEE 37th International Conference on Distributed Computing Systems, pp. 1389–1396, June 2017
25. Nomura, S., Mitsuishi, T., Suzuki, J., Hayashi, Y., Kan, M., Amano, H.: Performance analysis of the multi-GPU system with expether. SIGARCH Comput. Archit. News **42**(4), 9–14 (2014)
26. Ben-Nun, T., Sutton, M., Pai, S., Pingali, K.: Groute: an asynchronous multi-GPU programming model for irregular computations. In: Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, New York, NY, USA, pp. 235–248. ACM (2017)
27. Awan, A.A., Chu, C.H., Subramoni, H., Panda, D.K.: Optimized broadcast for deep learning workloads on dense-GPU infiniband clusters: MPI or NCCL? arXiv preprint arXiv:1707.09414 (2017)

# Comparison of Parallelisation Approaches, Languages, and Compilers for Unstructured Mesh Algorithms on GPUs

G. D. Balogh[1(✉)], I. Z. Reguly[1], and G. R. Mudalige[2]

[1] Faculty of Information Technology and Bionics, Pazmany Peter Catholic University, Budapest, Hungary
balogh.gabor.daniel@hallgato.ppke.hu, reguly.istvan@itk.ppke.hu
[2] Department of Computer Science, University of Warwick, Coventry, UK
g.mudalige@warwick.ac.uk

**Abstract.** Efficiently exploiting GPUs is increasingly essential in scientific computing, as many current and upcoming supercomputers are built using them. To facilitate this, there are a number of programming approaches, such as CUDA, OpenACC and OpenMP 4, supporting different programming languages (mainly C/C++ and Fortran). There are also several compiler suites (clang, nvcc, PGI, XL) each supporting different combinations of languages. In this study, we take a detailed look at some of the currently available options, and carry out a comprehensive analysis and comparison using computational loops and applications from the domain of unstructured mesh computations. Beyond runtimes and performance metrics (GB/s), we explore factors that influence performance such as register counts, occupancy, usage of different memory types, instruction counts, and algorithmic differences. Results of this work show how clang's CUDA compiler frequently outperform NVIDIA's nvcc, performance issues with directive-based approaches on complex kernels, and OpenMP 4 support maturing in clang and XL; currently around 10% slower than CUDA.

**Keywords:** Compilers · CUDA · OpenACC · OpenMP · GPU Benchmarking

## 1 Introduction

The last ten years has seen the widespread adoption of Graphical Processing Units (GPUs) by the high performance computing community. For a wide range of highly parallel workloads they offer higher performance and efficiency. Programming techniques for GPUs have also evolved significantly. The CUDA [1] language extensions to C/C++ and the OpenCL language [2] provide a low-level programming abstraction commonly referred to as Single Instruction Multiple Thread (SIMT) that gives fine-grained control over GPU architectures. CUDA/OpenCL allows the exploitation of low-level features like scratch pad

© Springer International Publishing AG 2018
S. Jarvis et al. (Eds.): PMBS 2017, LNCS 10724, pp. 22–43, 2018.
https://doi.org/10.1007/978-3-319-72971-8_2

memory, warp operations, and block-level synchronization. However, converting existing applications to use CUDA or OpenCL is a substantial undertaking that require significant effort and considrable changes to the design of the programe and the source code. Furthermore, getting good performance can entail detailed work in orchestrating parallelism.

To simplify the adoption of GPUs, particularly for existing codes, high-level directive based programming abstractions were introduced. OpenACC [3] introduced in 2011 was one of the first supporting GPUs. Subsequetly OpenMP standard introduced support for accelerators starting from version 4 [4], with refinements in 4.5 and 5.0. Of particular note is that the evolution of directive based approaches being driven by the acquisition of large US DoE systems such as Titan and the upcoming Summit and Sierra systems. To be able to efficiently utilize these systems it was necessary that existing codes be modified to support GPUs with relative ease. Many of these codes are written in Fortran and as such there is now compiler support for writing CUDA, OpenACC, and OpenMP with Fortran in various compilers.

It is generally agreed that the best performance can be achieved by using CUDA, but the difference between CUDA and directive-based approaches vary significantly based on a multitude of factors. Primarily these include the type of computation being parallelized, as well as the language being used (C or Fortran), and the compiler. This motivates the present study: for a number of parallel loops, coming from the domain of unstructured mesh computations, we wanted to get an idea of what performance looks like on different GPUs, different languages, and different compilers. Given the available systems and compilers, we would like to acertain what the state-of-th-art is with regard to utilizing GPU based systems for this class of applications.

We evaluate some of the most commonly used compilers and parallelization approaches. We explore the performance of CUDA C, compiled with nvcc, as well as with Google's recent clang based compiler [5]. We also explore the performance of the compilers by Portland Group (PGI, now owned by NVIDIA) which has had support for wirting CUDA applications in Fortran [6,7]. Additionally, as part of a recent push by IBM, preparing for the Summit and Sierra machines there has been support for CUDA Fortran with the XL compilers since v15.1.5 [8]. We also explore XL compiler performance in this paper. For OpenACC we use the PGI compilers which support both C and Fortran. There is also good support for OpenACC by the Cray compilers, however we did not have access to such a machine and therefore will not be part of this analysis. For OpenMP 4 there are two compilers developed by IBM directed at developing applications using C: the XL compilers (since v13.1.5), and an extension to Clang [9]. There is also support for writing OpenMP 4 parallizations in Fortran applications using the XL compilers (since v15.1.5).

While there is a tremendous amount of research on performance evaluation of various combinations of languages and compilers, we believe our work is unique in its breadth: it directly compares C and Fortran implementations of the same code (Airfoil), and with three different parallelizations: CUDA, OpenACC, and

OpenMP, and with five different state-of-the-art compilers. We also present an in-depth study trying to explain the differences with the help of instruction counters and the inspection of low-level code. Specifically, we make the following contributions:

1. Using a representative CFD application called Airfoil, we run the same algorithms on NVIDIA K40 and P100 GPUs, with CUDA, OpenMP 4, and OpenACC parallelizations written in both C and Fortran, compiled with a number of different compilers.
2. We carry out a detailed analysis of the results with the help of performance counters to help identify differences between algorithms, languages, and compilers.
3. We evaluate these parallelizations and compilers on two additional applications, Volna (C) and BookLeaf (Fortran) to confirm the key trends and differences observed on Airfoil.

The rest of the paper is structured as follows: Sect. 2 discusses some related work, Sect. 3 briefly introduces the applications being studied, then Sect. 4 presents the test setup, compilers and flags. Section 5 carries out the benchmarking of parallelizations and the detailed analysis, and finally Sect. 6 draws conclusions.

## 2 Related Work

There is a significant body of existing research on performance engineering for GPUs, and compiler engineering, as well as some comparisons between parallelization approaches - the latter however is usually limited in scope due to the lack of availability of multiple implementations of the same code. Here we cite some examples, to show how this work offers a wider look at the possible combinations.

Work by Ledur et al. compares a few simple testcases such as Mandelbrot and N-Queens implemented with CUDA and OpenACC (PGI) [10], Herdman et al. [11] take a larger stencil code written in C, and study CUDA, OpenCL and OpenACC implementations, but offer no detailed insights into the differences. Work by Hoshino et al. [12] offers a detailed look at CUDA and OpenACC variants of a CFD code and some smaller benchmarks written in C, and show a few language-specific optimizations, but analysis stops at the measured runtime. Normat et al. [13] compare CUDA Fortran and OpenACC versions of an atmospheric model, CAM-SE, which offers some details about code generated by the PGI and Cray compilers, and identifies a number of key differences that let CUDA outperform OpenACC, thanks to lower level optimizations, such as the use of shared memory. Kuan et al. [14] also compare runtimes of CUDA and OpenACC implementations of the same statistical algorithm (phylogenetic inference). Gonge et al. [15] compare CUDA Fortran and OpenACC implementations of Nekbone, and scale up to 16k GPUs on Titan - but no detailed study of performance differences.

Support in compilers for OpenMP 4 and GPU offloading is relatively new [16] and there are only a handful of papers evaluating their performance: Martineau et al. [17] present some runtimes of basic computational loops in C compiled with Cray and clang, and comparisons with CUDA. Karlin et al. [18] port three CORAL benchmark codes to OpenMP 4.5 (C), compile them with clang, and compare them with CUDA implementations - the analysis is focused on runtimes and register pressure. Hart el. al. [19] compare OpenMP 4.5 with Cray to OpenACC on Nekbone, however the analysis here is also restricted to runtimes, the focus is more on programmability. We are not aware of academic papers studying the performance of CUDA Fortran or OpenMP 4 in the IBM XL compilers aside from early results in our own previous work [20]. There is also very little work on comparing the performance of CUDA code compiled with nvcc and clang.

Thus we believe that there is a significant gap in current research: a comparison of C and Fortran based CUDA, OpenACC, and OpenMP 4, the evaluation of the IBM XL compilers, the maturity of OpenMP 4 compared to CUDA in terms of performance and a more detailed investigation into the reasons for the performance difference between various languages, compilers, and parallelization approaches. With the present study, we work towards filling this gap.

## 3 Applications

The applications being studied in this work come from the unstructured mesh computations domain solving problems in the areas of computational fluid dynamics, shallow-water simulation and Lagrangian hydrodynamics. As such, they consist of parallel loops over some set in the mesh, such as edges, cells or nodes, and on each set element some computations are carried out, while accessing data either directly on the iteration set, or indirectly via a mapping to another set. Our applications are all written using the OP2 domain specific language [21] embedded in C and Fortran, targeting unstructured mesh computations. For OP2, the user has to give a high level description of the simulation using the OP2 API. Then the OP2 source-to-source translator generates all parallelized versions from the abstract description [22]. While OP2 is capable of many things, its relevant feature for this work is that it can generate different parallelizations such as CUDA, OpenACC, and OpenMP4, based on the abstract description of parallel loops.

A key challenge in unstructured mesh computations is the handling of race conditions when data is indirectly written. For the loops with indirect increments (which means we incrementing some value through a mapping so there are multiple iterations incrementing the same value), we use coloring to ensure that no two threads will write to the same memory at the same time. We can use a more sophisticated coloring approach for GPUs using CUDA as described in [23], where we create and color mini-partitions such that no two mini-partitions of the same color will update the same cell. This allows mini-partitions of the same color to be processed by the blocks of one CUDA kernel. Within these mini-partitions, each assigned to a different CUDA thread block, each thread

will process a different element within these blocks, and thus is it necessary to introduce a further level of coloring. For an edges to cells mapping, we color all edges in a mini-partition so that no two edges with the same color update the same cell. Such a coloring is shown in Fig. 1. Here, we first calculate the increment of every thread in the block, then we iterate through the colors and add the increment to the cell with synchronization between each color. The benefit of such an execution scheme is that there is a possibility that the data we loaded from the global memory can be reused within a block, which can lead to a performance increase due to fewer memory transactions. This technique is referred to as hierarchical coloring in the paper.

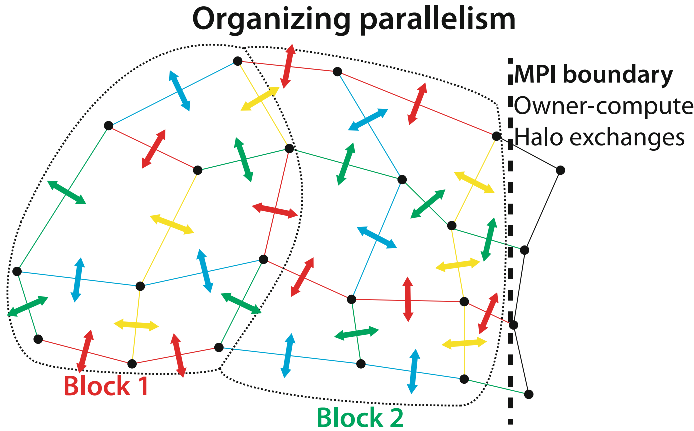



**Fig. 1.** Illustration for hierarchical coloring on a computation on edges that write data on the cells. The blocks are colored so that there is no neighboring blocks with the same color and inside the blocks threads colored so that no two threads with the same color write the same data.

With other methods such as OpenACC and OpenMP4 there is no method for thread synchronization and data sharing in blocks, which is essential for the hierarchical coloring technique described above. Therefore a global coloring technique is used in case of these parallelization approaches. This technique is similar to the thread coloring inside the mini-partitions, but works on the full set. We assign colors to each thread in a way that no two edges of the same color update the same cell and threads from the same color can run parallel in a separate CUDA kernel with synchronization between the kernels. This however excludes the possibility of the reuse of the data of the cells.

### 3.1 Airfoil

Airfoil is a benchmark application, representative of large industrial CFD applications. It is a non-linear 2D inviscid airfoil code that uses an unstructured grid and a finite-volume discretisation to solve the 2D Euler equations

using a scalar numerical dissipation. The algorithm iterates towards the steady state solution, in each iteration using a control volume approach, meaning the change in the mass of a cell is equal to the net flux along the four edges of the cell, which requires indirect connections between cells and edges. Airfoil is implemented using OP2, where two versions exists, one implemented with OP2's C/C++ API and the other using OP2's Fortran API [21,24].

The application consists of five parallel loops: **save_soln**, **adt_calc**, **res_calc**, **bres_calc** and **update** [22]. The **save_soln** loop iterates through cells and is a simple loop accessing two arrays directly. It basically copies every four state variables of cells from the first array to the second one. The **adt_calc** kernel also iterates on cells and it computes the local area/timestep for every single cell. For the computation it reads values from nodes indirectly and writes in a direct way. There are some computationally expensive operations (such as square roots) performed in this kernel. The **res_calc** loop is the most complex loop with both indirect reads and writes; it iterates through edges, and computes the flux through them. It is called 2000 times during the total execution of the application and performs about 100 floating-point operations per mesh edge. The **bres_calc** loop is similar to **res_calc** but computes the flux for boundary edges. Finally **update** is a direct kernel that includes a global reduction which computes a root mean square error over the cells and updates the state variables.

All test are executed with double precision on a mesh containing 2.8 million cells and with SOA data layout described in [22].

### 3.2 Volna

Volna is a shallow water simulation capable of handling the complete life-cycle of a tsunami (generation, propagation and run-up along the coast) [25]. The simulation algorithm works on unstructured triangular meshes and uses the finite volume method. Volna is written in C/C++ and converted to use the OP2 library [21]. For Volna we examined the top three kernels where most time is pent: **computeFluxes**, **SpaceDiscretization** and **NumericalFluxes**. In the **computeFluxes** kernel there are indirect reads and direct writes, in **NumericalFluxes** there are indirect reads with direct writes and a global reduction for calculating the minimum timestep and in **SpaceDiscretization** there are indirect reads and indirect increments.

Tests are executed in single precision, on a mesh containing 2.4 million triangular cells, simulating a tsunami run-up to the US pacific coast.

### 3.3 BookLeaf

BookLeaf is a 2D unstructured mesh Lagrangian hydrodynamics application from the UK Mini-App Consortium [26]. It uses a low order finite element method with an arbitrary Lagrangian-Eulerian method. Bookleaf is written entirely in Fortran 90 and has been ported to use the OP2 API and library. Bookleaf has a large number of kernels with different access patterns such as

indirect increments similar to increments inside **res_calc** in Airfoil. For testing we used the SOD testcase with a 4 million element mesh. We examined the top five kernels with the highest runtimes which are **getq_christiensen1**, **getq_christiensen_q**, **getacc_scatter**, **gather**, **getforce_visc**. Among these there is only one kernel (**getacc_scatter**) with indirect increments (where coloring is needed), the **gather** and **getq_christiensen1** have indirect reads and direct writes as **adt_calc** in Airfoil, and the other two kernels have only direct reads and writes.

# 4 Test Setup

For testing we used NVIDIA K40 and P100 GPUs in IBM S824L systems (both systems has 2*10 cores) with Ubuntu 16.04. We used nvcc in CUDA 9.0 and clang 6.0.0 (r315446) for compiling CUDA with C/C++. For compiling CUDA Fortran, we used PGI 17.4 compilers and IBM's XL compiler 15.1.6 beta 12 for Power systems. For OpenMP4, we tested clang version 4.0.0 (commit 6dec6f4 from the clang-ykt repo), and the XL compilers (13.1.6 beta 12). Finally, for OpenACC, we used the PGI compiler version 17.4. The specific compiler versions and flags are shown in Table 1.

**Table 1.** Compiler flags used on K40 GPU (for P100 cc60 and sm_60 is used)

| | Version | Flags |
|---|---|---|
| PGI | 17.4-0 | -O3 -ta=nvidia,cc35 -Mcuda=fastmath<br>-Minline=reshape (-acc for OpenACC) |
| XL | 15.1.6 beta 12<br>13.1.6 beta 12 | -O3 -qarch=pwr8 -qtune=pwr8 -qhot<br>-qxflag=nrcptpo -qinline=level=10<br>-Wx,-nvvm-compile-options=-ftz=1<br>-Wx,-nvvm-compile-options=-prec-div=0<br>-Wx,-nvvm-compile-options=-prec-sqrt=0<br>(-qsmp=omp -qthreaded -qoffload for OpenMP4) |
| clang for OpenMP4 | 4.0 | -O3 -ffast-math -fopenmp=libomp -Rpass-analysis<br>-fopenmp-targets=nvptx64-nvidia-cuda<br>-fopenmp-nonaliased-maps -ffp-contract=fast |
| clang for CUDA | 6.0 | -O3 –cuda-gpu-arch=sm_35 -ffast-math |
| nvcc | 9.0.176 | -O3 -gencode arch=compute_35,code=sm_35<br>–use_fast_math |

# 5 Benchmarking

## 5.1 Airfoil

The run times of different versions of Airfoil on the K40 and P100 GPUs are shown in Fig. 2. The hierarchical coloring is used in **res_calc** and **bres_calc**,

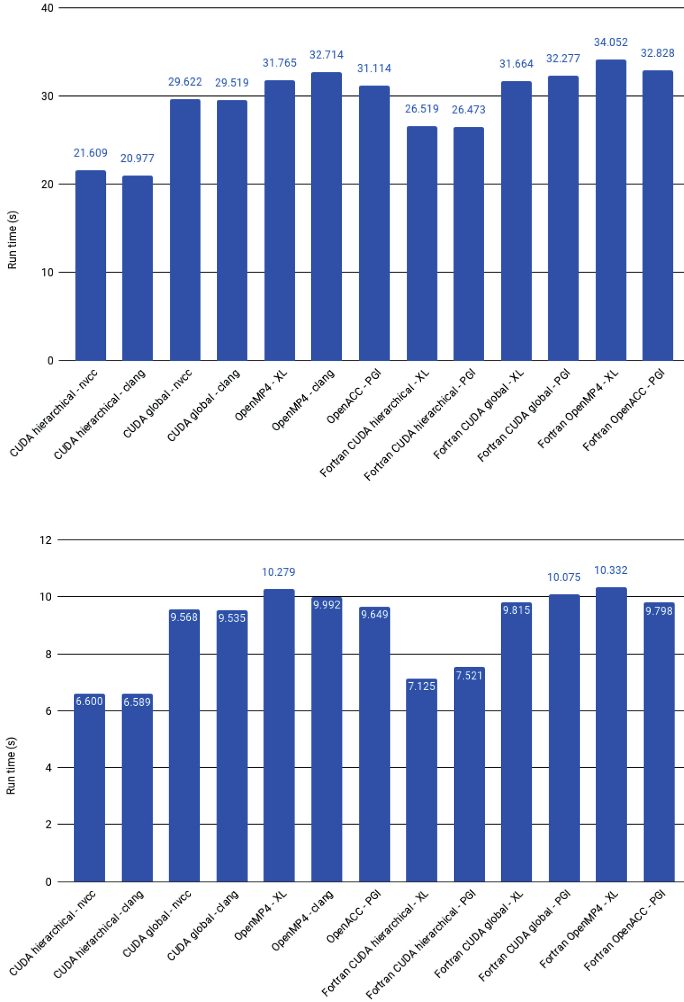



**Fig. 2.** Measured run times of versions on the K40 and P100 GPU

because these have indirect increments and in the case of other kernels we don't need coloring because they have only direct updates. The versions using the hierarchical coloring scheme have the best performance, due to the huge performance gains in **res_calc** thanks to data reuse. The main differences between versions with the same coloring strategy is in the run times of the **res_calc** and **adt_calc** kernels, where most of the computation is performed. In the following, we examine performance in detail on all five kernels.

**save_soln**: **save_soln** is a really simple kernel with only direct reads and writes. It copies state variables of the cells, and thus is highly memory bounded. In CUDA versions we used 200 blocks so each thread processes more than one cell

to save on integer instructions. However this leads us to a for loop inside the kernel, increasing control instructions, and slowing performance. In Table 2 the runtimes of the **save_soln** kernel are shown: all versions have approximately the same performance. The bandwidth values shown in the table are the useful bandwidth from the users perspective, that is the sum of the moved simulation data and mappings for the kernel divided by the run time of the kernel. In case of C/C++, OpenMP4 and OpenACC versions have about 5–7% better runtimes and bandwidth than CUDA versions (even though the OpenMP4 version compiled with clang on the K40 GPU is the only version that have only 75% occupancy). If we run one thread per cell and delete the loop from the kernel the performance of CUDA matches the performance of OpenMP4. The results shows that in a simple case such as **save_soln** Fortran performs about as well as the C/C++ versions, and the high level approaches such as OpenMP4 and OpenACC can reach the performance of CUDA.

**Table 2.** Measured run time, bandwidth, register count and occupancy values in case of **save_soln**

| | K40 | | | P100 | | |
|---|---|---|---|---|---|---|
| | Run time (s) | BW (GB/s) | Reg. count (Occupancy) | Run time (s) | BW (GB/s) | Reg. count (Occupancy) |
| nvcc - CUDA | 1.055 | 175 | 21 (100%) | 0.362 | 509 | 24 (100%) |
| clang - CUDA | 1.055 | 175 | 21 (100%) | 0.362 | 509 | 24 (100%) |
| PGI - OpenACC | 1.006 | 183 | 26 (100%) | 0.351 | 526 | 29 (100%) |
| XL - OpenMP4 | 1.003 | 184 | 17 (100%) | 0.357 | 517 | 19 (100%) |
| clang - OpenMP4 | 0.982 | 188 | 35 (75%) | 0.356 | 518 | 32 (100%) |
| PGI - F_CUDA | 1.061 | 174 | 32 (100%) | 0.368 | 502 | 32 (100%) |
| PGI - F_OpenACC | 1.012 | 182 | 24 (100%) | 0.349 | 528 | 29 (100%) |
| XL - F_CUDA | 1.060 | 174 | 32 (100%) | 0.362 | 502 | 32 (100%) |
| XL - F_OpenMP4 | 1.009 | 183 | 22 (100%) | 0.352 | 524 | 24 (100%) |

**adt_calc**: In case of **adt_calc** the loop iterates over cells and reads data indirectly from the nodes while updating a single value per cell multiple times. The operation contains some expensive square root calculations which introduce high numbers of additional floating point operations and increase the register counts for the kernel. For **adt_calc** the directive based approaches use significantly higher numbers of registers than CUDA as shown in Table 3; this means lower occupancy and about 30% worse performance on the K40 machine (on the P100 machine the difference is only about 10–20%). In case of OpenMP4 with the XL compiler and OpenACC with the PGI compiler every time the value on the cell is written we see a global store instruction instead of calculating the intermediate results in registers and only write the final results to the global memory as other versions do. Another source of performance difference for OpenMP4 with clang compiler comes from the lack of usage of texture caches for loading the

read only data from the nodes. The cause of clang CUDA slightly outperforming nvcc CUDA on the K40 machine is that it computes the expensive square root operations with fewer floating point instructions, which leads to about 16% less floating point instructions than nvcc (this also holds for the P100 card) for **adt_calc**. The Fortran versions have high register counts, thus lower occupancy, this is one of the key reasons for the 30% lower performance on the K40 GPU (on the P100 GPU the difference is about 10–15%), also Fortran versions use about 50% more integer instructions than C/C++ versions. The directive based approaches perform within 20% of CUDA Fortran's performance and the with the PGI compiler the versions execute twice as many integer instructions than other versions.

**Table 3.** Measured run time, bandwidth, register count and occupancy values in case of **adt_calc**

| | K40 | | | P100 | | |
|---|---|---|---|---|---|---|
| | Run time (s) | BW (GB/s) | Reg. count (Occupancy) | Run time (s) | BW (GB/s) | Reg. count (Occupancy) |
| nvcc - CUDA | 2.810 | 148 | 40 (75%) | 0.869 | 477 | 40 (75%) |
| clang - CUDA | 2.756 | 151 | 36 (75%) | 0.867 | 478 | 40 (75%) |
| PGI - OpenACC | 4.071 | 102 | 86 (31.25%) | 0.978 | 424 | 96 (31.25%) |
| XL - OpenMP4 | 3.775 | 110 | 64 (50%) | 0.984 | 421 | 72 (43.75%) |
| clang - OpenMP4 | 4.108 | 101 | 88 (31.25%) | 1.077 | 385 | 96 (31.25%) |
| PGI - F_CUDA | 3.753 | 116 | 64 (50%) | 0.955 | 434 | 56 (56.25%) |
| PGI - F_OpenACC | 4.341 | 96 | 86 (31.25%) | 1.053 | 394 | 96 (31.25%) |
| XL - F_CUDA | 3.581 | 116 | 78 (37.5%) | 1.001 | 415 | 88 (31.25%) |
| XL - F_OpenMP4 | 3.905 | 106 | 80 (37.5%) | 1.090 | 380 | 86 (31.25%) |

**res_calc**: In the case of **res_calc** we have indirect updates, therefore we need coloring to avoid race conditions. The runtime and bandwidth results are shown in Table 4 for hierarchical coloring, and for global coloring in Table 5. In this kernel there is a lot of indirectly read and written data, therefore the runtime can be significantly improved with the hierarchical coloring approach due to data reuse. However, hierarchical coloring leads to higher register counts and arithmetic instruction counts, but the impact of these factors are smaller than the gain from better memory usage. As we saw it in **adt_calc** for CUDA versions clang performs better than nvcc in terms of integer and floating point instruction counts (clang has 2–5% lower instruction counts on both GPUs). The OpenMP4 and OpenACC versions have 2–5% higher run time because of low occupancy (caused by register pressure) and the OpenMP4 versions don't use the texture caches as much (or at all in case of clang) as other versions which lead to 3 times as much global loads and high number of integer instructions. The results are shown in Tables 6 and 7.

**Table 4.** Measured run time, bandwidth, register count and occupancy values of **res_calc** in case of hierarchical coloring

| | K40 | | | P100 | | |
|---|---|---|---|---|---|---|
| | Run time (s) | BW (GB/s) | Reg. count (Occupancy) | Run time (s) | BW (GB/s) | Reg. count (Occupancy) |
| nvcc - CUDA | 13.118 | 67 | 53 (56.25%) | 3.727 | 235 | 50 (56.25%) |
| clang - CUDA | 12.537 | 70 | 56 (56.25%) | 3.721 | 235 | 51 (56.25%) |
| PGI - F_CUDA | 16.880 | 72 | 69 (43.75%) | 4.425 | 198 | 78 (37.5%) |
| XL - F_CUDA | 16.235 | 54 | 72 (43.75%) | 3.968 | 221 | 70 (43.75%) |

**Table 5.** Measured run time, bandwidth, register count and occupancy values of **res_calc** in case of global coloring

| | K40 | | | P100 | | |
|---|---|---|---|---|---|---|
| | Run time (s) | BW (GB/s) | Reg. count (Occupancy) | Run time (s) | BW (GB/s) | Reg. count (Occupancy) |
| nvcc - CUDA | 21.133 | 41 | 46 (62.5%) | 6.706 | 131 | 40 (56.25%) |
| clang - CUDA | 21.083 | 42 | 46 (62.5%) | 6.676 | 131 | 40 (56.25%) |
| PGI - OpenACC | 21.472 | 41 | 72 (43.75%) | 6.617 | 132 | 88 (31.25%) |
| XL - OpenMP4 | 22.277 | 39 | 71 (43.75%) | 7.200 | 122 | 80 (37.5%) |
| clang - OpenMP4 | 22.245 | 39 | 96 (31.25%) | 6.676 | 131 | 96 (31.25%) |
| PGI - F_CUDA | 22.700 | 38 | 87 (31.25%) | 6.993 | 125 | 88 (31.25%) |
| PGI - F_OpenACC | 22.992 | 38 | 87 (31.25%) | 6.713 | 130 | 96 (31.25%) |
| XL - F_CUDA | 22.236 | 39 | 88 (31.25%) | 6.806 | 129 | 94 (31.25%) |
| XL - F_OpenMP4 | 23.755 | 37 | 110 (25%) | 7.229 | 121 | 104 (25%) |

**Table 6.** Average number of instructions and transactions performed in **res_calc** kernel with hierarchical coloring (absolute values for nvcc and for other versions relative to nvcc) on k40 GPU

| | nvcc | clang | Fortran PGI | Fortran XL |
|---|---|---|---|---|
| Integer instructions | 191743K | 0.86 | 0.82 | 0.90 |
| Floating point (64 bit) instructions | 88698K | 0.87 | 0.97 | 0.95 |
| Control instructions | 8955K | 0.90 | 0.64 | 0.26 |
| Texture read transactions | 761K | 1.00 | 16.91 | 8.88 |
| Global read transactions | 188K | 1.00 | 0.95 | 4.22 |

Fortran versions with hierarchical coloring have 23–30% worse performance than the same C/C++ versions, while with global coloring this difference is only 5% (15% on P100), but generally Fortran versions have high register counts and lower occupancy, as well as higher numbers of integer instructions and global load transactions. With Fortran the differences between the performance of CUDA and directive based approaches are about the same as described above.

**Table 7.** Average number of instructions and transactions performed in **res_calc** kernel with global coloring (absolute values for nvcc and for other versions relative to nvcc) on K40 GPU

| | fp (64 bit) | Integer | Control | Texture read transaction | Global read transaction |
|---|---|---|---|---|---|
| nvcc - CUDA | 93555K | 94994K | 1439K | 2175K | 334K |
| clang - CUDA | 0.98 | 0.94 | 1.00 | 1.01 | 1.04 |
| PGI - OpenACC | 1.03 | 1.38 | 1.00 | 0.98 | 1.00 |
| XL - OpenMP4 | 1.00 | 1.50 | 1.00 | 0.28 | 3.53 |
| clang - OpenMP4 | 0.97 | 1.26 | 1.00 | 0.00 | 3.42 |
| PGI - fortran CUDA | 1.03 | 1.80 | 2.00 | 1.05 | 13.47 |
| PGI - fortran OpenACC | 1.03 | 1.55 | 1.00 | 3.73 | 3.73 |
| XL - fortran CUDA | 1.00 | 2.20 | 2.00 | 3.74 | 3.73 |
| XL - fortran OpenMP4 | 1.00 | 1.82 | 1.00 | 3.77 | 3.73 |

**Table 8.** Measured run time, bandwidth, register count and occupancy values in case of **bres_calc** in case of hierarchical coloring

| | K40 | | | P100 | | |
|---|---|---|---|---|---|---|
| | Run Time (s) | BW (GB/s) | Reg. count (Occupancy) | Run Time (s) | BW (GB/s) | Reg. count (Occupancy) |
| nvcc - CUDA | 0.064 | 32 | 44 (62.5%) | 0.032 | 64 | 48 (62.5%) |
| clang - CUDA | 0.064 | 32 | 44 (62.5%) | 0.032 | 64 | 46 (62.5%) |
| PGI - F_CUDA | 0.082 | 25 | 53 (56.25%) | 0.029 | 71 | 72 (43.75%) |
| XL - F_CUDA | 0.061 | 33 | 48 (62.5%) | 0.035 | 59 | 64 (50%) |

**Table 9.** Measured run time, bandwidth, register count and occupancy values in case of **bres_calc** in case of global coloring

| | K40 | | | P100 | | |
|---|---|---|---|---|---|---|
| | Run time (s) | BW (GB/s) | Reg. count (Occupancy) | Run time (s) | BW (GB/s) | Reg. count (Occupancy) |
| nvcc - CUDA | 0.072 | 28 | 44 (62.5%) | 0.035 | 58 | 42 (62.5%) |
| clang - CUDA | 0.071 | 29 | 38 (75%) | 0.034 | 59 | 37 (75%) |
| PGI - OpenACC | 0.072 | 28 | 71 (43.75%) | 0.034 | 60 | 56 (56.25%) |
| XL - OpenMP4 | 0.084 | 24 | 72 (43.75%) | 0.037 | 55 | 80 (37.5%) |
| clang - OpenMP4 | 0.079 | 26 | 88 (31.25%) | 0.039 | 52 | 94 (31.25%) |
| PGI - F_CUDA | 0.096 | 21 | 56 (56.25%) | 0.038 | 54 | 72 (43.75%) |
| PGI - F_OpenACC | 0.073 | 28 | 102 (25%) | 0.036 | 57 | 88 (31.25%) |
| XL - F_CUDA | 0.078 | 26 | 70 (43.75%) | 0.037 | 55 | 80 (37.5%) |
| XL - F_OpenMP4 | 0.078 | 26 | 94 (31.25%) | 0.035 | 57 | 80 (37.5%) |

**bres_calc**: The **bres_calc** kernel also has indirect reads and writes, so we need coloring like with **res_calc**. In **bres_calc** the versions using hierarchical coloring performs equally good except the Fortran CUDA version compiled with the PGI compiler as shown in Table 8. The CUDA Fortran version with the PGI compiler has 30% lower performance compared to other versions with hierarchical coloring. On the K40 GPU in **res_calc** CUDA PGI has high number of load transactions but in this case the PGI version doesn't use the texture cache. However on the P100 GPU the version using the PGI compiler have same amount of memory transactions as nvcc, but executes less floating point operations. In case of global coloring on the C/C++ side OpenACC performs as good as CUDA versions despite the lower occupancy as shown in Table 9. However the OpenMP4 versions have the same issue as in case of **res_calc** and get high number of global read transactions while don't use the texture cache, which (with the lower occupancy due to high register counts) leads to the 20% lower performance.

In this case Fortran versions have only 10% lower performance than C/C++ versions (except for CUDA with the PGI compiler which has the same issue as with hierarchical coloring). The key reason for the difference is the lower occupancy of the Fortran versions and the higher instruction and memory transaction counts on both GPU. However in this case the directive based approaches performing equally to CUDA Fortran with the XL compiler. Surprisingly for **bres_calc** the Fortran OpenACC version has as low register count as the CUDA versions on the C/C++ side.

**Update:** The CUDA Fortran versions have lower occupancy because of the high register usage (the OpenACC version has a separate kernel for reduction thus have lower register count for the bulk of the kernel and the OpenMP4 version performs about the same as the C/C++ versions). All of the Fortran versions ended up with about 4 times more texture read (except OpenMP4 which doesn't use texture cache, but has 12 times more global loads), global load and store transactions than CUDA with nvcc. CUDA Fortran versions also have spilled registers (which introduce about 10 k–20 k local load and store transactions).

**Effect of tuning the number of registers per thread.** In case of the Airfoil application, the key performance limiter is the latency of accesses to global memory. To achieve high bandwidth, we need many loads in flight. This requires increasing the occupancy, which is limited by the number of registers used in these kernels. To get better occupancy we can limit the maximum number of registers per thread during the compilation. The register counts where the occupancy decreases if we use one more register per thread are the same for both K40 and P100 GPUs with 128 thread per block. For CUDA C/C++ versions we restricted the register counts to 56, 48 and 40 in order to increase occupancy, while for other versions we got higher register counts thus the restricted the register usage to 80, 72 and 64. With hierarchical coloring the shared memory required by the kernel could be the bottleneck for occupancy. In Figs. 3 and 4 the runtime of limited versions relative to the original version in percentage are

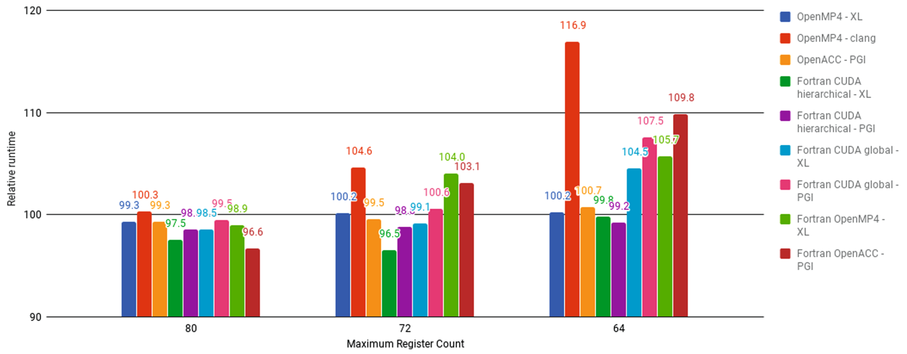



**Fig. 3.** Runtime of C OpenACC/OpenMP4 and Fortran versions with limited register per thread relative to original versions measured on K40. Lower is better.

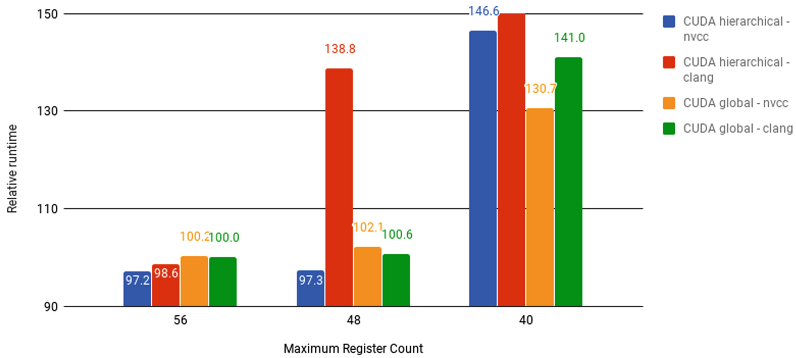



**Fig. 4.** Runtime of CUDA C versions with limited register per thread relative to original versions measured on K40. Lower is better.

shown. The shared memory requirement of **res_calc** and **bres_calc** is roughly 4 KB per block which limits the occupancy at 68.8% on the K40, meaning that we cannot reach better occupancy by further reducing the maximum register count (reducing the count to 48 would lead to 62.5% and to 40 would lead to 75% occupancy). On the P100 GPU shared memory requirement maximizes the occupancy at 94% thanks to more available shared memory. For most language-compiler combinations, limiting the register count only affects the **adt_calc**, **res_calc** and **bres_calc** kernels. In the OpenMP4 - clang, Fortran OpenMP4 - XL, and Fortran CUDA - PGI combinations, **update** is also affected by the limiting because of the high register count as shown in Table 10.

With the increased occupancy, we do get better run times in most cases (a limit of 56 in case of C/C++ and CUDA and 80 for other versions), except for the clang OpenMP4 and CUDA with nvcc. However further limitation of register counts leads to performance degradation, with the exception of CUDA Fortran code compiled with XL (which have the best performance with register count

**Table 10.** Measured run time, bandwidth, register count and occupancy values in case of **update** (for OpenACC versions the second register count belongs to the reduction kernel, the run times are the sum of the two kernels)

| | K40 | | | P100 | | |
|---|---|---|---|---|---|---|
| | Run time (s) | BW (GB/s) | Reg. count (Occupancy) | Run time (s) | BW (GB/s) | Reg. count (Occupancy) |
| nvcc - CUDA | 4.478 | 175 | 31 (100%) | 1.519 | 516 | 32 (100%) |
| clang - CUDA | 4.481 | 175 | 32 (100%) | 1.519 | 516 | 32 (100%) |
| PGI - OpenACC | 4.416 | 177 | 36 (75%)<br>18 (100%) | 1.588 | 493 | 38 (75%)<br>12 (100%) |
| XL - OpenMP4 | 4.497 | 174 | 32 (100%) | 1.660 | 472 | 32 (100%) |
| clang - OpenMP4 | 5.175 | 151 | 86 (31.25%) | 1.719 | 456 | 86 (31.25%) |
| PGI - F_CUDA | 4.598 | 170 | 79 (43.75%) | 1.654 | 474 | 48 (62.5%) |
| PGI - F_OpenACC | 4.350 | 180 | 37 (75%)<br>18 (100%) | 1.583 | 495 | 40 (75%)<br>16 (100%) |
| XL - F_CUDA | 4.598 | 169 | 80 (37.5%) | 1.566 | 500 | 80 (37.5%) |
| XL - F_OpenMP4 | 5.074 | 154 | 46 (62.5%) | 1.712 | 458 | 40 (75%) |

limited to 72). The reason for the loss of performance is the increasing number of spilled registers, and the latency introduced by the usage of these registers.

The main differences lie in the run times of **res_calc** and **adt_calc**. For **res_calc** on C/C++ side limiting the register count increases the performance by 2–5% in case of CUDA with hierarchical coloring, the OpenMP4 XL compiler and the Fortran versions also get better run times by 1–2% but the OpenACCversion performs the same, while the OpenMP4 clang versions get 2% higher run time thus get higher total run time despite of the 5% performance increase in **update** and **adt_calc**. For Fortran CUDA with XL and Fortran OpenACC with PGI compiler reach 15% better performance in **adt_calc** for the first level limitation. These results implies that for the most cases the increased occupancy gained with the restriction of the register usage could increase performance significantly (especially for kernels with low occupancy). In terms of instruction counts the limitation of register usage leads to slightly increased integer instruction counts in our cases.

### 5.2 Volna

For Volna the **SpaceDiscretization** kernel has a huge impact on runtime (half of the time is spent in this kernel when using global coloring), and so the hierarchical coloring leads to significant overall performance gain as shown on Fig. 5 (the measurements are in single precision because Volna requires only single precision to get correct results). However the presence of the local reads in **computeFluxes** in case of clang CUDA leads to 20% performance loss in this kernel. On other kernels we found the same tendencies as we observed on Airfoil, i.e. clang reaches lower floating point and integer instruction counts compared to

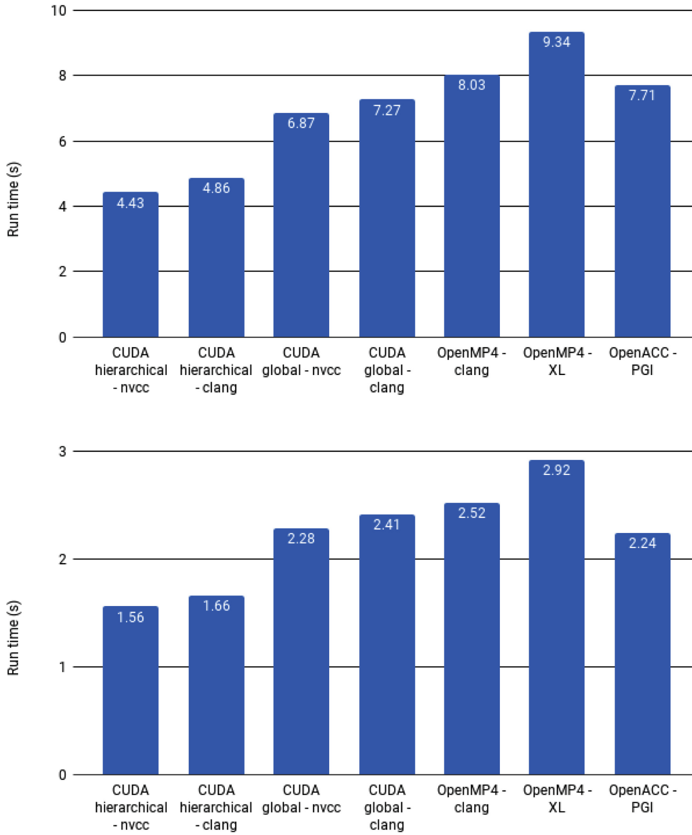



**Fig. 5.** Measured run times of Volna versions on the K40 and P100 GPU

nvcc. The directive based approaches have lower performance in the two most time consuming kernels. The OpenMP4 with XL has about 50% lower performance in **SpaceDiscretization** on the K40 GPU (the difference is 40% on the P100 machine), while for the other kernels these approaches performed within 10% of CUDA's performance and in some cases even better as shown in Table 11. In terms of occupancy the OpenMP4 with XL reach about the same occupancy in most cases as CUDA, while OpenMP4 clang and OpenACC have high register counts as shown in Table 12. In terms of instruction counts in case of Volna the directive based approaches performed the same as in Airfoil. The OpenMP4 versions don't use texture caches (in case of XL the texture cache usage is about 15% of nvcc's) and all directive based approach have higher global read transactions and about 30% higher integer instruction counts.

**Table 11.** Run times of the five most time consuming Volna kernels on the K40 GPU

| | nvcc CUDA | clang CUDA | nvcc CUDA global | clang CUDA global | clang OpenMP4 | XL OpenMP4 | PGI OpenACC |
|---|---|---|---|---|---|---|---|
| **compute Fluxes** | 1.336 | 1.693 | 1.323 | 1.734 | 2.186 | 1.613 | 1.623 |
| **Space Discretization** | 1.758 | 1.834 | 4.150 | 4.134 | 3.973 | 6.261 | 4.762 |
| **Numerical Fluxes** | 0.431 | 0.431 | 0.507 | 0.511 | 0.549 | 0.528 | 0.496 |
| **Evolve Values RK2_2** | 0.312 | 0.313 | 0.326 | 0.325 | 0.416 | 0.300 | 0.302 |
| **Evolve Values RK2_1** | 0.371 | 0.372 | 0.366 | 0.365 | 0.648 | 0.383 | 0.338 |

**Table 12.** Register counts and occupancy of the five most time consuming Volna kernels on the K40 GPU (for OpenACC the second register count belongs to the reduction kernel)

| | nvcc CUDA | clang CUDA | nvcc CUDA global | clang CUDA global | clang OpenMP4 | XL OpenMP4 | PGI OpenACC |
|---|---|---|---|---|---|---|---|
| **compute Fluxes** | 56 (56.25%) | 60 (56.25%) | 22 (100%) | 22 (100%) | 93 (31.25%) | 78 (37.5%) | 77 (37.5%) |
| **Space Discretization** | 32 (100%) | 36 (75%) | 28 (100%) | 25 (100%) | 64 (50%) | 30 (100%) | 30 (100%) |
| **Numerical Fluxes** | 28 (100%) | 16 (100%) | 45 (62.5%) | 46 (62.5%) | 40 (75%) | 30 (100%) | 33 (75%) 12 (100%) |
| **Evolve Values RK2_2** | 26 (100%) | 24 (100%) | 26 (100%) | 24 (100%) | 80 (37.5%) | 25 (100%) | 28 (100%) |
| **Evolve Values RK2_1** | 28 (100%) | 27 (100%) | 28 (100%) | 27 (100%) | 86 (31.25%) | 32 (100%) | 33 (75%) |

### 5.3 BookLeaf

Considering that in BookLeaf most of the time is spent in direct kernels or indirect read kernels, there is not as much difference between hierarchical and global coloring versions in total run time, as shown in Fig. 6. However in case of **getacc_scatter**, which is the only kernel with indirect increments among the top five most time consuming kernels, the runtime of the hierarchical coloring is at least 50% better than that of the global coloring versions. All versions are within 7% of the performance of the best version which is Fortran CUDA with hierarchical coloring compiled with the XL compiler on the K40, while on the P100 machine the PGI compiler performance is about 10% lower than the performance of the versions compiled with the XL compiler. As we saw in Airfoil, the CUDA versions have high register count in the most cases, but OpenACC and OpenMP4 reach better occupancy as shown in Table 14 which leads even better runtime than in case of CUDA versions as shown in Table 13.

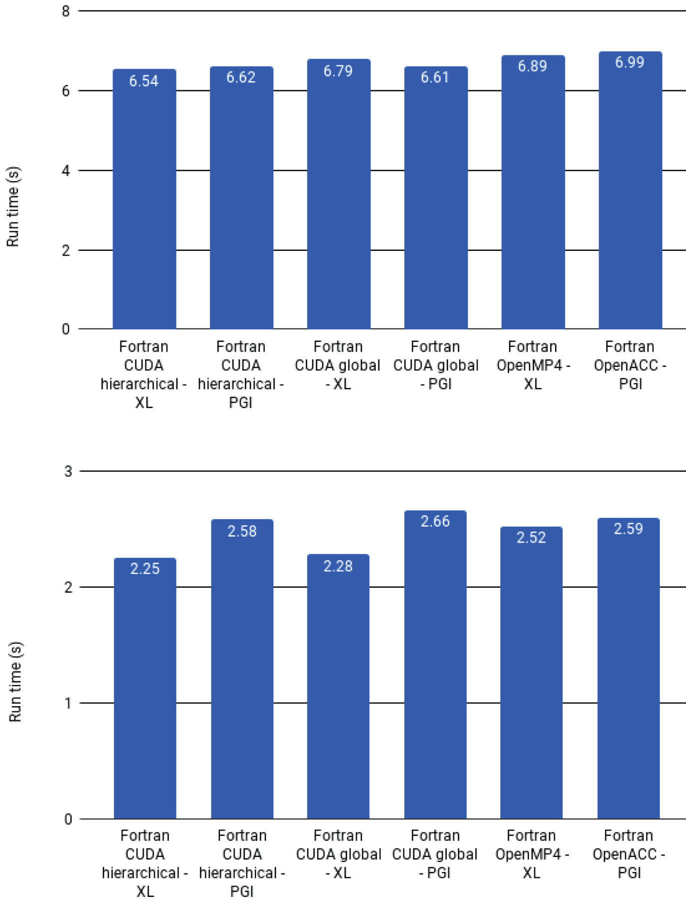



**Fig. 6.** Measured run times of BookLeaf versions on K40 and P100 GPU

**Table 13.** Run times of the five most time consuming BookLeaf kernel on K40 GPU

| | CUDA - PGI | CUDA global - PGI | OpenACC | CUDA - XL | CUDA global - XL | OpenMP4 |
|---|---|---|---|---|---|---|
| **getq_christiensen1** | 0.937 | 0.937 | 1.033 | 0.979 | 0.987 | 0.866 |
| **getq_christiensen_q** | 0.933 | 0.934 | 0.975 | 0.888 | 0.889 | 0.751 |
| **getacc_scatter** | 0.457 | 0.450 | 0.917 | 0.497 | 0.785 | 0.769 |
| **gather** | 0.526 | 0.526 | 0.525 | 0.523 | 0.523 | 0.542 |
| **getforce_visc** | 0.493 | 0.493 | 0.484 | 0.421 | 0.421 | 0.390 |

**Table 14.** Register counts of the five most time consuming BookLeaf kernel on K40 GPU

| | CUDA - PGI | CUDA global - PGI | OpenACC | CUDA - XL | CUDA global - XL | OpenMP4 |
|---|---|---|---|---|---|---|
| **getq_christiensen1** | 78 (37.5%) | 78 (37.5%) | 77 (37.5%) | 144 (18.75%) | 86 (31.25%) | 78 (37.5%) |
| **getq_christiensen_q** | 86 (31.25%) | 86 (31.25%) | 143 (18.75%) | 126 (25%) | 126 (25%) | 70 (43.75%) |
| **getacc_scatter** | 75 (37.5%) | 79 (37.5%) | 28 (100%) | 96 (31.25%) | 54 (56.25%) | 23 (100%) |
| **gather** | 30 (100%) | 30 (100%) | 23 (100%) | 32 (100%) | 32 (100%) | 23 (100%) |
| **getforce_visc** | 44 (62.5%) | 40 (75%) | 32 (100%) | 56 (56.25%) | 56 (56.25%) | 32 (100%) |

## 6 Conclusions

In this paper we have carried out a detailed study of some of the most popular parallelization approaches, programming languages, and compilers used to program GPUs, on a number of parallel loops coming from the domain of unstructured mesh computations. OpenMP4 and OpenACC are high level models using directives on loops in order to utilize GPUs, while CUDA use a lower level Single Instruction Multiple Threads model.

In this class of applications, a key common computational pattern is the indirect incrementing of data: to avoid race conditions we explored the use of coloring. The high level models must use global coloring of the iteration set to ensure that no two threads writes the same value when running simultaneously, whereas with lower-level models (CUDA) it is possible to apply a "two-level" coloring approach permitting better data reuse.

In case of Fortran, the CUDA versions with global coloring and OpenACC versions are within 10% of each other's performance. However the OpenMP4 versions use higher number of registers per thread in some cases, leading to low occupancy, as well as lower performance executing reductions. Directive based approaches also use higher numbers of integer and control instructions.

On the C/C++ side, CUDA code compiled with the clang compiler performs 2–5% better in terms of runtime and in most cases can outperform nvcc in the optimization of computations thus perform 20% fewer integer and floating point instructions compared to nvcc. The higher level approaches currently using more registers (even for simple kernels in case of OpenMP4 with the clang compiler) which leads to lower occupancy that lowers the performance. Also these versions now executing 30% more integer instructions than CUDA, but in some cases they performs within 5% of nvcc's performance. Since the support for OpenMP4 is relatively new there are still some issues that lowers performance, such as the more infrequent use of the texture cache and the lower performance when performing reductions. Also the OpenACC and OpenMP4 with the XL compiler currently have problems with computations with multiple increment of the same

data as in **adt_calc** where these versions write back all intermediate result to the global memory introducing the gap between the their performance and CUDA's.

We have also shown that using CUDA one can handle race conditions more efficiently thanks to block-level synchronization; this in turn enables an execution approach with much higher data reuse. Kernels with indirect increments using hierarchical coloring have significantly better performance than the versions using global coloring; in case of Airfoil hierarchical coloring leads to about 35% better overall performance, for Volna the difference is about 50% and with BookLeaf about 3%.

In summary, we have demonstrated that support for C is only slightly better than for Fortran, for all possible combinations, with a 3–10% performance gap. Our work is among the first ones comparatively evaluating the clang CUDA compiler and IBM's XL compilers; clang's CUDA support is showing great performance already, often outperforming nvcc. Even though the XL compilers are only about one year old, they are already showing competitive performance and good stability - on the OpenMP 4 side often outperforming clang's OpenMP 4 and PGI's OpenACC. Directive based approaches demonstrate good performance on simple computational loops, but struggle with more complex kernels due to increased register pressure and instruction counts - lagging behind CUDA on average by 5–15%, but in the worst cases by up to 50%. It still shows that OpenMP 4 GPU support isn't yet as mature as OpenACC, nevertheless, they are within 5–10%. Our results also demonstrate how CUDA allows for more flexibility in applying optimizations that are currently not possible with OpenACC or OpenMP 4.

**Acknowledgements.** The authors would like to thank the IBM Toronto compiler team, and Rafik Zurob in particular, for access to beta compilers and help with performance tuning, and Michal Iwanski and József Surányi at IBM for access to a Minsky system. Thanks to Carlo Bertolli at IBM TJ Watson for help with the clang OpenMP 4 compiler. This paper was supported by the János Bólyai Research Scholarship of the Hungarian Academy of Sciences. The authors would like to acknowledge the use of the University of Oxford Advanced Research Computing (ARC) facility in carrying out this work http://dx.doi.org/10.5281/zenodo.22558. The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013).

## References

1. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. Queue **6**(2), 40–53 (2008). https://doi.org/10.1145/1365490.1365500. ISSN: 1542-7730
2. Stone, J.E., Gohara, D., Shi, G.: OpenCL: a parallel programming standard for heterogeneous computing systems. IEEE Des. Test **12**(3), 66–73 (2010). https://doi.org/10.1109/MCSE.2010.69. ISSN: 0740-7475
3. Wienke, S., Springer, P., Terboven, C., an Mey, D.: OpenACC — first experiences with real-world applications. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 859–870. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32820-6_85

4. OpenMP 4.5 specification. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
5. Wu, J., Belevich, A., Bendersky, E., Heffernan, M., Leary, C., Pienaar, J., Roune, B., Springer, R., Weng, X., Hundt, R.: Gpucc: an open-source GPGPU Compiler. In: Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, pp. 105–116. ACM (2016). https://doi.org/10.1145/2854038.2854041. ISBN: 978-1-4503-3778-6
6. The Portland Group. http://www.pgroup.com
7. Ruetsch, G., Fatica, M.: CUDA Fortran for Scientists and Engineers: Best Practices for Efficient CUDA Fortran Programming. Elsevier, Amsterdam (2013)
8. Getting Started with CUDA Fortran programming using XL Fortran for Little Endian Distributions. http://www-01.ibm.com/support/docview.wss?uid=swg27047958&aid=11
9. Clang with OpenMP 4 support. https://github.com/clang-ykt
10. Ledur, C.L., Zeve, C.M., dos Anjos, J.C.: Comparative analysis of OpenACC, OpenMP and CUDA using sequential and parallel algorithms. In: 11th Workshop on Parallel and Distributed Processing (WSPPD) (2013)
11. Herdman, J., Gaudin, W., McIntosh-Smith, S., Boulton, M., Beckingsale, D.A., Mallinson, A., Jarvis, S.A.: Accelerating hydrocodes with OpenACC, OpenCL and CUDA. In: High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion, pp. 465–471. IEEE (2012)
12. Hoshino, T., Maruyama, N., Matsuoka, S., Takaki, R.: CUDA vs OpenACC: performance case studies with kernel benchmarks and a memory-bound CFD application. In: 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, pp. 136–143, May 2013. https://doi.org/10.1109/CCGrid.2013.12
13. Norman, M., Larkin, J., Vose, A., Evans, K.: A case study of CUDA FORTRAN and OpenACC for an atmospheric climate kernel. J. Comput. Sci. **9**, 1–6 (2015)
14. Kuan, L., Neves, J., Pratas, F., Tomás, P., Sousa, L.: Accelerating phylogenetic inference on GPUs: an OpenACC and CUDA comparison. In: IWBBIO 2014, pp. 589–600 (2014)
15. Gong, J., Markidis, S., Laure, E., Otten, M., Fischer, P., Min, M.: Nekbone performance on GPUs with OpenACC and CUDA fortran implementations. J. Supercomput. **72**(11), 4160–4180 (2016)
16. Antao, S.F., Bataev, A., Jacob, A.C., Bercea, G.-T., Eichenberger, A.E., Rokos, G., Martineau, M., Jin, T., Ozen, G., Sura, Z., et al.: Offloading support for OpenMP in clang and LLVM. In: Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC, pp. 1–11. IEEE Press (2016)
17. Martineau, M., Price, J., McIntosh-Smith, S., Gaudin, W.: Pragmatic performance portability with OpenMP 4.x. In: Maruyama, N., de Supinski, B.R., Wahib, M. (eds.) IWOMP 2016. LNCS, vol. 9903, pp. 253–267. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45550-1_18
18. Karlin, I., et al.: Early experiences porting three applications to OpenMP 4.5. In: Maruyama, N., de Supinski, B.R., Wahib, M. (eds.) IWOMP 2016. LNCS, vol. 9903, pp. 281–292. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45550-1_20
19. Hart, A.: First experiences porting a parallel application to a hybrid supercomputer with OpenMP4.0 device constructs. In: Terboven, C., de Supinski, B.R., Reble, P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2015. LNCS, vol. 9342, pp. 73–85. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24595-9_6

20. Reguly, I.Z., Keita, A.-K., Zurob, R., Giles, M.B.: High performance computing on the IBM power8 platform. In: Taufer, M., Mohr, B., Kunkel, J.M. (eds.) ISC High Performance 2016. LNCS, vol. 9945, pp. 235–254. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46079-6_17
21. OP2 github repository. https://github.com/OP2/OP2-Common
22. Mudalige, G., Giles, M., Reguly, I., Bertolli, C., Kelly, P.: OP2: an active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In: Innovative Parallel Computing (InPar), pp. 1–12. IEEE (2012)
23. Giles, M.B., Mudalige, G.R., Sharif, Z., Markall, G., Kelly, P.H.: Performance analysis and optimization of the OP2 framework on many-core architectures. Comput. J. **55**(2), 168–180 (2011)
24. Giles, M., Mudalige, G., Reguly, I.: OP2 airfoil example (2012)
25. Dutykh, D., Poncet, R., Dias, F.: The VOLNA code for the numerical modeling of tsunami waves: generation, propagation and inundation. Euro. J. Mech. B Fluids **30**(6), 598–615 (2011)
26. Uk mini-app consortium. https://uk-mac.github.io

# Periodic I/O Scheduling for Super-Computers

Guillaume Aupy[1], Ana Gainaru[2], and Valentin Le Fèvre[1,3(✉)]

[1] Inria and University of Bordeaux, Talence, France
[2] Vanderbilt University, Nashville, TN, USA
[3] École Normale Supérieure de Lyon, Lyon, France
valentin.le-fevre@ens-lyon.fr

**Abstract.** With the ever-growing need of data in HPC applications, the congestion at the I/O level becomes critical in super-computers. Architectural enhancement such as burst-buffers and pre-fetching are added to machines, but are not sufficient to prevent congestion. Recent online I/O scheduling strategies have been put in place, but they add an additional congestion point and overheads in the computation of applications.

In this work, we show how to take advantage of the periodic nature of HPC applications in order to develop efficient periodic scheduling strategies for their I/O transfers. Our strategy computes once during the job scheduling phase a pattern where it defines the I/O behavior for each application, after which the applications run independently, transferring their I/O at the specified times. Our strategy limits the amount of I/O congestion at the I/O node level and can be easily integrated into current job schedulers. We validate this model through extensive simulations and experiments by comparing it to state-of-the-art online solutions.

Specifically, we show that not only our scheduler has the advantage of being de-centralized, thus overcoming the overhead of online schedulers, but we also show that on Mira one can expect an average dilation improvement of 22% with an average throughput improvement of 32%! Finally, we show that one can expect those improvements to get better in the next generation of platforms where the compute - I/O bandwidth imbalance increases.

## 1 Introduction

Nowadays, supercomputing applications create or have to deal with TeraBytes of data. This is true in all fields: as example LIGO (gravitational wave detection) generates 1500 TB/year [22], the Large Hadron Collider generates 15 PB/year, light source projects deal with 300 TB of data per day and climate modeling applications are expected to have to deal with 100EB of data [17]. According to experts "Very few large scale applications of practical importance are not data intensive" (Alok Choudhary, Apr 2012).

Management of I/O operations is critical at scale. However, observations on the Intrepid machine at Argonne National Lab show that I/O transfer can be slowed down up to 70% due to congestion [14]. In 2013, Argonne upgraded its

© Springer International Publishing AG 2018
S. Jarvis et al. (Eds.): PMBS 2017, LNCS 10724, pp. 44–66, 2018.
https://doi.org/10.1007/978-3-319-72971-8_3

house supercomputer: moving from Intrepid (Peak performance: 0.56 PFlop/s; peak I/O throughput: 88 GB/s) to Mira (Peak performance: 10 PFlop/s; peak I/O throughput: 240 GB/s). In 2018, the new machine at Argonne, Aurora, is expected to have a Peak performance of 450 PFlops/s and a peak I/O throughput of 1 TB/s. While both criteria seem to continuously improve considerably, the reality behind is that for a given application, its I/O throughput scales linearly (or worse) with its performance, and hence, what should be noticed is a downgrade from 160 GB/PFlop (Intrepid) to 24 GB/PFlop (Mira) and finally 2.2 GB/PFlop (Aurora)!

With this in mind, to be able to scale, conception of new algorithms has to change paradigm: going from a compute-centric model to a data-centric model.

To help with the ever growing amount of data created, architectural improvement such as burst buffers [23] have been added to the system. Work is being done to transform the data before sending it to the disks in the hope of reducing the I/O sent [11]. However, even with the current I/O footprint burst buffers are not able to completely hide congestion. Moreover, the data used is always expected to grow. Recent works [14] have started working on novel online, centralized I/O scheduling strategies at the I/O node level. However one of the risk noted on these strategies is the scalability issue caused by potentially high overheads (between 1 and 5% depending on the number of nodes used in the experiments) [14]. Moreover, it is expected this overhead to increase at larger scale since it need centralized information about all applications running in the system.

*In this paper, we present a decentralized I/O scheduling strategy for supercomputers. We show how to take known HPC application behaviors (namely their periodicity) into account to derive novel static algorithms.*

Many recent HPC studies have observed independent patterns in the I/O behavior of HPC applications. The periodicity of HPC applications has been well observed and documented [7,12,14]: HPC applications alternate between computation and I/O transfer, this pattern being repeated over-time. Furthermore, fault-tolerance techniques (such as periodic checkpointing [10]) also add to this periodic behavior. Carns et al. [7] observed with Darshan the periodicity of four different applications (MADBench2 [8], Chombo I/O benchmark [9], S3D IO [27] and HOMME [26]). Furthermore, in our previous work [14] we were able to verify the periodicity of gyrokinetic toroidal code (GTC) [13], Enzo [6], HACC application [15] and CM1 [5].

Recently, Hu et al. [18] summed up the four key characteristics of HPC applications observed in the literature:

1. *Periodicity*: Applications alternate between compute phases and I/O phases. Furthermore they do so in a periodic fashion: a regular pattern of computation - I/O is repeated over time.
2. *Burstiness*: In addition to the periodicity observed, sometimes, short I/O bursts occur.
3. *Synchronization*: I/O accesses of an application are performed in a synchronized way between the different parallel processes.

4. *Repeatability*: The same jobs are often run many times with only different input, hence the compute-I/O pattern of an application can be predicted before it is executed.

The key idea in this project is to take into account those known structural behaviors of HPC applications and to include them in scheduling strategies.

Using this periodicity property, we compute a static periodic scheduling strategy, which provides a way for each application to know when they should start transferring their I/O (i) hence reducing potential bottlenecks either due to I/O congestion, and (ii) without having to consult with I/O nodes every time I/O should be done and hence adding an extra overhead. The main contributions of this paper are:

– A novel light-weight I/O algorithm that looks at optimizing both application-oriented (dilation or fairness) and platform-oriented (maximum system efficiency) objectives;
– A set of extensive simulations and experiments that show that this algorithm performs as well or better than current state of the art heavy-weight online algorithms.

Note that the algorithm presented here is done as a proof of concept to show the efficiency of these kind of light-weight techniques. We believe our scheduler can be implemented naturally into a job scheduler and we provide experimental results backing this claim. However, this integration is beyond the scope of this paper. For the purpose of this paper the applications are already scheduled on the system and are able to receive information about their I/O scheduling. The goal of our I/O scheduler is to eliminate congestion points caused by application interference while keeping the overhead seen by all applications to the minimum. Computing a full I/O schedule over all iterations of all applications is not realistic at today's scale. The process would be too expensive both in time and space. Our scheduler overcomes this by computing a period of I/O scheduling that includes different number of iterations for each application.

The rest of the paper is organized as follows: in Sect. 2 we present the application model and optimization problem. In Sect. 3 we present our novel algorithm technique as well as a brief proof of concept for a future implementation. In Sect. 4 we present extensive simulations based on the model to show the performance of our algorithm compared to state of the art. We then confirm the performance on a super-computer to validate the model. We give some background and related work in Sect. 5. We provide concluding remarks and ideas for future research directions in Sect. 6.

## 2 Model

In this section we use the model introduced in our previous work [14] that has been verified experimentally to be consistent with the behavior of Intrepid and Mira, super-computers at Argonne.

We consider scientific applications running at the same time on a parallel platform. The applications consist of series of computations followed by I/O operations. On a super-computer, the computations are done independently because each application uses its own nodes. However, the applications are concurrently sending and receiving data during their I/O phase on a dedicated I/O network. The consequence of this I/O concurrency is congestion between an I/O node of the platform and the file storage.

## 2.1 Parameters

We assume that we have a parallel platform made up of $N$ identical unit-speed nodes, each equipped with an I/O card of bandwidth $b$ (expressed in bytes per second). We further assume having "a centralized I/O system with a total bandwidth $B$ (also expressed in bytes per second). This means that the total bandwidth between the computation nodes and an I/O node is $N \cdot b$ while the bandwidth between an I/O node and the file storage is $B$, with usually $N \cdot b \gg B$. We have instantiated this model for the Intrepid platform on Fig. 1.

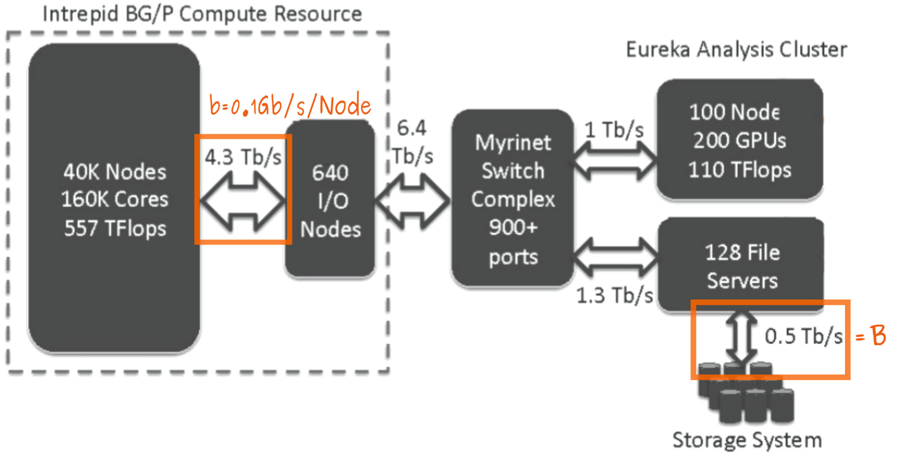



**Fig. 1.** Model instantiation for the Intrepid platform [14].

We have $K$ applications, all assigned to independent and dedicated computational resources, but competing for I/O. For each application $\text{App}^{(k)}$ we define:

- Its size: $\text{App}^{(k)}$ executes with $\beta^{(k)}$ dedicated nodes;
- Its pattern: $\text{App}^{(k)}$ obeys a pattern that repeats over time. There are $n_{\text{tot}}^{(k)}$ *instances* of $\text{App}^{(k)}$ that are executed one after the other. Each instance consists of two disjoint phases: computations that take a time $w^{(k)}$, followed by I/O transfers for a total volume $\text{vol}_{\text{io}}^{(k)}$. The next instance cannot start before I/O operations for the current instance is terminated.

We further denote by $r_k$ the time when $\text{App}^{(k)}$ is released on the platform and $d_k$ the time when the last instance is completed. Finally, we denote by $\gamma^{(k)}(t)$, the bandwidth used by a node on which application $\text{App}^{(k)}$ is running, at instant $t$. For simplicity we assume just one I/O transfer in each loop. However, our model can be extended to work with multiple I/O patterns as long as these are periodic in nature or as long as they are known in advance.

### 2.2 Execution Model

As the computation resources are dedicated, we can always assume w.l.o.g that the next computation chunk starts right away after completion of the previous I/O transfers, and is executed at full (unit) speed. On the contrary, all applications compete for I/O, and congestion will likely occur. The simplest case is that of a single periodic application $\text{App}^{(k)}$ using the I/O system in dedicated mode during a time-interval of duration $D$. In that case, let $\gamma$ be the I/O bandwidth used by each processor of $\text{App}^{(k)}$ during that time-interval. We derive the condition $\beta^{(k)}\gamma D = \text{vol}_{\text{io}}^{(k)}$ to express that the entire I/O data volume is transferred. We must also enforce the constraints that (i) $\gamma \leq b$ (output capacity of each processor); and (ii) $\beta^{(k)}\gamma \leq B$ (total capacity of I/O system). Therefore, the minimum time to perform the I/O transfers for an instance of $\text{App}^{(k)}$ is $\text{time}_{\text{io}}^{(k)} = \frac{\text{vol}_{\text{io}}^{(k)}}{\min(\beta^{(k)}b, B)}$. However, in general many applications will use the I/O system simultaneously, whose bandwidth capacity $B$ will be shared among all these applications (see Fig. 2). Scheduling application I/O will guarantee that the I/O network will not be loaded with more than its designed capacity. Figure 2 presents the view of the machine when 3 applications are sharing the I/O system. This translates at the application level to delays inserted before I/O bursts (see Fig. 3 for application 2's point of view).

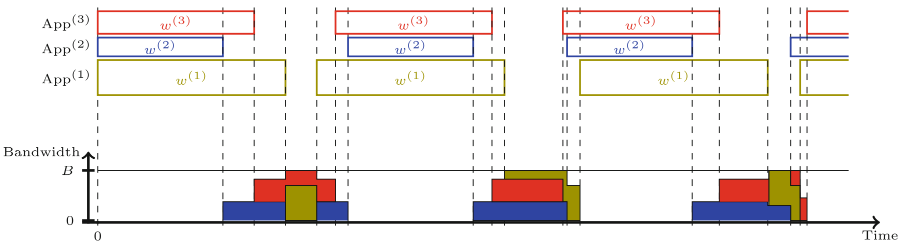



**Fig. 2.** Scheduling the I/O of three periodic applications (top: computation, bottom: I/O).

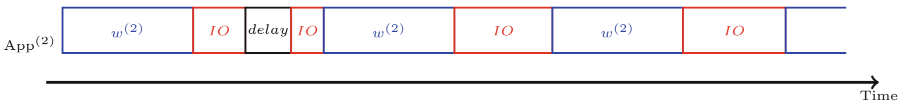



**Fig. 3.** Application 2 execution view

This model is very flexible, and the only assumption is that at any instant, all nodes assigned to a given application are assigned the same bandwidth. This assumption is transparent for the I/O system and simplifies the problem statement without being restrictive. Again, in the end, the total volume of I/O transfers for an instance of $\text{App}^{(k)}$ must be $\text{vol}_{\text{io}}^{(k)}$, and at any instant, the rules of the game are simple: never exceed the individual bandwidth $b$ of each processor ($\gamma^{(k)}(t) \leq b$ for any $k$ and $t$), and never exceed the total bandwidth $B$ of the I/O system ($\sum_{k=1}^{K} \beta^{(k)}\gamma^{(k)}(t) \leq B$ for any $t$).

## 2.3 Objectives

We now focus on the optimization objectives at hand here. We use the objectives introduced in [14].

First, the *application efficiency* achieved for each application $\text{App}^{(k)}$ at time $t$ is defined as

$$\tilde{\rho}^{(k)}(t) = \frac{\sum_{i \leq n^{(k)}(t)} w^{(k,i)}}{t - r_k},$$

where $n^{(k)}(t) \leq n_{\text{tot}}^{(k)}$ is the number of instances of application $\text{App}^{(k)}$ that have been executed at time $t$, since the release of $\text{App}^{(k)}$ at time $r_k$. Because we execute $w^{(k,i)}$ units of computation followed by $\text{vol}_{\text{io}}^{(k,i)}$ units of I/O operations on instance $\mathcal{I}_i^{(k)}$ of $\text{App}^{(k)}$, we have $t - r_k \geq \sum_{i \leq n^{(k)}(t)} \left( w^{(k,i)} + \text{time}_{\text{io}}^{(k,i)} \right)$. Due to I/O congestion, $\tilde{\rho}^{(k)}$ never exceeds the optimal efficiency that can be achieved for $\text{App}^{(k)}$, namely

$$\rho^{(k)} = \frac{w^{(k)}}{w^{(k)} + \text{time}_{\text{io}}^{(k)}}$$

The two key optimization objectives, together with a rationale for each of them, are:

- SysEfficiency: where we maximize the peak performance of the platform, namely maximizing the amount of operations per time unit:

$$\text{maximize } \frac{1}{N} \sum_{k=1}^{K} \beta^{(k)} \tilde{\rho}^{(k)}(d_k). \tag{1}$$

- Dilation: where we minimize the largest slowdown imposed to each application (hence optimizing fairness across applications):

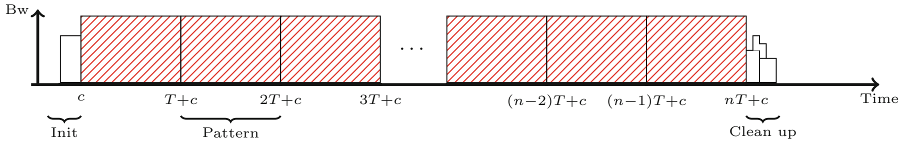
$$\text{minimize } \max_{k=1..K} \frac{\rho^{(k)}}{\tilde{\rho}^{(k)}(d_k)}. \tag{2}$$

Note that it is known that both problems are NP-complete, even in an (easier) offline setting [14].

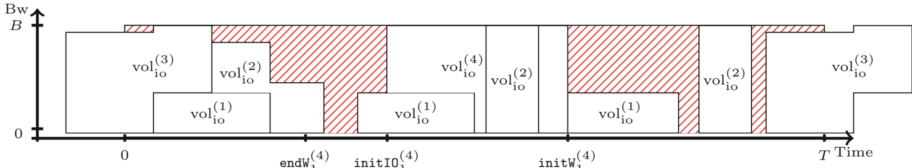
## 3 Periodic Scheduling Strategy

In general, for an application $\mathrm{App}^{(k)}$, $n_{\mathrm{tot}}^{(k)}$ the number of instances of $\mathrm{App}^{(k)}$ is very large and not polynomial in the size of the problem. For this reason, online schedule have been preferred until now. The key novelty of this paper is to introduce *periodic schedules* for the $K$ applications. Intuitively, we are looking for a computation and I/O *pattern* of duration $T$ that will be repeated over time (except for *initialization* and *clean up* phases), as shown on Fig. 4a. In this section, we start by introducing the notion of periodic schedule and a way to compute the application efficiency differently. We then provide the algorithms that are at the core of this work.

Because there is no competition on computation (no shared resources), we can consider that a chunk of computation directly follows the end of the I/O transfer, hence we need only to represent I/O transfers in this pattern. The bandwidth used by each application during the I/O operations is represented over time, as shown in Fig. 4b. We can see that an operation can overlap with the one of the previous pattern or the next pattern, but overall, the pattern will just repeat.




(a) Periodic schedule (phases)




(b) Detail of I/O in a period/pattern

**Fig. 4.** A schedule (above), and the detail of one of its regular pattern (below), where $(w^{(1)} = 3.5; \mathrm{vol}_{\mathrm{io}}^{(1)} = 240; n_{\mathrm{per}}^{(1)} = 3)$, $(w^{(2)} = 27.5; \mathrm{vol}_{\mathrm{io}}^{(2)} = 288; n_{\mathrm{per}}^{(2)} = 3)$, $(w^{(3)} = 90; \mathrm{vol}_{\mathrm{io}}^{(3)} = 350; n_{\mathrm{per}}^{(3)} = 1)$, $(w^{(4)} = 75; \mathrm{vol}_{\mathrm{io}}^{(4)} = 524; n_{\mathrm{per}}^{(4)} = 1)$.

To describe a pattern, we use the following notations:

- $n_{\mathrm{per}}^{(k)}$: the number of instances of $\mathrm{App}^{(k)}$ during a pattern.
- $\mathcal{I}_i^{(k)}$: the $i$-th instance of $\mathrm{App}^{(k)}$ during a pattern.
- $\mathtt{initW}_i^{(k)}$: the time of the beginning of $\mathcal{I}_i^{(k)}$. So, $\mathcal{I}_i^{(k)}$ has a computation interval going from $\mathtt{initW}_i^{(k)}$ to $\mathtt{endW}_i^{(k)} = \mathtt{initW}_i^{(k)} + w^{(k)} \bmod T$.

- $\texttt{initIO}_i^{(k)}$: the time when the I/O transfer from the $i$-th instance of $\text{App}^{(k)}$ starts (between $\texttt{endW}_i^{(k)}$ and $\texttt{initIO}_i^{(k)}$, $\text{App}^{(k)}$ is idle). Therefore, we have

$$\int_{\texttt{initIO}_i^{(k)}}^{\texttt{initW}^{(k)}_{(i+1)\%n_{\text{per}}^{(k)}}} \beta^{(k)}\gamma^{(k)}(t)dt = \text{vol}_{\text{io}}^{(k)}.$$

Globally, if we consider the two dates per instance $\texttt{initW}_i^{(k)}$ and $\texttt{initIO}_i^{(k)}$, that define the change between computation and I/O phases, we have a total of $S \leq \sum_{k=1}^{K} 2n_{\text{per}}^{(k)}$ distinct dates, that are called the *events* of the pattern.

We define the periodic efficiency of a pattern of size $T$:

$$\tilde{\rho}_{\text{per}}^{(k)} = \frac{n_{\text{per}}^{(k)} w^{(k)}}{T}. \tag{3}$$

For periodic schedules, we use it to approximate the actual efficiency achieved for each application. The rationale behind this can be seen on Fig. 4. If $\text{App}^{(k)}$ is released at time $r_k$, and the first pattern starts at time $r_k + c$, that is after an initialization phase, then the main pattern is repeated $n$ times (until time $n \cdot T + r_k + c$), and finally $\text{App}^{(k)}$ ends its execution after a clean-up phase at time $d_k = r_k + c + n \cdot T + c'$. If we assume that $n \cdot T \gg c + c'$, then $d_k - r_k \approx n \cdot T$. Then the value of the $\tilde{\rho}^{(k)}(d_k)$ for $\text{App}^{(k)}$ is:

$$\begin{aligned} \tilde{\rho}^{(k)}(d_k) &= \frac{\left(n \cdot n_{\text{per}}^{(k)} + \delta\right) w^{(k)}}{d_k - r_k} = \frac{\left(n \cdot n_{\text{per}}^{(k)} + \delta\right) w^{(k)}}{c + n \cdot T + c'} \\ &\approx \frac{n_{\text{per}}^{(k)} w^{(k)}}{T} = \tilde{\rho}_{\text{per}}^{(k)} \end{aligned}$$

where $\delta$ can be 1 or 0 depending whether $\text{App}^{(k)}$ was executed or not during the clean-up or init phase.

### 3.1 PerSched: A Periodic Scheduling Algorithm

*For details in the implementation, we refer the interested reader to the source code available at* https://github.com/vlefevre/IO-scheduling-simu.

The difficulties of finding an efficient periodic schedule are three-fold:

- The first one is that the right pattern size has to be determined;
- The second one is that for a given pattern size, the number of instances of each application that should be included in this pattern need to be determined;
- Finally, the time constraint between two consecutive I/O transfers of a given application, due to the computation in-between makes naive scheduling strategies harder to implement.

*Finding the right pattern size.* A solution is to find schedules with different pattern sizes between a minimum pattern size $T_{\min}$ and a maximum pattern size $T_{\max}$.

Because we want a pattern to have at least one instance of each application, we can trivially set up $T_{\min} = \max_k(w^{(k)} + \text{time}_{\text{io}}^{(k)})$. Intuitively, the larger $T_{\max}$ is, the more possibilities we can have to find a good solution. However this also increases the complexity of the algorithm. We want to limit the number of instances of all applications in a schedule. For this reason we chose to have $T_{\max} = O(\max_k(w^{(k)} + \text{time}_{\text{io}}^{(k)}))$. We discuss this hypothesis in Sect. 4, where we give better experimental intuition on finding the right value for $T_{\max}$. Experimentally we observe (see the companion report [1]) that $T_{\max} = 10T_{\min}$ seems to be sufficient.

We then decided on an iterative search where the pattern size increases exponentially at each iteration from $T_{\min}$ to $T_{\max}$. In particular, we use a precision $\varepsilon$ as input and we iteratively increase the pattern size from $T_{\min}$ to $T_{\max}$ by a factor $(1+\varepsilon)$. This allows us to have a polynomial number of iterations. The rationale behind the exponential increase is that when the pattern size gets large, we expect performance to converge to an optimal value, hence needing less the precision of a precise pattern size. Furthermore while we could try only large pattern sizes, it seems important to find a good small pattern size as it would simplify the scheduling step. Hence a more precise search for smaller pattern sizes. Finally, we expect the best performance to cycle with the pattern size. We verify these statements experimentally in the companion report [1].

*Determining the number of instances of each application.* By choosing $T_{\max} = O(\max_k(w^{(k)} + \text{time}_{\text{io}}^{(k)}))$, we guarantee the maximum number of instances of each application that fit into a pattern is $O\left(\frac{\max_k(w^{(k)}+\text{time}_{\text{io}}^{(k)})}{\min_k(w^{(k)}+\text{time}_{\text{io}}^{(k)})}\right)$.

*Instance scheduling.* Finally, our last item is, given a pattern of size $T$, how to schedule instances of applications into a periodic schedule.

To do this, we decided on a strategy where we insert instances of applications in a pattern, without modifying dates and bandwidth of already scheduled instances. Formally, we call an application schedulable:

**Definition 1 (Schedulable).** *Given an existing pattern $\mathcal{P} = \cup_{k=1}^{K}\left(n_{per}^{(k)}, \cup_{i=1}^{n_{per}^{(k)}}\{\mathtt{initW}_i^{(k)}, \mathtt{initIO}_i^{(k)}, \gamma^{(k)}()\}\right)$, we say that an application $App^{(k)}$ is schedulable if there exists $1 \le i \le n_{per}^{(k)}$, such that:*

$$\int_{\mathtt{initW}_i^{(k)}+w^{(k)}}^{\mathtt{initIO}_i^{(k)}-w^{(k)}} \min\left(\beta^{(k)}b, B - \sum_l \beta^{(l)}\gamma^{(l)}(t)\right) dt \ge vol_{io}^{(k)} \tag{4}$$

To understand Eq. (4): we are checking that during the end of the computation of the $i^{\text{th}}$ instance ($\mathtt{initW}_i^{(k)} + w^{(k)}$), and the beginning of the computation

of the $i+1^{\text{th}}$ instance ($\texttt{initIO}_i^{(k)} - w^{(k)}$: this will represent the beginning of computation of the $i+1^{\text{th}}$ instance after the insertion of the new one, but currently it is just some time before the I/O transfer of the $i^{\text{th}}$ instance), there is enough bandwidth to perform at least a volume of I/O of $\text{vol}_{\text{io}}^{(k)}$. We represent it graphically on Fig. 5.

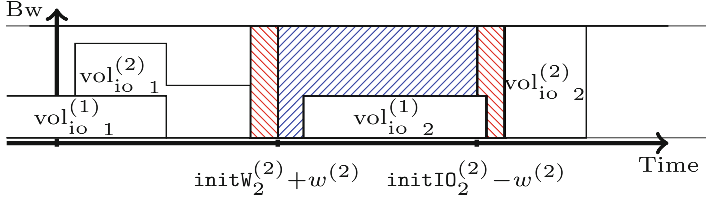



**Fig. 5.** Graphical description of Definition 1: to insert an instance of $\text{App}^{(2)}$, we need to check that the blue area is greater than $\text{vol}_{\text{io}}^{(2)}$ with the bandwidth constraint. The red area is off limit for I/O as it would be used for computations. (Color figure online)

With Definition 1, we can now explain the core idea of the instance scheduling part of our algorithm. Starting from an existing pattern, while there exist applications that are schedulable:

– Amongst the applications that are schedulable, we choose the application that has the worse Dilation. The rationale is that even though we want to increase SysEfficiency, we do it in a way that ensures that all applications are treated fairly;
– We insert the instance into an existing scheduling using a procedure Insert-In-Pattern such that (i) the first instance of each application is inserted so that it minimizes its I/O transfer time, (ii) the other instances are inserted just after the last inserted one.

With all of this in mind, we can now write PerSched (Algorithm 1), our algorithm to construct a periodic pattern. For all pattern sizes tried between $T_{\min}$ and $T_{\max}$, we return the pattern with maximal SysEfficiency. For space concerns, we present here a simplified version of the real PerSched algorithm used in the simulations. You can find the minor improvement in the companion report [1].

### 3.2 Complexity Analysis

Due to lack of space, we only give the complexity of our algorithm, the proof is in the companion report [1].

**Theorem 1.** *Let* $n_{\max} = \left(\frac{\max_k(w^{(k)}+time_{io}^{(k)})}{\min_k(w^{(k)}+time_{io}^{(k)})}\right)$,
$\textsc{PerSched}(K', \varepsilon, \{App^{(k)}\}_{1 \le k \le K})$ *runs in*

$$O\left(\left(\left\lceil\frac{1}{\varepsilon}\right\rceil + \left\lceil\frac{\log K'}{\log(1+\varepsilon)}\right\rceil\right) \cdot K^2 \left(n_{\max} + \log K'\right)\right).$$

Note that in practice, both $K'$ and $K$ are small ($\approx 10$), and $\varepsilon$ is close to 0, hence making the complexity $O\left(\frac{n_{\max}}{\varepsilon}\right)$.

**Algorithm 1.** Periodic Scheduling heuristic: PerSched

```
1  procedure PerSched(K', ε, {App^(k)}_{1≤k≤K})
2  begin
3      T_min ← max_k(w^(k) + time_io^(k));
4      T_max ← K' · T_min;
5      T = T_min;
6      SE ← 0;
7      T_opt ← 0;
8      P_opt ← {};
9      while T ≤ T_max do
10         P = {};
11         while exists a schedulable application do
12             A = {App^(k) | App^(k) is schedulable};
13             Let App^(k) be the element of A minimal with respect to the
               lexicographic order (ρ^(k)/ρ̃_per^(k), w^(k)/time_io^(k));
14             P ← Insert-In-Pattern(P, App^(k));
15         if SE < SysEfficiency(P) then
16             SE ← SysEfficiency(P);
17             T_opt ← T;
18             P_opt ← P
19         T ← T · (1 + ε);
20     return P_opt
```

We estimate SysEfficiency of a periodic pattern, by replacing $\tilde{\rho}^{(k)}(d_k)$ by $\tilde{\rho}_{\text{per}}^{(k)}$ in Eq. (1)

### 3.3 High-Level Implementation, Proof of Concept

We envision the implementation of this periodic scheduler to take place at two levels:

(1) The job scheduler would know the application profiles (using solutions such as Omnisc'IO [12]). Using the profiles, it would be in charge of computing a periodic pattern every time an application enters or leaves the system.

(2) Application-side I/O management strategies (such as [24,32,33]) then would be responsible to ensure the correct I/O transfer at the right time by limiting the bandwidth used by nodes that transfer I/O. The start and end time for each I/O as well as the used bandwidth are described in input files.

# 4 Evaluation and Model Validation

*Note that the data used for this section and the scripts to generate the figures are available at* https://github.com/vlefevre/IO-scheduling-simu.

In this section, (i) we assess the efficiency of our algorithm by comparing it to a recent dynamic framework [14], and (ii) we validate our model by comparing theoretical performance (as obtained by the simulations) to actual performance on a real system.

We perform the evaluation in three steps: first we simulate behavior of applications and input them into our model to estimate both Dilation and SysEfficiency of our algorithm (Sect. 4.4) and evaluate these cases on an actual machine to confirm the validity of our model. Once the model is validated, we perform extensive simulations.

## 4.1 Experimental Setup

The platform available for experimentation is Jupiter at Mellanox, Inc. To be able to verify our model, we use it to instantiate our platform model. Jupiter is a Dell PowerEdge R720xd/R720 32-node cluster using Intel Sandy Bridge CPUs. Each node has dual Intel Xeon 10-core CPUs running at 2.80 GHz, 25 MB of L3, 256 KB unified L2 and a separate L1 cache for data and instructions, each 32 KB in size. The system has a total of 64 GB DDR3 RDIMMs running at 1.6 GHz per node. Jupiter uses Mellanox ConnectX-3 FDR 56 Gb/s InfiniBand and Ethernet VPI adapters and Mellanox SwitchX SX6036 36-Port 56 Gb/s FDR VPI InfiniBand switches.

We measured the different bandwidths of the machine and obtained $b = 0.01$ GB/s and $B = 3$ GB/s. Therefore, when 300 cores transfer at full speed (less than half of the 640 available cores), congestion occurs.

*Implementation of scheduler on Jupiter.* We simulate the existence of such a scheduler by computing beforehand the I/O pattern for each application and feeding it as input files. The experiments require a way to control for how long they use the CPU or stay idle waiting to start their I/O in addition to the amount of I/O they are writing to the disk. For this purpose, we modified the IOR benchmark [30] to read the input files that provide the start and end time for each I/O transfer as well as the bandwidth used. Our scheduler generates one such file for each application. The IOR benchmark is split in different sets of processes running independently on different nodes, where each set represents a different application. One separate process acts as the scheduler and receives I/O requests for all groups in IOR. Since we are interested in modeling the

**Table 1.** Details of each application.

| App$^{(k)}$ | $w^{(k)}$ (s) | $\text{vol}_{\text{io}}^{(k)}$ (GB) | $\beta^{(k)}$ |
|---|---|---|---|
| Turbulence1 (T1) | 70 | 128.2 | 32,768 |
| Turbulence2 (T2) | 1.2 | 235.8 | 4,096 |
| AstroPhysics (AP) | 240 | 423.4 | 8,192 |
| PlasmaPhysics (PP) | 7554 | 34304 | 32,768 |

**Table 2.** Number of applications of each type launched at the same time for each experiment scenario.

| Set # | T1 | T2 | AP | PP |
|---|---|---|---|---|
| 1 | 0 | **10** | 0 | 0 |
| 2 | 0 | **8** | **1** | 0 |
| 3 | 0 | **6** | **2** | 0 |
| 4 | 0 | **4** | **3** | 0 |
| 5 | 0 | **2** | 0 | **1** |
| 6 | 0 | **2** | **4** | 0 |
| 7 | **1** | **2** | 0 | 0 |
| 8 | 0 | 0 | **1** | **1** |
| 9 | 0 | 0 | **5** | 0 |
| 10 | **1** | 0 | **1** | 0 |

I/O delays due to congestion or scheduler imposed delays, the modified IOR benchmarks do not use inter-processor communications. Our modified version of the benchmark reads the I/O scheduling file and adapts the bandwidth used for I/O transfers for each application as well as delaying the beginning of I/O transfers accordingly.

We made experiments on our IOR benchmark and compared the results between periodic and online schedulers as well as with the performance of the original IOR benchmark without any extra scheduler.

### 4.2 Applications and Scenarios

In the literature, there are many examples of periodic applications. Carns et al. [7] observed with Darshan the periodicity of four different applications (MADBench2 [8], Chombo I/O benchmark [9], S3D IO [27] and HOMME [26]). Furthermore, in our previous work [14] we were able to verify the periodicity of gyrokinetic toroidal code (GTC) [13], Enzo [6], HACC application [15] and CM1 [5].

Unfortunately, few documents give the actual values for $w^{(k)}$, $\text{vol}_{\text{io}}^{(k)}$ and $\beta^{(k)}$. Liu et al. [23] provide different periodic patterns of four scientific applications: PlasmaPhysics, Turbulence1, Astrophysics and Turbulence2. They were also the top four write-intensive jobs run on Intrepid in 2011. We chose the most I/O intensive patterns for all applications (as they are the most likely to create I/O congestion). We present these results in Table 1. Note that to scale those values to our system, we divided the number of nodes $\beta^{(k)}$ by 64, hence increasing $w^{(k)}$ by 64. The I/O volume stays constant.

To compare our strategy, we tried all possible combinations of those applications such that the number of nodes used equals 640. That is a total of ten different scenarios that we report in Table 2.

### 4.3 Baseline and Evaluation of Existing Degradation

We ran all scenarios on Jupiter without any additional scheduler. In all tested scenarios congestion occurred and decreased the visible bandwidth used by each applications as well as significantly increased the total execution time. We present in Table 3 the average I/O bandwidth slowdown due to congestion for the most representative scenarios together with the corresponding values for SysEfficiency. Depending on the I/O transfers per computation ratio of each application as well as how the transfers of multiple applications overlap, the slowdown in the perceived bandwidth ranges between 25% to 65%.

**Table 3.** Bandwidth slowdown, performance and application slowdown for each set of experiments

| Set # | Application | BW slowdown | SysEfficiency |
|---|---|---|---|
| 1 | Turbulence 2 | 65.72% | 0.064561 |
| 2 | Turbulence 2 | 63.93% | 0.250105 |
| | AstroPhysics | 38.12% | |
| 3 | Turbulence 2 | 56.92% | 0.439038 |
| | AstroPhysics | 30.21% | |
| 4 | Turbulence 2 | 34.9% | 0.610826 |
| | AstroPhysics | 24.92% | |
| 6 | Turbulence 2 | 34.67% | 0.621977 |
| | AstroPhysics | 52.06% | |
| 10 | Turbulence 1 | 11.79% | 0.98547 |
| | AstroPhysics | 21.08% | |

Interestingly, set 1 presents the worst degradation. This scenario is running concurrently ten times the same application, which means that the I/O for all applications are executed almost at the same time (depending on the small differences in CPU execution time between nodes). This scenario could correspond to coordinated checkpoints for an application running on the entire system. The degradation in the perceived bandwidth can be as high as 65% which considerably increases the time to save a checkpoint. The use of I/O schedulers can decrease this cost, making the entire process more efficient.

### 4.4 Comparison to Online Algorithms

In this subsection, we present the results obtained by running PerSched and the online heuristics from our previous work [14]. Because in [14] we had different

heuristics to optimize either Dilation or SysEfficiency, in this work, the Dilation and SysEfficiency presented are the best reached by *any* of those heuristics. This means that *there are no online solution able to reach them both at the same time*! We show that even in this scenario, our algorithm outperforms simultaneously these heuristics *for both optimization objectives*!

The results presented in [14] represent the state of the art in what can be achieved with online schedulers. Other solutions show comparable results, with [34] presenting similar algorithms but focusing on dilation and [11] having the extra limitation of allowing the scheduling of only two applications.

PerSched takes as input a list of applications, as well as the parameters, presented in Sect. 3, $K' = \frac{T_{\max}}{T_{\min}}$, $\varepsilon$. All scenarios were tested with $K' = 10$ and $\varepsilon = 0.01$.

*Simulation results.* We present in Table 4 all evaluation results. The results obtained by running Algorithm 1 are called PerSched. To go further in our evaluation, we also look for the best Dilation obtainable with our pattern (we do so by changing line 15 of PerSched). We call this result *min* Dilation in Table 4. This allows us to estimate how far the Dilation that we obtain is from what we can do. Furthermore, we can compute an upper bound to SysEfficiency by replacing $\tilde{\rho}^{(k)}$ by $\rho^{(k)}$ in Eq. (1):

$$\text{Upper bound} = \frac{1}{N} \sum_{k=1}^{K} \frac{\beta^{(k)} w^{(k)}}{w^{(k)} + \text{time}_{\text{io}}^{(k)}}. \tag{5}$$

The first noticeable result is that PerSched almost always outperforms (when it does not, matches) both the Dilation and SysEfficiency attainable by the online scheduling algorithms! This is particularly impressive as these

**Table 4.** Best Dilation and SysEfficiency for our periodic heuristic and online heuristics.

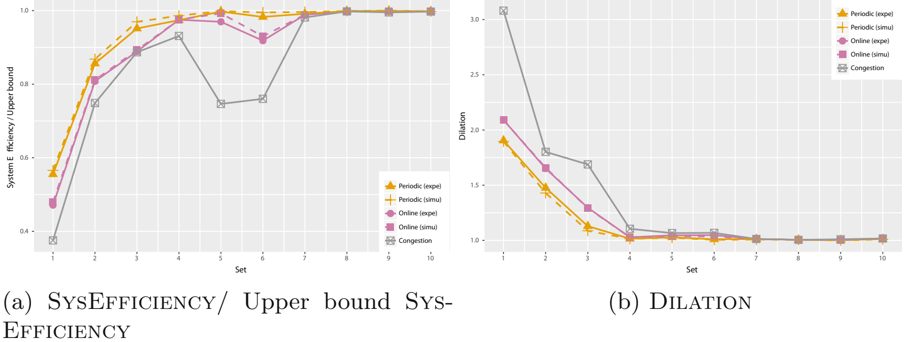
| Set | Min Dilation | Upper bound SysEff | PerSched | | Online | |
|---|---|---|---|---|---|---|
| | | | Dilation | SysEff | Dilation | SysEff |
| 1 | 1.777 | 0.172 | 1.896 | 0.0973 | 2.091 | 0.0825 |
| 2 | 1.422 | 0.334 | 1.429 | 0.290 | 1.658 | 0.271 |
| 3 | 1.079 | 0.495 | 1.087 | 0.480 | 1.291 | 0.442 |
| 4 | 1.014 | 0.656 | 1.014 | 0.647 | 1.029 | 0.640 |
| 5 | 1.010 | 0.816 | 1.024 | 0.815 | 1.039 | 0.810 |
| 6 | 1.005 | 0.818 | 1.005 | 0.814 | 1.035 | 0.761 |
| 7 | 1.007 | 0.827 | 1.007 | 0.824 | 1.012 | 0.818 |
| 8 | 1.005 | 0.977 | 1.005 | 0.976 | 1.005 | 0.976 |
| 9 | 1.000 | 0.979 | 1.000 | 0.979 | 1.004 | 0.978 |
| 10 | 1.009 | 0.988 | 1.009 | 0.986 | 1.015 | 0.985 |

objectives are not obtained by the same online algorithms (hence conjointly), contrarily to the PerSched result.

While the gain is minimal (from 0 to 3%, except SysEfficiency increased by 7% for case 6) when little congestion occurs (cases 4 to 10), the gain is between 9% and 16% for Dilation and between 7% and 18% for SysEfficiency when congestion occurs (cases 1, 2, 3)!

The value of $\varepsilon$ has been chosen so that the computation stays short. It seems to be a good compromise as the results are good and the execution times vary from 4 ms (case 10) to 1.8 s (case 5) using a Intel Core I7-6700Q. Note that the algorithm is easily parallelizable, as each iteration of the loop is independent. Thus it may be worth considering a smaller value of $\varepsilon$, but we expect no big improvement on the results.

*Model validation through experimental evaluation.* We used the modified IOR benchmark to reproduce the behavior of applications running on HPC systems and analyze the benefits of I/O schedulers. We made experiments on the 640 cores of the Jupiter system. Additionally to the results from both periodic and online heuristics, we present the performance of the system with no additional I/O scheduler.




(a) SysEfficiency/ Upper bound SysEfficiency

(b) Dilation

**Fig. 6.** Performance for both experimental evaluation and theoretical (simulated) results. The performance estimated by our model is accurate within 3.8% for periodic schedules and 2.3% for online schedules.

Figure 6 shows the SysEfficiency (normalized using the upper bound in Table 4) and Dilation when using the periodic scheduler in comparison with the online scheduler. The results when applications are running without any scheduler are also shown. As observed in the previous section, the periodic scheduler gives better or similar results to the best solutions that can be returned by the online ones, in some cases increasing the system performance by 18% and the dilation by 13%. When we compare to the current strategy on Jupiter, the SysEfficiency reach 48%! In addition, the periodic scheduler has the benefit of

not requiring a global view of the execution of the applications at every moment of time (by opposition to the online scheduler).

Finally, a key information from those results is the precision of our model introduced in Sect. 2. The theoretical results (based on the model) are within 3% of the experimental results!

*This observation is key in launching more thorough evaluation via extensive simulations and is critical in the experimentation of novel periodic scheduling strategies.*

*Synthetic applications.* The previous experiments showed that our model can be used to simulate real life machines[1]. In this next step, we now rely on synthetic applications and simulation to test extensively the efficiency of our solution.

We considered two platforms (Intrepid and Mira) to run the simulations with concrete values of bandwidths $(B, b)$ and number of nodes $(N)$. The values are reported in Table 5.

**Table 5.** Bandwidth and number of nodes of each platform used for simulations.

| Platform | $B$ (GB/s) | $b$ (GB/s) | $N$ | GFlops/node |
|---|---|---|---|---|
| Intrepid | 64 | 0.0125 | 40,960 | 2.87 |
| Mira | 240 | 0.03125 | 49,152 | 11.18 |

The parameters of the synthetic applications are generated as followed:

- $w^{(k)}$ is chosen uniformly at random between 2 and 7500 s for Intrepid (and between 0.5 and 1875 s for Mira whose nodes are about 4 times faster than Intrepid's nodes),
- the volume of I/O data $\text{vol}_{\text{io}}^{(k)}$ is chosen uniformly at random between 100 GB and 35 TB.

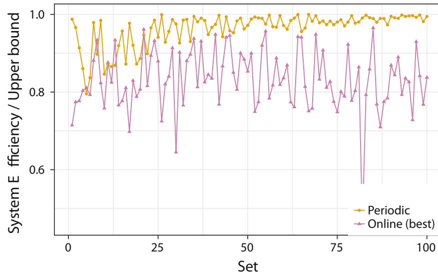
These values where based on the applications we previously studied.

We generate the different sets of applications using the following method: let $n$ be the number of unused nodes. At the beginning we set $n = N$.

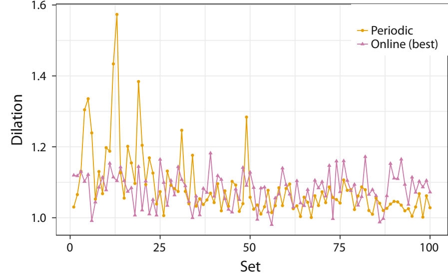
1. Draw uniformly at random an integer number $x$ between 1 and $\max(1, \frac{n}{4096} - 1)$ (to ensure there are at least two applications).
2. Add to the set an application $\text{App}^{(k)}$ with parameters $w^{(k)}$ and $\text{vol}_{\text{io}}^{(k)}$ set as previously detailed and $\beta^{(k)} = 4096x$.
3. $n \leftarrow n - 4096x$.
4. Go to step 1 if $n > 0$.

[1] Note that in our previous work [14] we already showed that this model was also fitting Intrepid and Mira.

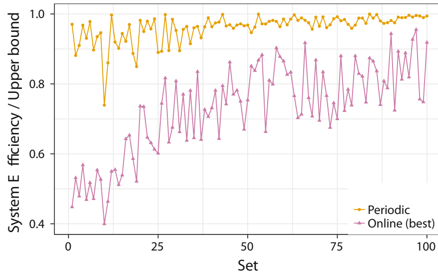
We then generated 100 sets for Intrepid (using a total of 40,960 nodes) and 100 sets for Mira (using a total of 49,152 nodes) on which we run the online algorithms (either maximizing the system efficiency or minimizing the dilation) and PerSched. The results are presented on Figs. 7a and b for simulations using the Intrepid settings and Figs. 7c and d for simulations using the Mira settings.



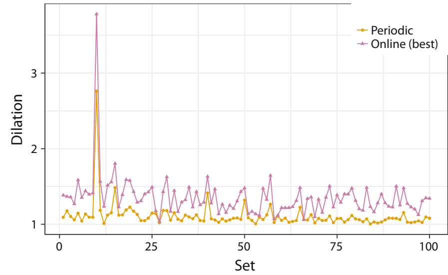

(a) Intrepid - SysEfficiency/ Upper bound SysEfficiency




(b) Intrepid - Dilation




(c) Mira - SysEfficiency/ Upper bound SysEfficiency




(d) Mira - Dilation

**Fig. 7.** Comparison between online heuristics and PerSched on synthetic applications.

We can see that overall, our algorithm increases the system efficiency in almost every case. On average the system efficiency is improved by 16% on intrepid (32% on Mira) with peaks up to 116%! On Intrepid the dilation has overall similar values (an average of 0.6% degradation over the best online algorithm, with variation between 11% improvement and 42% degradation). However on Mira in addition to the improvement in system efficiency, PerSched improves on average by 22% the dilation!

The main difference between Intrepid and Mira is the ratio *compute* over *I/O bandwidth*, that is the speed at which data is created/used over the speed at which data is transfered. This ratio increases a lot (and hence incurring more I/O congestion) on Mira. Hence we expect our algorithm to be a lot more efficient on systems where congestion is even more critical.

*These two experiments show two things: (i) our algorithm improves a lot the system efficiency compared to the online algorithms, without degrading too much the dilation and (ii) our algorithm is expected to scale extremely well, that is when the computing power increases faster than the bandwidth of the platform, as we can see from the results on Mira.*

## 5 Related Work

Performance variability due to resource sharing can significantly detract from the suitability of a given architecture for a workload as well as from the overall performance realized by parallel workloads [31]. Over the last decade there have been studies to analyze the sources of performance degradation and several solutions have been proposed. In this section, we first detail some of the existing work that copes with I/O congestion and then we present some of the theoretical literature that is similar to our Periodic problem.

The storage I/O stack of current HPC systems has been increasingly identified as a performance bottleneck. Significant improvements in both hardware and software need to be addressed to overcome oncoming scalability challenges. The study in [19] argues for making data staging coordination driven by generic cross-layer mechanisms that enable global optimizations by enforcing local decisions at node granularity at individual stack layers.

While many other studies suggest that I/O congestion is one of the main problems for future scale platforms [4,25], few papers focus on finding a solution at the platform level. Some papers consider application-side I/O management and transformation (using aggregate nodes, compression etc.) [24,32,33]. We consider those work to be orthogonal to our work and able to work jointly. Recently, numerous works focus on using machine learning for auto tuning and performance studies [3,21]. However these solution also work at the application level, do not have a global view of the I/O requirements of the system and they need to be supported by a platform level I/O management for better results.

Some papers consider the use of burst buffers to reduce I/O congestion by delaying accesses to the file storage, as they found that congestion occurs on a short period of time and the bandwidth to the storage system is often underutilized [23]. Note that because the computation power increases faster than the I/O bandwidth, this assumption may not hold in the future and the bandwidth may tend to be saturated more often and thus decreasing the efficiency of burst buffers. [20] presents a dynamic I/O scheduling at the application level using burst buffers to stage I/O and to allow computations to continue uninterrupted. They design different strategies to mitigate I/O interference, including partitioning the PFS, which reduces the effective bandwidth non-linearly. For now, these strategies are designed for only two applications.

The study from [28] offers ways of isolating the performance experienced by applications of one operating system from variations in the I/O request stream characteristics of applications of other operating systems. While their solution cannot be applied to HPC systems, the study offers a way of controlling the

coarse grain allocation of disk time to the different operating system instances as well as determining the fine-grain interleaving of requests from the corresponding operating systems to the storage system.

Closer to this work, online schedulers for HPC systems were developed such as our previous work [14], the study by Zhou et al. [34], and a solution proposed by Dorier et al. [11]. In [11], the authors investigate the interference of two applications and analyze the benefits of interrupting or delaying either one in order to avoid congestion. Unfortunately their approach cannot be used for more than two applications. Another main difference with our previous work is the light-weight approach of this study where the computation is only done once.

Our previous study [14] is more general by offering a range of options to schedule each I/O performed by an application. Similarly, the work from [34] also utilizes a global job scheduler to mitigate I/O congestion by monitoring and controlling jobs' I/O operations on the fly. Unlike online solutions, this paper focuses on a decentralized approach where the scheduler is integrated into the job scheduler and computes ahead of time, thus overcoming the need to monitor the I/O traffic of each application at every moment of time.

As a scheduling problem, our problem is somewhat close to the cyclic scheduling problem (we refer to Hanen and Munier [16] for a survey) and periodic scheduling problems [2,29]. Namely there are given a set of activities with time dependency between consecutive tasks stored in a DAG that should be executed on $N$ nodes. The main difference is that in cyclic scheduling there is no consideration of a constant time between the end of the previous instance and the next instance. More specifically, if an instance of an application has been delayed, the next instance of the same application is not delayed by the same time. With our model this could be interpreted as not overlapping I/O and computation.

## 6 Conclusion

Performance variation due to resource sharing in HPC systems is a reality and I/O congestion is currently one of the main causes of degradation. Current storage systems are unable to keep up with the amount of data handled by all applications running on an HPC system, either during their computation or when taking checkpoints. In this document we have presented a novel I/O scheduling technique that offers a decentralized solution for minimizing the congestion due to application interference. Our method takes advantage of the periodic nature of HPC applications by allowing the job scheduler to pre-define each application's I/O behavior for their entire execution. Recent studies [12] have shown that HPC applications have predictable I/O patterns even when they are not completely periodic, thus we believe our solution is general enough to easily include the large majority of HPC applications.

We conducted simulations for different scenarios and made experiments to validate our results. Decentralized solutions are able to improve both total system efficiency by 32% and application dilation by 22% simultaneously compared to dynamic state-of-the-art schedulers. Moreover, they do not require a constant

daemon capable of monitoring the state of all applications, nor do they require a change in the current I/O stack. One particularly interesting result is for scenario 1 with 10 identical periodic behaviors (such as what can be observed with periodic checkpointing for fault-tolerance). In this case the periodic scheduler shows a 30% improvement in SysEfficiency. Thus, system wide applications taking global checkpoints could benefit from such a strategy.

*Future work:* We believe this work is the initialization of a new set of techniques to deal with the I/O requirements of HPC system. In particular, by showing the efficiency of the periodic technique on simple pattern, we expect to open a door to multiple extensions. We give here some examples that we will consider in the future. The next natural directions is to take more complicated periodic shapes for applications (an instance could be composed of sub-instances) as well as different points of entry inside the job scheduler (multiple I/O nodes). This would be modifying the Insert-In-Pattern procedure and we expect that this should work well as well. Another future step would be to study how variability in the compute or I/O volumes impact a periodic schedule or the impact of non periodic applications. Finally we plan to model burst buffers and to show how to use them conjointly with periodic schedules.

Our method is used for minimizing the congestion caused by concurrent I/O accesses. However, the methodology and concepts are general and can be applied to any resource sharing problem. We will continue to investigate the causes for performance degradation in HPC applications and adapt our findings to each case.

**Acknowledgement.** This work was supported in part by the ANR Dash project. Part of this work was done when Guillaume Aupy and Valentin Le Fèvre were in Vanderbilt University. The authors would like to thank Anne Benoit and Yves Robert for helpful discussions.

## References

1. Aupy, G., Gainaru, A., Le Fèvre, V.: Periodic I/O scheduling for super-computers. Research report 9037, Inria Bordeaux Sud-Ouest (2017)
2. Baruah, S.K., Gehrke, J.E., Plaxton, C.G.: Fast scheduling of periodic tasks on multiple resources. In: Proceedings of the 9th International Parallel Processing Symposium, pp. 280–288. IEEE (1995)
3. Behzad, B., et al.: Taming parallel I/O complexity with auto-tuning. In: Proceedings of SC 2013 (2013)
4. Biswas, R., Aftosmis, M., Kiris, C., Shen, B.-W.: Petascale computing: impact on future NASA missions. In: Petascale Computing: Architectures and Algorithms, pp. 29–46 (2007)
5. Bryan, G.H., Fritsch, J.M.: A benchmark simulation for moist nonhydrostatic numerical models. Mon. Weather Rev. **130**(12), 2917–2928 (2002)
6. Bryan, G.L., et al.: Enzo: an adaptive mesh refinement code for astrophysics. arXiv:1307.2265 (2013)
7. Carns, P., et al.: 24/7 characterization of petascale I/O workloads. In: Proceedings of CLUSTER 2009, pp. 1–10. IEEE (2009)

8. Carter, J., Borrill, J., Oliker, L.: Performance characteristics of a cosmology package on leading HPC architectures. In: Bougé, L., Prasanna, V.K. (eds.) HiPC 2004. LNCS, vol. 3296, pp. 176–188. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30474-6_23
9. Colella, P., et al.: Chombo infrastructure for adaptive mesh refinement (2005). https://seesar.lbl.gov/ANAG/chombo/
10. Daly, J.T.: A higher order estimate of the optimum checkpoint interval for restart dumps. FGCS **22**(3), 303–312 (2004)
11. Dorier, M., Antoniu, G., Ross, R., Kimpe, D., Ibrahim, S.: CALCioM: mitigating I/O interference in HPC systems through cross-application coordination. In: Proceedings of IPDPS (2014)
12. Dorier, M., Ibrahim, S., Antoniu, G., Ross, R.: Omnisc'IO: a grammar-based approach to spatial and temporal I/O patterns prediction. In: SC, pp. 623–634. IEEE Press (2014)
13. Ethier, S., Adams, M., Carter, J., Oliker, L.: Petascale parallelization of the gyrokinetic toroidal code. In: VECPAR 2012 (2012)
14. Gainaru, A., Aupy, G., Benoit, A., Cappello, F., Robert, Y., Snir, M.: Scheduling the I/O of HPC applications under congestion. In: IPDPS, pp. 1013–1022. IEEE (2015)
15. Habib, S., et al.: The universe at extreme scale: multi-petaflop sky simulation on the BG/Q. In: Proceedings of SC 2012, p. 4. IEEE Computer Society (2012)
16. Hanen, C., Munier, A.: Cyclic scheduling on parallel processors: an overview. Citeseer (1993)
17. Harrod, B.: Big data and scientific discovery (2014)
18. Hu, W., Liu, G.-M., Li, Q., Jiang, Y.-H., Cai, G.-L.: Storage wall for exascale supercomputing. Front. Inf. Technol. Electron. Eng. **17**, 1154–1175 (2016). Zhejiang University Press
19. Isaila, F., Carretero, J.: Making the case for data staging coordination and control for parallel applications. In: Workshop on Exascale MPI at Supercomputing Conference (2015)
20. Kougkas, A., Dorier, M., Latham, R., Ross, R., Sun, X.-H.: Leveraging burst buffer coordination to prevent I/O interference. In: IEEE International Conference on eScience. IEEE (2016)
21. Kumar, S., et al.: Characterization and modeling of PIDX parallel I/O for performance optimization. In: SC. ACM (2013)
22. Lazzarini, A.: Advanced LIGO data & computing (2003)
23. Liu, N., et al.: On the role of burst buffers in leadership-class storage systems. In: MSST/SNAPI (2012)
24. Lofstead, J., et al.: Managing variability in the IO performance of petascale storage systems. In: SC. IEEECS (2010)
25. Lofstead, J., Ross, R.: Insights for exascale IO APIs from building a petascale IO API. In: Proceedings of SC 2013, p. 87. ACM (2013)
26. Nair, R., Tufo, H.: Petascale atmospheric general circulation models. J. Phys. Conf. Ser. **78**, 012078 (2007). IOP Publishing
27. Sankaran, R., et al.: Direct numerical simulations of turbulent lean premixed combustion. J. Phys. Conf. Ser. **46**, 38 (2006). IOP Publishing
28. Seelam, S.R., Teller, P.J.: Virtual I/O scheduler: a scheduler of schedulers for performance virtualization. In: Proceedings VEE, pp. 105–115. ACM (2007)
29. Serafini, P., Ukovich, W.: A mathematical model for periodic scheduling problems. SIAM J. Discret. Math. **2**(4), 550–581 (1989)

30. Shan, H., Shalf, J.: Using IOR to analyze the I/O performance for HPC platforms. In: Cray User Group (2007)
31. Skinner, D., Kramer, W.: Understanding the causes of performance variability in HPC workloads. In: IEEE Workload Characterization Symposium, pp. 137–149 (2005)
32. Tessier, F., Malakar, P., Vishwanath, V., Jeannot, E., Isaila, F.: Topology-aware data aggregation for intensive I/O on large-scale supercomputers. In: Proceedings of the First Workshop on Optimization of Communication in HPC, pp. 73–81. IEEE Press (2016)
33. Zhang, X., Davis, K., Jiang, S.: Opportunistic data-driven execution of parallel programs for efficient I/O services. In: Proceedings of IPDPS, pp. 330–341. IEEE (2012)
34. Zhou, Z., Yang, X., Zhao, D., Rich, P., Tang, W., Wang, J., Lan, Z.: I/O-aware batch scheduling for petascale computing systems. In: 2015 IEEE International Conference on Cluster Computing, pp. 254–263, September 2015

# A Performance Study of Quantum ESPRESSO's PWscf Code on Multi-core and GPU Systems

Joshua Romero[1(✉)], Everett Phillips[1], Gregory Ruetsch[1], Massimiliano Fatica[1], Filippo Spiga[2], and Paolo Giannozzi[3]

[1] NVIDIA Corporation, Santa Clara, USA
joshr@nvidia.com
[2] Research Computing Service, University of Cambridge, Cambridge, UK
[3] Dip. Scienze Matematiche Informatiche e Fisiche, University of Udine, Udine, Italy

**Abstract.** We describe the porting of PWscf (Plane-Wave Self Consistent Field), a key component of the Quantum ESPRESSO open-source suite of codes for materials modeling, to GPU systems using CUDA Fortran. Kernel loop directives (CUF kernels) have been extensively used in order to have a single source code for both CPU and GPU implementations. The results of the GPU version have been carefully validated and the performance of the code on several GPU systems (both x86 and POWER8 based) has been compared with traditional Intel multi-core (CPU only) systems. This current GPU version can reduce the time-to-solution by an average factor of 2–3 running two different input cases widely used as benchmarks on small and large high performance computing systems.

**Keywords:** DFT · Materials science · Eigensolver · GPU computing CUDA Fortran

## 1 Introduction

Computer simulations of materials, in particular *first-principle* simulations based on density-functional theory [9,13], pseudo-potentials, and plane-wave basis sets [14], have become widespread in many fields of science as well as in industry. These applications are run on a variety of computing systems, from desktop PCs to very large parallel machines, depending on the physical system under investigation and the property to be computed. The search for better methodologies and for better algorithms is a very active field of research.

Among the various packages implementing first-principle techniques, we focus on Quantum ESPRESSO (QE) [6], an integrated suite of open-source software released under the terms of the GNU General Public License (GPL). Programs included in QE can perform many different kinds of calculations. The complete distribution consists of approximately 520,000 lines of Fortran 95 source code, some additional code written in C, auxiliary scripts, and Python utilities.

© Springer International Publishing AG 2018
S. Jarvis et al. (Eds.): PMBS 2017, LNCS 10724, pp. 67–87, 2018.
https://doi.org/10.1007/978-3-319-72971-8_4

Due to accuracy requirements in electronic-structure computations, double precision floating point arithmetic is always used. In this study, we will concentrate on the PWscf code which solves self-consistently the Kohn-Sham equations arising in density-functional theory.

QE is designed to work on a variety of computing architectures and has evolved into a complex application with multiple layers of parallelism and key dependencies on mathematical libraries. The suite is able to run in serial and in parallel, targeting multi-core systems via multi-threaded libraries and explicit OpenMP and distributed systems using the Message Passing Interface (MPI) [12] and parallel libraries such as ScaLAPACK [2] or ELPA [11]. QE also supports modern approaches to effectively exploit multi-core and many-core architectures via hybrid parallelism based on MPI and OpenMP combined [17].

The need to accelerate time to discovery and tackle bigger and more challenging problems has motivated the first porting of QE to the programmable Graphics Processing Unit (GPU). GPUs are remarkable pieces of technology that have evolved into highly parallel many-core processors with floating-point performance and memory bandwidth that far exceed that of today's central processing units (CPUs). GPUs are especially well suited to address problems that can be expressed as data-parallel computations, where the same program is executed on different data elements in parallel. The CUDA programming model developed by NVIDIA has become the de-facto standard in GPU computing.

Today, the highest performing GPUs available on the market, suitable for scientific computation in fields like materials science, computational fluid dynamics, astrophysics and many others, are those within the NVIDIA Pascal family. In this paper, we will focus our evaluation on several computing platforms based on NVIDIA Pascal P100. This GPU is available in both PCI and SMX2 form-factors, with slightly different technical specifications (such as peak memory bandwidth and peak floating-point throughput). It is now possible to program GPUs in several languages, from the original CUDA C to the new OpenACC directive based compilers. QE is written in Fortran 90, so the natural choices for a GPU port are either CUDA Fortran or OpenACC. We decided to use CUDA Fortran as the structure of the code allows for the extensive use of CUF kernels, making the effort comparable to an OpenACC port, while also retaining the possibility of using explicit CUDA kernels when needed. In addition, the explicit nature of data movement in CUDA Fortran allows us to better optimize the CPU/GPU data movement and network traffic.

An initial GPU version of QE was developed several years ago [18] written in CUDA C and bundled with the original Fortran source code. This version, still available for reference and performance comparison [16], has been discontinued due to the complexity of managing and maintaining a large code base of mixed Fortran and CUDA C. This original version offloaded only limited portions of the workload to GPUs. A brand new version compatible with QE version 6 has been developed from the ground-up based on CUDA Fortran, focused on delivering performance on both large-scale and dense GPU system configurations, with all significant computation carried out on GPUs. As a consequence, unlike the

original plugin, this new version requires the complete dataset to fit in GPU memory.

The following section will first introduce the CUDA programming model and then provide an overview of CUDA Fortran and some specific features used in the porting effort. A detailed guide of the CUDA Fortran language extensions and features used can be found in [4].

## 2 CUDA Programming Model and CUDA Fortran

CUDA-enabled GPUs can contain anything from a few to thousands of processor cores which are capable of running tens of thousands of threads concurrently. To allow for the same CUDA code to run efficiently on different GPUs with varying specifications, a hierarchy of resources exists both in physical hardware, and in available programming models. In hardware, the processor cores on a GPU are grouped into multiprocessors. The programming model mimics this grouping: a subroutine run on the device, called a kernel, is launched with a grid of threads grouped into thread blocks. Within a thread block, data can be shared between threads, and there is a fine-grained thread and data parallelism. Thread blocks run independently of one another, which allows for scalability in the programming model: each block of threads can be scheduled on any of the available multiprocessors within a GPU, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on a device with any number of multiprocessors. This scheduling is performed behind the scenes, the CUDA programmer needs only to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, where each sub-problem is solved cooperatively in parallel by all threads within the block.

The CUDA platform enables hybrid computing, where both the host (CPU and its memory) and device (GPU and its memory) can be used to perform computations. A typical sequence of operations for a simple CUDA Fortran code is:

- Declare and allocate host and device memory
- Initialize host data
- Transfer data from the host to the device
- Execute one or more kernels
- Transfer results from the device to the host

From a performance perspective, the bandwidth of the PCIe bus is over an order of magnitude less than the bandwidth between the device's memory and GPU, and therefore a special emphasis needs to be placed on limiting and hiding PCIe traffic. For MPI applications, data transfers between the host and device are required to transfer data between MPI processes. Therefore, the use of asynchronous data transfers, i.e. performing data transfers concurrently with computation, becomes mandatory. This will be discussed in detail in Sect. 5.

**Data Declaration, Allocation, and Transfers.** The first Fortran extension we discuss is the variable attribute `device` used when declaring data that resides in GPU memory. Such declarations can be allocatable. The `allocate()` command has been overloaded so allocation occurs on the device when the argument is declared with the `device` attribute. Similarly, the assignment operator has been overloaded to perform data transfers between host and device memory spaces.

The Fortran 2003 `sourced` allocation construct, `allocate(lhs, source=rhs)`, is also supported and extended. When `allocate` is invoked with the optional `source=` argument, `lhs` becomes a clone of `rhs`: it is allocated with the same shape of `rhs` and each element of `rhs` is copied into the corresponding element of `lhs`. In CUDA Fortran, if the `lhs` array was defined as a device array, `lhs` will be a GPU array and the content from the CPU array `rhs` will be copied over the PCIe bus to GPU memory.

The above methods of data transfer are all blocking transfers, in that control is not returned to the CPU thread until the transfer is complete. This is sufficient in many cases, but prevents the possibility of overlapping data transfers with computation on both the host and device. The CUDA API function `cudaMemcpyAsync()` and its variants can be used to perform asynchronous transfers between host and device which allows concurrent computation.

**Kernels.** Kernels, or subroutines that are executed on the device, are denoted using the `attributes(global)` function attribute. Kernels are typically invoked in host code just as any subroutine is called, with the exception that an additional *execution configuration* specifying the number of thread blocks and number of threads per thread block to be used is included. In the device code itself, the automatically defined variables `threadIdx`, `blockIdx`, `blockDim`, and `gridDim` can be used to map threads to data elements. Aside from this, kernel code looks similar to the subroutines in the host code. The difference is that the kernel code is executed by many threads in parallel.

**CUF Kernels.** CUDA Fortran can automatically generate and invoke kernel code from a region of host code containing tightly nested loops. Such code is referred to as a CUF kernel. A simple example of a CUF kernel is:

```
!$cuf kernel do <<<*,*>>>
 do i=1, n
  a_d(i) = a_d(i) + b
 enddo
```

where the directive indicates that the following loop has to be performed on the device. One can specify the execution configuration in the chevrons. In the example above we use wild-cards and let the runtime system determine these parameters. The arrays in CUF kernels, such as `a_d` above, are required to be device arrays; however, the scalar `b` can be a host variable which will be passed as a kernel argument by value.

One can port host code to the device using CUF kernels without modifying the contents of the loops using the following programming convention. If the arrays used in the loops are declared in a module, along with a device equivalent:

```
module m
 ...
 real :: a(n)
 real,device :: a_d(n)
 ...
end module
```

then the rename option to the `use` statement can be invoked to allow conditional execution of the code either on the host or device:

```
subroutine update
#ifdef USE_CUDA
 use m, only: a => a_d
#else
 use m, only: a
#endif
...
!$cuf kernel do <<<*,*>>>
 do i=1, n
  a(i) = a(i) + b
 enddo
...
```

If the arrays used in the loops are explicitly passed to the subroutine, the only change required is to add the `device` attribute:

```
subroutine update(a,n)
real:: a(n)
#ifdef USE_CUDA
attributes(device) :: a
#endif
...
!$cuf kernel do <<<*,*>>>
 do i=1, n
  a(i) = a(i) + b
 enddo
...
```

Note that here the contents of the loop are unaltered. The only changes to the host code are the conditional renaming of module variables or the additional `device` attribute and the CUF kernel directive. The directive will appear as a comment to the compiler if GPU code generation is disabled or if the compiler does not support them (similar to the OpenMP directives that are ignored if OpenMP is not enabled).

## 3 Profiling Using NVTX

Profiling is an essential tool to identify parts of the code that may require additional tuning. When dealing with GPU codes, profiling is even more important as new opportunities for better interactions between the CPUs and the GPUs can be discovered. The standard profiling tools in CUDA, `nvprof` and `nvvp`, are able to show the GPU timeline but do not present CPU activity. The NVIDIA Tools Extension (NVTX) is a C-based API (application program interface) to annotate the profiler time line with events and ranges and to customize their appearance and assign names to resources such as CPU threads and devices [10]. We have written a Fortran module to instrument CUDA/OpenACC Fortran codes using Fortran ISO C bindings [3]. Using this module is very simple: once the NVTX module is included, the developer only needs to mark the region of interest with `nvtxRangePush` and `nvtxRangePop` calls. Calls to `nvtxStartRange("text")` with a single argument will insert green markers with a *text* label in the timeline. Different colors can be selected using an optional integer parameter and the regions of interest can be nested.

Since QE already has a built-in performance report that summarizes the time spent in the important parts of the code, we added the NVTX calls to the timing functions. This allowed a minimal code change.

To eliminate profiling overhead during production runs, we use a preprocessor variable to make the profiling calls return immediately. During the runs, one or more MPI processes generate the traces that are later imported and visualized with `nvvp`, the NVIDIA Visual Profiler.

Figure 1 shows a typical output for a PWscf run (when the mouse rolls over the markers, it will indicate the name of the marker and information on the kernel configurations).

## 4 Structure of the PWscf Code

As noted in the introduction, QE is not a monolithic program but a modular suite of codes sharing common libraries and data structures. The two major packages that are the foundation of every material science simulation work-flow are PWscf (Plane-Wave Self-Consistent Field) and `CP` (Car-Parrinello).

In this GPU porting effort, PWscf has been the main focus. The basic computations of the PWscf code involve the calculation of the Kohn-Sham (KS) orbitals and energies for isolated or extended/periodic systems and the complete structural optimizations of the microscopic (atomic coordinates) and macroscopic (unit cell) degrees of freedom. The KS orbitals are quantum-mechanical states of electrons under an effective Kohn-Sham potential. The solution is *self-consistent*: the KS potential depends upon the KS orbitals via the charge density (the sum of the square moduli of Kohn-Sham orbitals). This non-linear problem can be solved with an iterative procedure (see [6], Appendix A.2). Figure 2 illustrates the main activities performed in a typical execution of PWscf, where both high-level structural optimization and self-consistency [8] are explored.

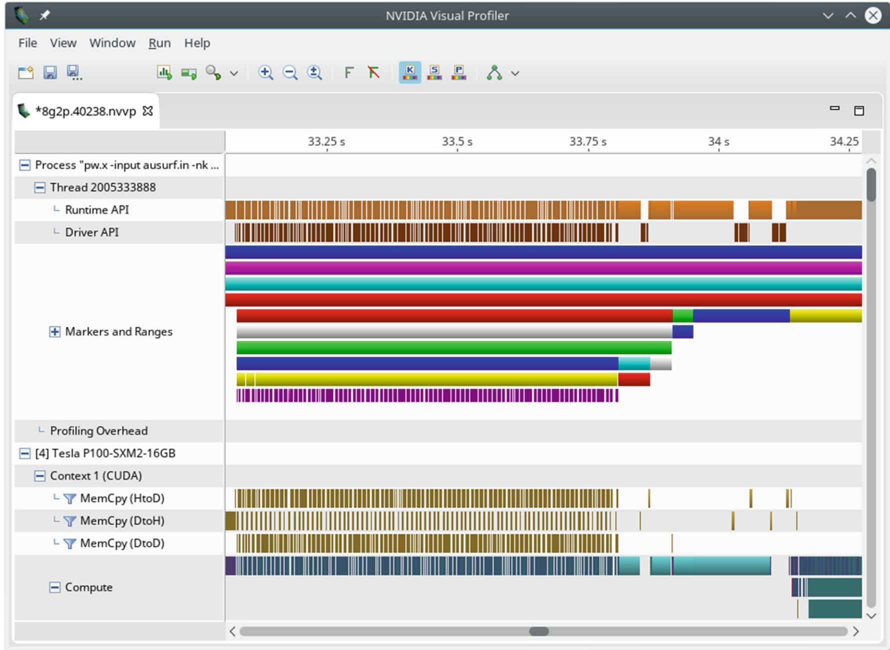



**Fig. 1.** Segment of `nvvp` output for AUSURF112 case on the DGX-1 system with 8 GPUs and no GPUDirect (GDR) features enabled. "Markers and Ranges" section contains colored markers corresponding to various NVTX ranges.

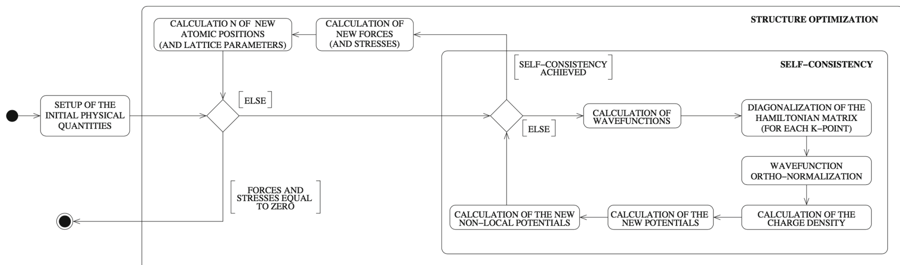



**Fig. 2.** Schematic view of PWscf internal steps.

In a plane-wave basis set, each KS orbital, $\psi$, is represented by a vector of plane-wave coefficients. The self-consistency loop is an iteration over the charge density, until input and output charge densities are the same within a predefined threshold. The output charge density is computed from KS orbitals, obtained by diagonalizing the matrix of the Hamiltonian operator, $H_{KS}$, which depends on the KS potential. By default, iterative diagonalization is completed using a block Davidson method. The calculation of the charge density requires all

occupied KS orbitals in the system. In a crystal, KS orbitals are classified by a Bloch vector, or "$k$-point", and by a "band" index. In practice, a discrete number of $k$-points, ranging from one to a few tens or hundreds at most, is needed [5]. The diagonalization is separately performed for each $k$-point. The number of occupied KS orbitals is determined by the number of electrons in the unit cell.

The iterative diagonalization and computation of charge density account for the majority of the time spent in the solver, with the remaining cost attributed to initialization and post processing routines. In the iterative diagonalization, the time-consuming step is the direct calculation of products $H_{KS}\psi$. Note that those products are *not* computed as matrix-vector products: the $H_{KS}$ matrix would be far too large for all but the simplest systems. Using the so-called dual-space technique, all computationally expensive terms can be expressed in terms of the following basic operations:

- 3-dimensional Fast Fourier Transforms (FFT);
- basic linear-algebra operations on vectors and matrices, in particular matrix-matrix multiplications (Level-3 BLAS);
- dense matrix diagonalization (LAPACK or ScaLAPACK).

The code offers a number of run-time options that affect the parallelization and enable distributed operation. A list of options used in this study includes:

- *k-point parallelization* using `-npool`: distributes the $k$-points into $N_K$ pools, allowing embarrassingly parallel execution of the iterative diagonalizations. If $N$ is the total number of MPI processes, there are $N_P = \frac{N}{N_K}$ processes per pool.
- *linear-algebra parallelization* using `-ndiag`: distributes the solution of the subspace diagonalization, needed by the block Davidson algorithm, to $N_D \leq N_P$ processes, enabling usage of ScaLAPACK or similar distributed linear solver library.

These options can be applied simultaneously, resulting in a wide array of possible combinations, not all valid or equally effective. The $k$-point parallelization takes precedence, splitting all available processes into equal pools. Within each pool, plane waves are distributed (this is also referred as *plane-wave* or *g-parallelism*). This distribution of plane waves across multiple MPI processes results in the need to perform parallel distributed 3D FFTs in order to transform physical quantities (KS orbitals, charge density and potentials) between reciprocal and real space. The FFT grids are generally of modest size (with dimensions in the hundreds); however, the FFT computation is repeated many times throughout the course of the calculation.

## 5 GPU Porting of Key Routines

While the full GPU porting effort involved the translation of a number routines in the original CPU code to GPU either by the use of CUF directives or CUDA kernels, we focus our discussion here on the routines that are considered most

performance critical. Without delving too deep into the specifics, it is informative to breakdown the major components of the PWscf iteration and identify the key computational operations involved. The iterative diagonalization involves the heavy use of three main computational components: a dense generalized eigensolver to diagonalize the subspace projected linear system, double-precision complex GEMMs which are mostly used to process the approximated eigenvalues and eigenvectors and expand the basis, and distributed forward and inverse 3D FFTs used in the procedure to compute the local potential term in $H_{KS}\psi$ for each unconverged band using the dual-space technique. The computation of the symmetrized charge density is dominated by the accumulation of wavefunction contributions to the charge from each $k$-point which involves numerous distributed forward 3D FFT computations, one for each band.

Of the operations identified, the matrix-matrix multiplications are the most straightforward and can be easily computed on GPU using the CUBLAS library. The porting effort of the other computational components is more involved and requires further discussion.

### 5.1 Forward and Inverse 3D FFTs

Forward and inverse 3D FFTs are required in both the iterative diagonalization process and the computation of charge. As such, they account for a large share of the total computational load. While the component-wise 1D FFT computations can be carried out on GPU using simple calls to the CUFFT library, the complete computation is typically distributed among a number of processes, requiring transposition and communication of data across processes.

Currently, QE uses a 1D decomposition of the domain to distribute the 3D FFTs. With this decomposition, a typical 3D FFT computation of dimension $NX \times NY \times NZ$, distributed across $N_P$ processes in the pool, is completed in the following steps:

1. Begin with contiguous columns of data along $z$-dimension. Each process contains a $NX/N_P \times NY \times NZ$ sized chunk of the domain. Perform 1D FFTs on the $z$-columns.
2. Transpose result into planes representation via `MPI_Alltoall` or similar communication pattern. After communication, each process contains a $NX \times NY \times NZ/N_P$ sized chunk of the domain.
3. Perform 2D FFTs on the $xy$-planes.

This process also occurs in reverse within the solver, but the forward description is sufficient for this discussion.

The existing CPU implementation of this distributed 3D FFT procedure is fairly basic, with a few characteristics making a direct translation to GPU low performing. The first of these characteristics is that the FFT computation is carried out in a loop over bands, with relatively small FFT computations for each band. These small FFT computations are problematic on GPUs due to the lack of available concurrent work to fully saturate the GPU resources,

leading to inefficient device utilization and possible losses due to latency. The second characteristic is that the existing procedure does not make any attempt to overlap MPI communication with computation. This is especially problematic for a GPU implementation where, when direct peer-to-peer access between GPUs is unavailable, MPI communication buffers must be staged through CPU memory. Therefore, in addition to efficiency losses due to non-overlapped MPI communication, there are additional losses attributed to data movement of the communication buffers between host and device memory.

To address these issues, a new batched FFT strategy was implemented for GPUs which processes the 3D FFTs for several bands together. By processing multiple 3D FFTs at a time, there is naturally more concurrent work available to fully saturate the GPUs which addresses the first issue with the original implementation. In addition to this, further separation of batches into smaller sub-batches yields an opportunity for pipelining data movement and computation between sub-batches which we leveraged in our implementation. As a further optimization, the all-to-all communication pattern was carried out using non-blocking `MPI_Isend` and `MPI_Irecv`. This is of particular importance on GPU systems with fully-connected subsets of GPUs via NVLink, like the DGX-1, where numerous peer-to-peer transfers can occur simultaneously via GPUDirect (GDR). A simple method to enable these concurrent peer-to-peer transfers is through the use of a CUDA-aware MPI distribution. With that being said, early experimentation indicated that several issues arise in a number of available MPI implementations of these features, leading to suboptimal utilization of available peer-to-peer bandwidth on systems with numerous peer-to-peer links. To address this, an explicit handling of peer-to-peer communication was implemented using CUDA inter-process communication (IPC) features, with non-peer transfers handled by the linked MPI library. Lastly, by finely controlling the all-to-all communication, self-to-self buffer transfers on the GPU can be handled specifically to avoid any unneeded use of host resources. It should be noted that batching the FFT computation does increase memory requirements, as multiple FFT domains must be resident in device memory. For the benchmark cases tested in this study, this was not a limiting factor; however, for larger cases, the batch size can be adjusted to fit within available memory.

### 5.2 Solving the Eigenproblem

The final major computational component to discuss is the dense eigensolver, which is used to solve the subspace projected problem generated through the Davidson iteration process. In the existing CPU implementation, the dense eigensolve can either be computed sequentially, using one process in a $k$-point pool group, or distributed across $N_D$ processes in the pool group using ScaLAPACK or a similar distributed linear algebra package.

The initial GPU port targets only the serial path, using a custom developed GPU eigensolver. A custom solver was chosen in lieu of several existing GPU-enabled eigensolvers, like those available in MAGMA [7]. The custom GPU eigensolver was developed to specifically limit dependencies on CPU resources, using

the CPU only for the solution of a reduced tridiagonal eigensystem using available functionality from Intel MKL or other LAPACK implementations. This is in contrast to implementations available in MAGMA, where many more operations are offloaded to the CPU, with a complex pipelining of CPU computation, GPU computation, and data movement between the host and device. This is especially beneficial on "fat" GPU nodes, nodes with a high ratio of GPU to CPU sockets, where available CPU resources (host memory bandwidth, PCIe bandwidth between host and device, available CPU FLOPS) per GPU can be limited. By limiting the use of CPU resources, the custom eigensolver can achieve more consistent performance across these types of node topologies, with less sensitivity to available CPU resources per GPU. Even with node topologies with one full CPU socket available per GPU, limiting these CPU dependencies has been shown to improve performance of the custom solver relative to MAGMA and MKL [15].

While only the serial eigensolver path has been ported, the results of several benchmark cases to be discussed in later sections will show that our custom eigensolver, even operating on a single GPU, provides competitive performance relative to high-performance distributed CPU solvers, like the ELPA solver [1].

## 6 Performance Comparison

Performance results were obtained on a number of GPU systems ranging in size from a small workstation containing only two GPUs up to several large GPU accelerated clusters, with reference CPU performance results obtained on a private development cluster.

The reference CPU system (labeled "Broadwell" in the results) is a private development system of a few hundred nodes fully based on Intel technology. Each node has dual socket 18-core Intel Xeon E5-2697 v4 (Broadwell) CPUs, 128 GB of system memory and one single Intel Omni-Path interconnect to provide 100 Gb/s connectivity for both parallel jobs and I/O.

The small systems used in this study were a workstation with a 6-core Intel Core i7-5930K CPU with two 16 GB NVIDIA P100 GPUs and an NVIDIA DGX-1 system. The DGX-1 contains dual socket 20-core Intel Xeon E5-2698 v4 (Broadwell) CPUs with eight 16 GB NVIDIA P100 GPUs, with fully-connected clusters of four GPUs with NVLink associated with each CPU socket.

The large GPU systems used in this study were Piz Daint at the Swiss National Supercomputing Centre (CSCS), SummitDev at the Oak Ridge National Laboratory (ORNL) and Wilkes-2 cluster at the University of Cambridge.

Piz Daint is a Cray XC50 with 5,272 nodes, each with a 12-core Intel Xeon E5-2690 v3 (Haswell) CPU, 64 GB of system memory and a 16 GB NVIDIA P100 GPU. The network uses Aries routing and communications ASICs and a dragonfly network topology. Piz Daint is currently number three on the June 2017 Top500 list with 19.59PF and is one of the most efficient petaFLOP class machines in the world: in the Green 500 list published in June 2017, the machine was able to achieve 10398 MFLOP/s/W with level 3 measurements, the most accurate available.

The SummitDev system is an early access system that is one generation removed from ORNL's next big supercomputer, Summit. The system has 54 IBM POWER8 S822LC nodes. Each node has dual socket IBM POWER8 CPUs, each with 10 cores and 80 HW threads, 256 GB of system memory, and four 16 GB NVIDIA P100 GPUs, with two NVLink connected GPUs per socket. In contrast to the Intel based systems, the GPUs on SummitDev are connected to the CPUs by NVLink 1.0 at 80 GB/s. The nodes are connected in a full fat-tree via EDR InfiniBand. SummitDev has access to Spider 2, the OLCF's center-wide Lustre parallel file system, and also local NVMe disks.

Wilkes-2 is a new GPU cluster at the University of Cambridge composed of 90 Dell PowerEdge C4130 compute nodes. Each node has a single socket 12-core Intel Xeon CPU E5-2650 v4 (Broadwell) CPU, 96 GB of system memory and four 16 GB NVIDIA P100 GPUs all connected to the same PCIe root complex. One single Mellanox Infiniband EDR card provides 100 Gb/s connectivity for both parallel jobs and access to the Lustre storage. Wilkes-2 is completely based on commodity hardware and it is currently number 100 on the June 2017 Top500 list with 1.193 PF and number 5 on the Green500 list with 10428 MFLOP/s/W.

### 6.1 Performance Analysis

**Benchmark Cases and Details.** For testing, two benchmark test cases were used which span a range of typical use cases for the PWscf solver. The cases used were:

- AUSURF112: computation of a surface of 112 gold atoms with two $k$-points. Small case suitable for testing on workstations and small distributed systems.
- Ta2O5: computation of tantalum pentoxide with 96 atoms and 26 $k$-points. Large case suitable for scaling from small to large distributed systems.

Detailed input specifications for these benchmark cases can be found in Table 1.

For cases run on GPU systems with Intel CPUs, multithreaded MKL was used for any BLAS and LAPACK routines computed using the CPU, including the tridiagonal eigensolve offloaded from the custom GPU eigensolver. On SummitDev, multithreaded ESSL was used in place of MKL; however, due to the lack of a linkable implementation of `ZSTEDC`, the CPU tridiagonal eigensolver routine we require for our GPU eigensolver, the program was linked against the LAPACK implementation provided with PGI, with underlying BLAS routines computed using ESSL.

For all runs on the reference CPU system, the eigenproblem is solved using the distributed ELPA library, with $N_D$ set to the closest square number to half the available MPI processes per pool group. Note that the number of available MPI processes per pool group is reduced if OpenMP threads are enabled. The results on the reference CPU system reported are the best-case results achieved using a variety of possible configurations of OpenMP threads and $N_D$ values.

For all runs on the GPU systems, $N_D$ is always set to one since only the serial eigensolver path was ported to GPU. For systems using Intel CPUs, OpenMP

**Table 1.** Benchmark case input specifications

| Parameter | Benchmark case | |
|---|---|---|
| | AUSURF112 | Ta2O5 |
| Number of atomic species | 1 | 2 |
| Number of atoms | 112 | 96 |
| Number of electrons | 1,232 | 544 |
| Number of Kohn-Sham states | 739 | 326 |
| Number of k-points | 2 | 26 |
| Number of plane waves | 100,747 | 477,247 |
| Kinetic energy cutoff | 25 Ry | 130 Ry |
| Charge density cutoff | 200 Ry | 520 Ry |
| Dimension of dense FFT grid | $\{180, 90, 288\}$ | $\{198, 168, 220\}$ |

threading was enabled to improve the offloaded CPU tridiagonal eigensolve using multithreaded MKL; as such, threads were distributed so that a larger portion of available cores were bound to processes within pool groups performing the serial eigensolve. On SummitDev, a similar thread distribution strategy was utilized with multi-threaded ESSL; however, OpenMP was disabled elsewhere in the code due to existing compatibility issues between the PGI and IBM OpenMP runtimes.

On GPU systems with available peer-to-peer connections between GPUs, the test cases were run both with and without using GPUDirect (GDR) features. For all communication except the all-to-all in the distributed FFTs, these features were enabled implicitly through the use of CUDA-aware MPI distributions, typically Open MPI or Cray MPICH on Piz Daint. On SummitDev, due to poor performance of the CUDA-aware features of Spectrum MPI, all MPI communication is staged through the host. For the all-to-all communication, peer-to-peer transfers were handled explicitly using our explicit CUDA IPC implementation, with non-peer transfers handled by the linked MPI library.

**Results and Discussion.** Performance results for the AUSURF112 test case can be found in Table 2, with timing breakdowns for the cases run with 4 GPUs or CPUs and cases run with 8 GPUs or CPUs plotted in Figs. 3 and 4 respectively. For accuracy considerations, the final converged total energy results on the reference CPU system for this test case were within the range $-11427.08997421$ Ry to $-11427.08997363$ Ry. This compares well with the converged total energy results obtained on the GPU systems, which ranged from $-11427.08997417$ Ry to $-11427.08997388$ Ry.

Similarly, performance results for the Ta2O5 test case across the tested systems can be found in Table 3, with timing breakdowns for the cases using 8 GPUs or CPUs, 104 GPUs or CPUs, and 208 GPUs or CPUs plotted in Figs. 5, 6 and 7 respectively. For this test case, the final converged total energy results

**Table 2.** PWscf time in seconds for AUSURF112 testcase

| System | $N_K$ | Number of CPUs or GPUs used | | | | |
|---|---|---|---|---|---|---|
| | | 2 | 4 | 8 | 16 | 32 |
| Broadwell (CPU) | 1 | 1142.24 | 642.03 | 369.66 | 272.00 | 266.20 |
| | 2 | 1190.13 | 586.84 | 335.00 | 196.54 | **144.07** |
| Piz Daint | 1 | 286.24 | 219.91 | 171.80 | – | – |
| | 2 | – | 149.21 | **115.87** | – | – |
| DGX-1 | 1 | 347.82 | 271.37 | 210.67 | – | – |
| | 2 | – | 184.10 | 142.15 | – | – |
| DGX-1, GDR | 1 | 270.21 | 190.12 | 174.75 | – | – |
| | 2 | – | 142.43 | **100.54** | – | – |
| Summit Dev | 1 | 321.69 | 234.32 | 187.69 | – | – |
| | 2 | – | 176.50 | 128.85 | – | – |
| Summit Dev, GDR | 1 | 308.52 | 227.74 | 188.39 | – | – |
| | 2 | – | 169.60 | **124.22** | – | – |
| Wilkes-2 | 1 | 395.26 | 326.71 | 227.61 | – | – |
| | 2 | – | 226.89 | 167.80 | – | – |
| Wilkes-2, GDR | 1 | 300.03 | 226.13 | 203.59 | – | – |
| | 2 | – | 164.63 | **116.50** | – | – |
| Workstation | 1 | 334.23 | – | – | – | – |
| Workstation, GDR | 1 | **279.54** | – | – | – | – |

on the reference CPU system were within the range −2370.63541806 Ry to −2370.63541801 Ry. This also compares well with the converged total energy results obtained on the GPU systems, which ranged from −2370.63541805 Ry to −2370.63541804 Ry.

Considering the tabulated performance results in Tables 2 and 3, several observations can be made. First, across most results for this case, it can be noted that for a fixed number of CPU or GPU resources, increasing $N_K$ provides a performance improvement. This indicates that the program on both CPU and GPU is more efficiently utilizing compute resources when there are fewer resources assigned per pool. If the program scaled perfectly with the number of resources per pool, the PWscf time, assuming the computation outside the scope of the pool parallelization is negligible, should remain nearly fixed if the number of pools is doubled. This is because the doubling of performance associated with processing more $k$-points concurrently would be counteracted by a halving in performance due to halving the number of compute resources per pool.

This reduction in efficiency can largely be attributed to the scaling characteristics of the distributed 3D FFT computations and the eigensolver. This can be observed clearly in the timing breakdowns plotted in Figs. 3, 4, 5, 6 and 7 when comparing the results for the different $N_K$ values on each system.

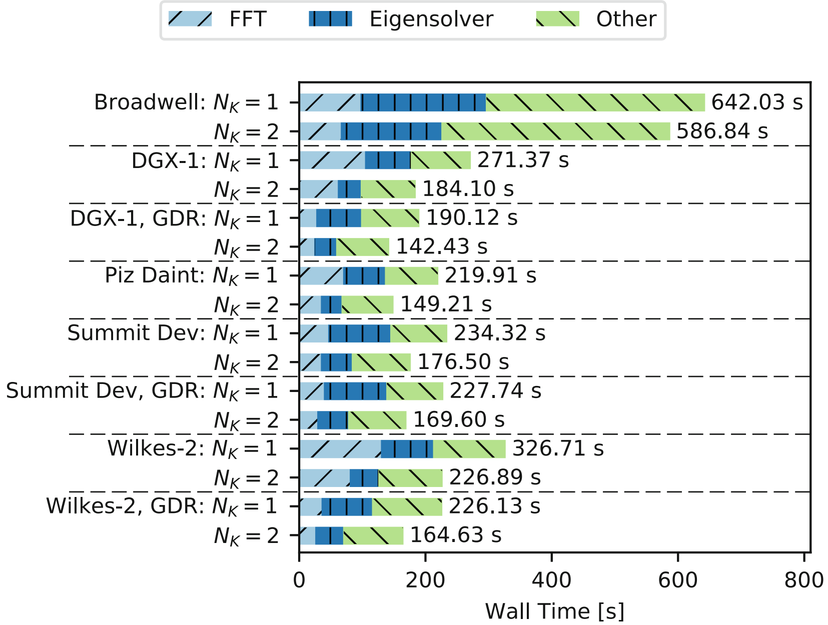



**Fig. 3.** Breakdown of PWscf time for AUSURF112 using 4 GPUs or CPUs by system and pool size.

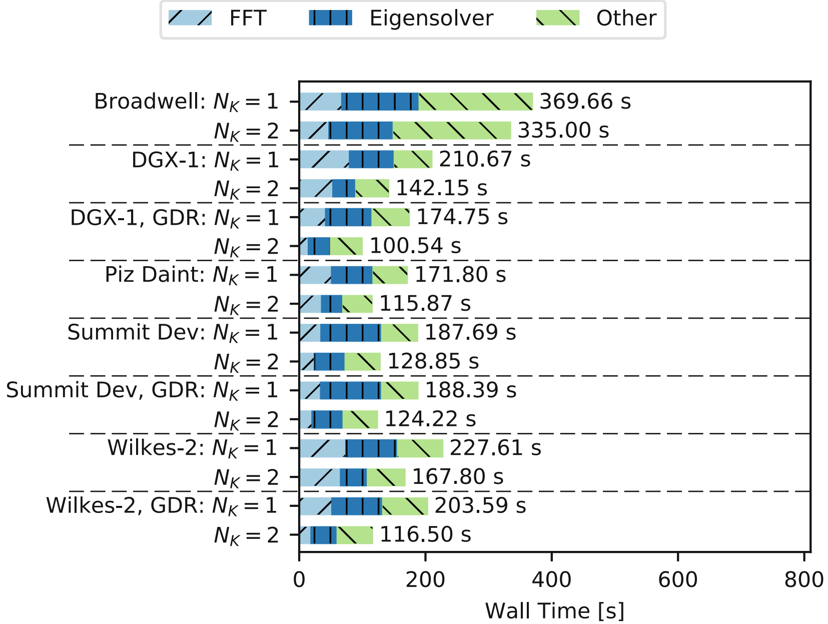



**Fig. 4.** Breakdown of PWscf time for AUSURF112 using 8 GPUs or CPUs by system and pool size.

**Table 3.** PWscf time in seconds for Ta2O5 testcase

| System | $N_K$ | Number of CPUs or GPUs used | | | | |
|---|---|---|---|---|---|---|
| | | 8 | 26 | 52 | 104 | 208 |
| Broadwell (CPU) | 13 | – | – | 1374.26 | 809.36 | 540.64 |
| | 26 | – | 3055.46 | 1566.95 | 682.05 | **378.73** |
| Piz Daint | 1 | 5273.93 | – | – | – | – |
| | 2 | 3602.07 | – | – | – | – |
| | 13 | – | – | 617.58 | 419.39 | 330.85 |
| | 26 | – | – | – | 315.60 | **217.29** |
| DGX-1 | 1 | 7253.06 | – | – | – | – |
| | 2 | 5008.94 | – | – | – | – |
| DGX-1, GDR | 1 | 4139.18 | – | – | – | – |
| | 2 | 2701.00 | – | – | – | – |
| Summit Dev | 1 | 4122.03 | – | – | – | – |
| | 2 | 3236.12 | – | – | – | – |
| | 13 | – | – | 581.15 | 394.62 | 289.30 |
| | 26 | – | – | – | 305.66 | 216.95 |
| Summit Dev, GDR | 1 | 3994.21 | – | – | – | – |
| | 2 | 2959.70 | – | – | – | – |
| | 13 | – | – | 544.83 | 398.91 | 292.87 |
| | 26 | – | – | – | 284.90 | **207.37** |
| Wilkes-2 | 1 | 7394.40 | – | – | – | – |
| | 2 | 6111.83 | – | – | – | – |
| | 13 | – | – | 1035.20 | 656.85 | – |
| | 26 | – | – | – | 515.78 | – |
| Wilkes-2, GDR | 1 | 5032.51 | – | – | – | – |
| | 2 | 3264.26 | – | – | – | – |
| | 13 | – | – | 572.43 | 460.16 | – |
| | 26 | – | – | – | **273.86** | – |

First, on both the CPU and GPU systems, the distributed FFT operates more efficiently when distributed across fewer processes. This is because distribution across fewer processes on the tested systems generally resulted in improved all-to-all communication performance. One reason for this performance improvement is that a smaller group of processes can maintain better locality, resulting in a larger percentage of communication occurring over higher bandwidth intra-node connections, either within local CPU memory, or through more direct NVLink or PCIe connections when using GDR features on GPU systems. In addition to this, with fewer processes, the self to self buffer involved in the all-to-all, which is a fast local memory movement, comprises a larger portion of the total communication volume.

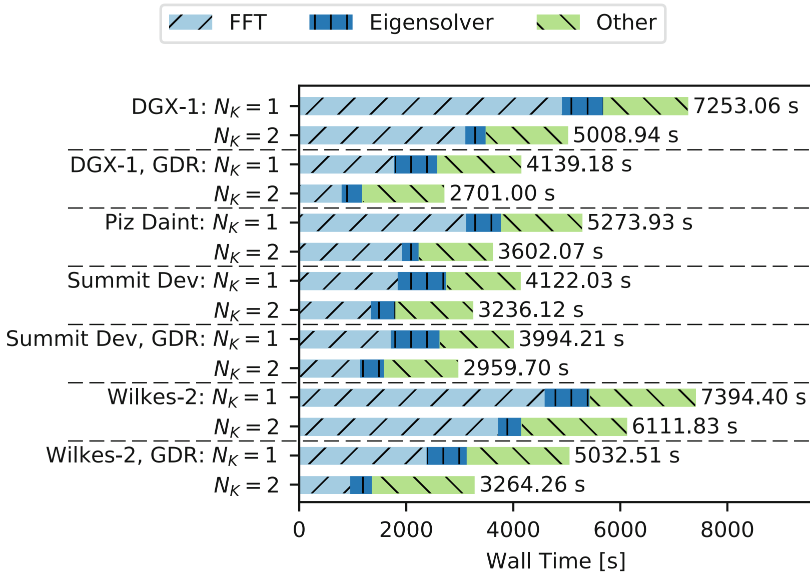



**Fig. 5.** Breakdown of PWscf time for Ta2O5 using 8 GPUs or CPUs by system and pool size.

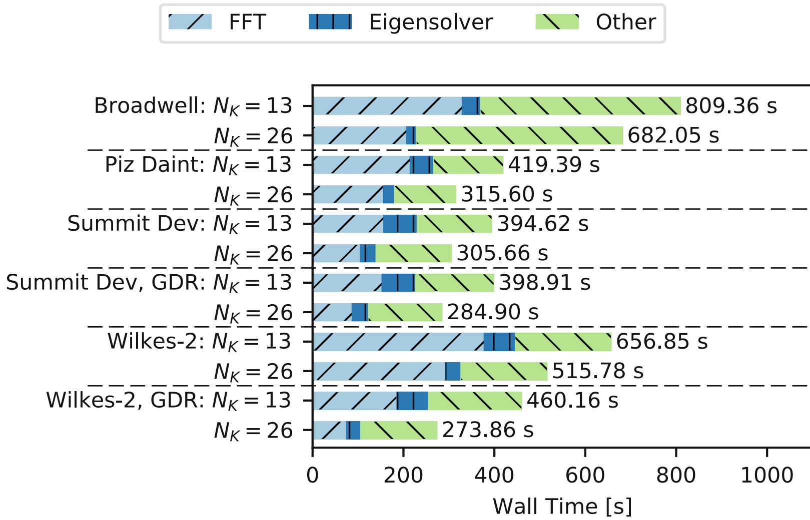



**Fig. 6.** Breakdown of PWscf time for Ta2O5 using 104 GPUs or CPUs by system and pool size.

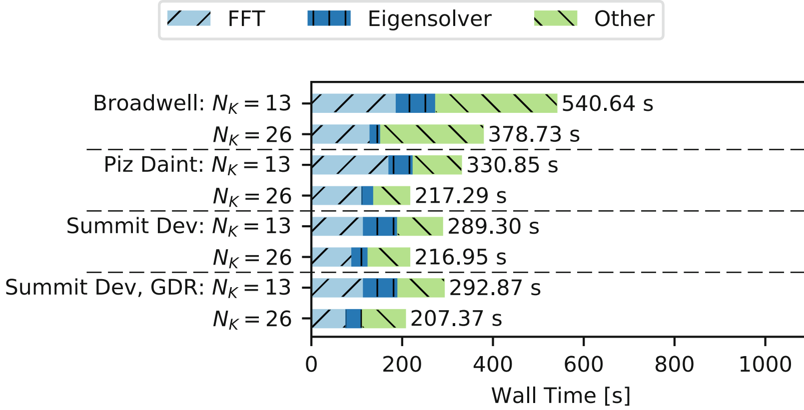



**Fig. 7.** Breakdown of PWscf time for Ta2O5 using 208 GPUs or CPUs by system and pool size.

Continuing on this point, the benefits of using GDR on the GPU systems with available peer-to-peer access can be substantial, with improved performance in most cases on systems with GDR features utilized. As expected, systems with more available peer-to-peer links between GPUs, like the DGX-1 and Wilkes-2 with fully connected clusters of four GPUs, benefit the most from these features; in contrast, SummitDev, which has only connected pairs of GPUs, benefits less in this case. Comparing plots in Figs. 3, 4, 5, 6 and 7 with and without GDR enabled indicates that the use of GDR primarily improves the performance of the distributed 3D FFTs. Additionally, it can be noted that on the DGX-1 and Wilkes-2, the FFT performance improves dramatically when the number of pools results in pool groups with four GPUs, where all communication within the all-to-all occurs over peer-to-peer connections.

Considering the eigensolver on the CPU, the scaling behavior aligns more closely with what is expected, with a small edge in efficiency when distributed across fewer processes. On the GPU systems, due to the use of a serial eigensolver, increasing the number of pools from one to two results in a halving of the eigensolve time. Since the serial eigensolver is always computed using a single GPU per pool group, the eigensolve time scales proportionally with the number of pools. This trend can be observed in Figs. 3, 4, 5, 6 and 7 on all the GPU systems. Consequently, for a given number of pools, the eigensolver performance will remain fixed regardless of the number of GPUs assigned to the pool group, leading to some loss in efficiency. Despite this, the serial GPU eigensolver outperforms the distributed ELPA library used on the reference CPU system for the AUSURF112 cases, while maintaining competitive performance in the Ta2O5 cases.

Comparing the reference CPU system to the GPU system results, the GPU systems are outperforming the reference CPU system in all tested configurations, when comparing single CPU socket performance to single GPU performance,

with relative speedups ranging from 2 to 4. Figures 3, 4, 5, 6 and 7 illustrate where to attribute these gains in performance. In all cases, a large portion of the improvement can be attributed to faster ZGEMM and DGEMM performance on the GPU systems. This is clear, since on the GPU systems, the GEMM dominated portion of the runtime outside of the FFT and eigensolve is significantly reduced on the GPU systems relative to the reference CPU system. Beyond this, additional performance improvements of varying degree can be attributed to the FFT and eigensolver.

Comparing GPU system results, there is some observed variability in the performance between the systems, which can be attributed to differences in node topology (how many GPUs are associated with each CPU socket and how are they connected) and node architecture (IBM POWER8 with host-to-device connections via NVLink compared to Intel Xeon with host-to-device connections via PCIe). As a first example, the slowest GPU system results occur on DGX-1 and Wilkes-2 when GDR features are disabled. These two systems have the highest ratio of GPUs to CPU sockets, with each system having four GPUs per CPU socket. In addition to this, the GPUs on these system share PCIe lanes, with two GPUs per PCIe root complex on the DGX-1, and four GPUs per PCIe root complex on Wilkes-2. Thus, with GDR disabled, the all-to-all communication during the distributed 3D FFTs become bottlenecked by a lack of PCIe bandwidth for transfer of communication buffers between the host and device and CPU memory bandwidth to handle all the MPI traffic. With GDR features enabled however, these bottlenecks are alleviated due to the substantial increase in device-to-device bandwidth offered via peer-to-peer connections, freeing up the CPU to handle only out of socket MPI traffic. This results in these systems showing the highest performance of all the systems tested when GDR features are enabled, demonstrating the importance of exploiting these peer-to-peer connections when possible.

On a related note, due to higher memory bandwidth offered by the POWER8 CPU and greater host-device bandwidth through NVLink, SummitDev is less impacted by these issues, leading to high distributed FFT performance even without GDR. The higher host-to-device bandwidth also gives SummitDev an improvement in distributed FFT performance over Piz Daint, due to faster transfer of communication buffers between host and device.

While SummitDev maintains an edge in the distributed FFT performance in non-GDR enabled cases, the eigensolver performance on this system lags behinds that of the other GPU systems. As a generic LAPACK implementation of the offloaded tridiagonal eigensolver was used for this system, the benefits of multithreading from ESSL was limited to the underlying BLAS calls, leading to a loss in performance relative to a fully multi-threaded implementation. Otherwise, the eigensolver performance is generally more consistent across the GPU systems using Intel CPUs with MKL, even with a varied number of cores available to the GPUs performing the eigensolve.

## 7 Conclusions

This paper presented development details and performance of PWscf on CPU and GPU systems. The new GPU version produces accurate results and can reduce the time-to-solution by an average factor of 2–3 relative to a reference CPU system.

The custom GPU eigensolver developed for this code is very competitive with both ScaLAPACK and ELPA, with little sensitivity to available host resources. Improvements to performance via distribution over multiple GPUs and removing existing CPU dependencies are being considered for future development.

The performance results in this study illustrate the importance of exploiting peer-to-peer connectivity between GPUs when available, implicitly via CUDA-aware MPI or explicitly using CUDA IPC or similar mechanisms. These features, when properly utilized, can provide a substantial performance boost, particularly on systems with high GPU to CPU socket ratios. The upcoming generation of NVIDIA GPUs, Volta, with a faster memory subsystem and double precision performance higher than 7 TeraFLOP/s, will help push the performance of this code even further.
The code is available for download at https://github.com/fspiga/qe-gpu.

**Acknowledgments.** This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This work was also supported by a grant from the Swiss National Supercomputing Centre (CSCS) under project ID g33. Wilkes-2 is part of the *Cambridge Service for Data Driven Discovery* (CSD3) system operated by the University of Cambridge Research Computing Service funded by EPSRC Tier-2 capital grant EP/P020259/1, the STFC DiRAC HPC Facility (BIS National E-infrastructure capital grant ST/K001590/1, STFC capital grants ST/H008861/1 and ST/H00887X/1, Operations grant ST/K00333X/1) and the University of Cambridge. CSD3 and DiRAC are part of the UK National e-Infrastructure. Paolo Giannozzi also acknowledges support from the European Union through the MaX Centre of Excellence (Grant No. 676598).

## References

1. Auckenthaler, T., Blum, V., Bungartz, H.J., Huckle, T., Johanni, R., Krämer, L., Lang, B., Lederer, H., Willems, P.R.: Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations. Parallel Comput. **37**(12), 783–794 (2011)
2. Blackford, L.S., Choi, J., Cleary, A., D'Azeuedo, E., Demmel, J., Dhillon, I., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK User's Guide. Society for Industrial and Applied Mathematics (1997)
3. Fatica, M.: Customize CUDA Fortran Profiling with NVTX (2015). https://devblogs.nvidia.com/parallelforall/customize-cuda-fortran-profiling-nvtx
4. Fatica, M., Ruetsch, G.: CUDA Fortran for Scientists and Engineers. Morgan Kaufmann, Burlington (2014)
5. Froyen, S.: Brillouin-zone integration by Fourier quadrature: special points for superlattice and supercell calculations. Phys. Rev. B **39**, 3168–3172 (1989)

6. Giannozzi, P., Baroni, S., Bonini, N., Calandra, M., Car, R., Cavazzoni, C., Ceresoli, D., Chiarotti, G.L., Cococcioni, M., Dabo, I., et al.: QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials. J. Phys. Condensed Matter **21**(39), 395502 (2009)
7. Dongarra, J., Gates, M., Haidar, A., Kurzak, J., Luszczek, P., Tomov, S., Yamazaki, I.: Accelerating numerical dense linear algebra calculations with GPUs. In: Kindratenko, V. (ed.) Numerical Computations with GPUs, pp. 3–28. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06548-9_1
8. Johnson, D.D.: Modified Broyden's method for accelerating convergence in self-consistent calculations. Phys. Rev. B **38**, 12807–12813 (1988)
9. Kohn, W.: Fundamentals of density functional theory. In: Joubert, D. (ed.) Density Functionals: Theory and Applications, pp. 1–7. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0106731
10. Kraus, J.: CUDA Pro Tip: generate custom application profile timelines with NVTX (2013). https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-generate-custom-application-profile-timelines-nvtx
11. Marek, A., Blum, V., Johanni, R., Havu, V., Lang, B., Auckenthaler, T., Heinecke, A., Bungartz, H.J., Lederer, H.: The ELPA library: scalable parallel eigenvalue solutions for electronic structure theory and computational science. J. Phys. Condensed Matter **26**(21), 213201 (2014)
12. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, Version 2.2. Technical report (2009). http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf
13. Parr, R.G., Yang, W.: Density-Functional Theory of Atoms and Molecules (International Series of Monographs on Chemistry). Oxford University Press, New York (1994)
14. Pickett, W.E.: Pseudopotential methods in condensed matter applications. Comput. Phys. Rep. **9**(3), 115–197 (1989)
15. Romero, J.: Developing an Improved Generalized Eigensolver with Limited CPU Offloading. In: GPU Technology Conference, San Jose, CA (2017). http://on-demand.gputechconf.com/gtc/2017/presentation/s7388-joshua-romero-developing-an-improved-generalized-eigensolver.pdf
16. Spiga, F.: Plug-in code to accelerate Quantum ESPRESSO v5 using NVIDIA GPU. https://github.com/fspiga/qe-gpu-plugin
17. Spiga, F.: Implementing and testing mixed parallel programming model into Quantum ESPRESSO. In: Science and Supercomputing in Europe - Research Highlights 2009, CINECA Consorzio Interuniversitario, Bologna, Italy (2010)
18. Spiga, F., Girotto, I.: phiGEMM: a CPU-GPU library for porting Quantum ESPRESSO on hybrid systems. In: 2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing, pp. 368–375 (2012)

# Performance Modeling and Simulation

# Modeling Large Compute Nodes with Heterogeneous Memories with Cache-Aware Roofline Model

Nicolas Denoyelle[1,3](✉), Brice Goglin[1](✉), Aleksandar Ilic[2](✉), Emmanuel Jeannot[1], and Leonel Sousa[2](✉)

[1] Inria – Bordeaux - Sud-Ouest, Univ. Bordeaux, Talence, France
{nicolas.denoyelle,brice.goglin,emmanuel.jeannot}@inria.fr
[2] INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal
{aleksandar.ilic,leonel.sousa}@inesc-id.pt
[3] Atos, Paris, France

**Abstract.** In order to fulfill modern applications needs, computing systems become more powerful, heterogeneous and complex. NUMA platforms and emerging high bandwidth memories offer new opportunities for performance improvements. However they also increase hardware and software complexity, thus making application performance analysis and optimization an even harder task. The Cache-Aware Roofline Model (CARM) is an insightful, yet simple model designed to address this issue. It provides feedback on potential applications bottlenecks and shows how far is the application performance from the achievable hardware upper-bounds. However, it does not encompass NUMA systems and next generation processors with heterogeneous memories. Yet, some application bottlenecks belong to those memory subsystems, and would benefit from the CARM insights. In this paper, we fill the missing requirements to scope recent large shared memory systems with the CARM. We provide the methodology to instantiate, and validate the model on a NUMA system as well as on the latest Xeon Phi processor equiped with configurable hybrid memory. Finally, we show the model ability to exhibits several bottlenecks of such systems, which were not supported by CARM.

## 1 Introduction

The increasing demands of current applications, both in terms of computation and amount of data, and the limited improvements of sequential performance of the cores led to the development of large multi-core and many-core systems [1]. These platforms embed complex memory hierarchies, spanning from registers, to private and shared caches, local main memory, and memory accessed remotely through the interconnection network. In these systems, memory throughput is not uniform anymore since the distance between processor and memory banks varies. On such Non-Uniform Memory Access (NUMA) architectures, the way data is allocated and accessed has a strong impact on performance [2]. Optimizing applications data locality for these machines requires a deep understanding

© Springer International Publishing AG 2018
S. Jarvis et al. (Eds.): PMBS 2017, LNCS 10724, pp. 91–113, 2018.
https://doi.org/10.1007/978-3-319-72971-8_5

of hardware bottlenecks as well as application needs. Hence, modeling of memory access performance is of high importance.

Recently, the latest Intel Xeon Phi processor, codename Knights Landing (KNL) [3], entered the NUMA landscape with a processor divisible into 4 *Sub-NUMA Clusters* (SNC-4 mode). Usually, NUMA platforms include several sockets interconnected with processor-specific links (e.g. Quick Path Interconnect [4]) or by custom technologies such as SGI NUMAlink or Bull Coherent Switch [5]. However, the KNL interconnects NUMA clusters at the chip scale (through a 2D mesh of up to 36 dual-core tiles). Though the software may see both types of system as similar homogeneous NUMA trees, the strong architectural differences between NUMA sockets and KNL chips, described above, can impact application performance in different ways and motivate the joint study of both systems.

Additionally, each cluster of the KNL may feature traditional DDR memory as well as 3D-stacked high-bandwidth memory named MCDRAM, that can be used as a hardware-managed cache or as an additional software-managed memory. Managing heterogeneous memories in runtime systems, applications or compilers brings another level of complexity and makes performance analysis harder and even more necessary. Hence, being able to understand the impact of the memory hierarchy and core layout on application performance as well as on attainable hardware upper-bounds is of high interest. This is especially true when modeling the architecture and tuning applications to this kind of hardware.

To optimize the application execution and to infer their ability to fully exploit the capabilities of those complex systems, it is necessary to model and acquire the knowledge about the realistically achievable performance upper-bounds of these systems and their components (including all levels of memory hierarchy and interconnection network). The Cache-Aware Roofline Model [6] (CARM) has been recently proposed (by some of the authors of this paper) as an insightful model and an associated methodology aimed at visually aiding the performance characterization and optimization of applications running on systems with cache memory subsystems. CARM has been integrated by Intel into their proprietary tools, and it is described as "*an incredibly useful diagnosis tool (that can guide the developers in the application optimization process), ensuring that they can squeeze the maximum performance out of their code with minimal time and effort.*"[1] However, the CARM only refers to systems based on a single-socket computational node with uniform memory access, without considering the NUMA effects that can also dramatically impact the performance.

To address these issues, we propose a new methodology to enhance the CARM insightfullness and provide locality hints for application optimization on contemporary large shared memory systems, such as multi-socket NUMA systems and Many Integrated Core processors equipped with heterogeneous memory technologies and with various hardware configurations. The proposed model is experimentally validated with high accuracy on both an Intel Knights Landing

[1] Intel Advisor Roofline - 2017-05-12: https://software.intel.com/en-us/articles/intel-advisor-roofline.

and a dual-socket Broadwell Xeon multi-core host by relying on a set of micro-benchmarks, a set of synthetic benchmarks, and finally proxy applications.

The remainder of this paper is organized as follows. Section 2 provides an in-depth overview of the Cache Aware Roofline Model and our contribution to make the model usable for NUMA and KNL architectures. Section 3 deep dives into the methodology to measure hardware upper-bounds for these systems. Sections 4 and 5 detail the model instantiation and validation for a Xeon E5-2650L v4 NUMA system composed of 4 NUMA nodes and the latest Xeon Phi many-core processor. Finally, Sect. 6 gives an overview of state-of-the-art related works.

## 2 Locality Aware Roofline Modeling

The generic Roofline modeling [7] is an insightful approach to represent the performance upper-bounds of a processor micro-architecture. Since computation and memory transfers can be simultaneously performed, this modeling is based on the assumption that the overall execution time can be limited either by the time to perform computations or by the time to transfer data. Hence, from the micro-architecture perspective, the overall performance can be limited either by the peak performance of computational units or by the capabilities of the memory system (*i.e.* bandwidth).

To model the performance limits of contemporary multi-core systems, the Cache-Aware Roofline Model (CARM) [6] explicitly considers both the throughput of computational unit and the realistically achievable bandwidth of each memory hierarchy level[2]. With this purpose, the CARM (see Fig. 1) includes several lines representing the system upper-bounds (*Roofs*). Oblique lines (representing the memory bandwidths) cross the horizontal lines (representing the

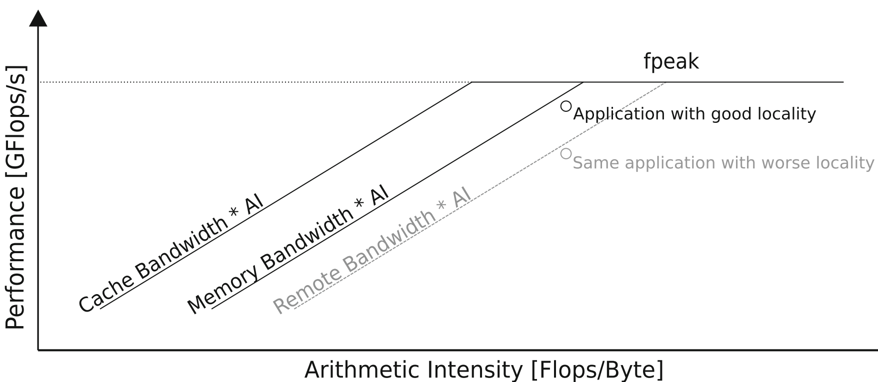



**Fig. 1.** CARM chart of an hypothetical compute node composed of one cache level and NUMA memories.

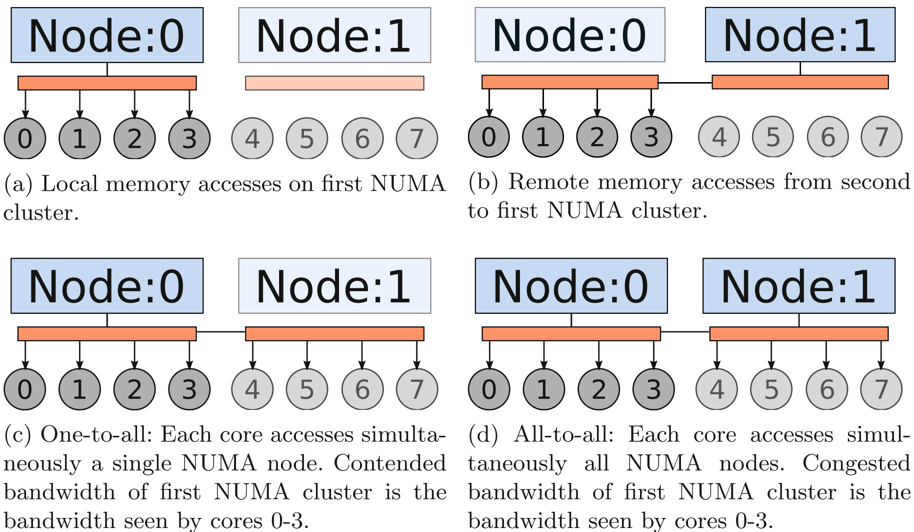
[2] Main memory and cache levels.

peak compute performance), bounding hereby the area of application characterization by respectively memory and compute regions. The CARM introduces a detailed and meticulous methodology for platform benchmarking from which this paper inherit and extends the content to NUMA platforms.

In contrast to the other roofline approaches [7], the CARM perceives the computations and memory transfers from a consistent micro-architecture point of view, *i.e.* cores where the instructions are issued. Hence, when characterizing the applications, the CARM relies on the performance (in GFlop/s) and the true *Arithmetic Intensity* (AI), *i.e.* the ratio of performed compute operations (flops) over the total volume of requested data (in bytes). The CARM is presented in the log-log scale, where the x-axis refers to the AI (in flops/byte) and the y-axis to the performance (in GFlop/s).

**Our Contribution: Extending the CARM to NUMA and KNL**

From the application perspective, the memory of modern computing systems is abstracted as a flat address space. However, the memory architecture of contemporary large compute nodes is made of remote and/or heterogeneous memories. In order to fully exploit those system capabilities, current software interfaces [8–10] require an explicit data allocation policy and/or thread binding policy to reach good performance [11,12]. Figure 2 depicts such a system, including two sockets with their local memory (also named NUMA node) and a set of cores. On these systems, the bandwidth is not uniform across the network, and it influences memory access performance. Hence, when modeling, the source and destination of memory access (*i.e.* from a core to a NUMA node, e.g. local access: Fig. 2a




(a) Local memory accesses on first NUMA cluster.

(b) Remote memory accesses from second to first NUMA cluster.

(c) One-to-all: Each core accesses simultaneously a single NUMA node. Contended bandwidth of first NUMA cluster is the bandwidth seen by cores 0-3.

(d) All-to-all: Each core accesses simultaneously all NUMA nodes. Congested bandwidth of first NUMA cluster is the bandwidth seen by cores 0-3.

**Fig. 2.** Modeled memory access patterns.

or remote access: Fig. 2b) should be taken into account to understand application performance. Moreover, such large scale systems contain a high amount of cores whose pressure on NUMA nodes can cause data accesses to be serialized when all of them are accessing a single memory. We qualify this situation as *Contention* and depict it in Fig. 2c. Finally, the network connecting the NUMA nodes to the cores can be subject to *Congestion*, when several data paths from the memory to the cores, cross the same link. In Fig. 2d, we consider the case where data is balanced on memories, i.e. there is no contention, however, each core will access data located over the whole system NUMA nodes and will eventually create Congestion because of the interleaved data paths. In the remainder of the paper, we use the term Cluster to refer to a set of neighbor cores and their local NUMA node(s)[3].

CARM metrics are consistent across the whole memory hierarchy of a single cluster (as illustrated for the first cluster in Fig. 2a). However, from the core perspective, memory access performance is not consistent across the system: bytes transferred from one NUMA node to a cluster are not transferred at the same speed to other clusters. This implies that the legacy CARM can only handle a single multi-core cluster, and fails to characterize accurately the cases in Figs. 2b, c, and d. Yet, as Fig. 1 shows, without proper (here remote) bandwidth representation in the CARM, locality issues are not obvious since the performance loss can come from many different sources: no vectorization, sparse memory access, *etc.*

Thus, we propose to extend the CARM with the Locality Aware Roofline Model (LARM), providing the lacking NUMA insights, represented in Fig. 2 and characterizing the three main throughput bottlenecks, characteristic of this type of hardware: non uniform network bandwidth (Fig. 2b), node contention (Fig. 2c), and network congestion (Fig. 2d). For this purpose, the LARM iterates the CARM over all the clusters of a computing system and keeps local consistency while minimizing the changes over the legacy model and taking into account the non-uniform aspect of the system. It follows that the LARM chart is a set of CARM charts, *i.e.* one chart per cluster, characterizing hereby the system performance upper-bounds under all perspectives. In each subsequent chart, the LARM includes three new groups of roof characterizing above mentioned bottlenecks. The remote roofs set the reference upper-bound of achievable bandwidth from remote nodes to a cluster. The congestion roof set the bandwidth achieved by a cluster when all the system cores are accessing simultaneously memory regions located across every NUMA nodes in a round-robin fashion. Finally the contention roofs characterize a cluster granted bandwidth when the whole system cores are accessing simultaneously a single NUMA node. Unlike usual CARM roofs, the new roofs stand as lower-bound roofs because they represent a reference below top expectations, *i.e.* local memory access roofs. However, as for the

[3] On usual platforms, a cluster is identical to the widely-used definition of a NUMA node. On KNL, there can exist two local NUMA nodes near each core (DDR and MCDRAM), hence two NUMA nodes per cluster.

CARM, the closer an application is to a roof, the more likely this application is to be bound by this hardware bottleneck.

To the best of our knowledge, there is no work using the CARM to characterize NUMA platforms. Hence, beside the contribution of extending the CARM, we present the following work in the remainder of this paper: (1) we implemented a tool based on CARM methodology and the proposed improvements to automatically instantiate and validate the model on multi-socket systems and Knights Landing (KNL) Xeon Phi (for various memory configurations); (2) we thus validate the new model with high accuracy micro-benchmarks, for both systems; (3) We also demonstrate the model usability with synthetic benchmarks from the BLAS package; and (4) we exhibit the model ability to pinpoint data locality issues on MG from the NAS parallel benchmarks [13] and Lulesh proxy-application [14], where several data allocation policies are applied.

# 3 Methodology for Memory and Micro-architecture Throughputs Evaluation

Initially, the Cache-Aware Roofline Model is built with two sets of parameters: micro-architecture instruction throughput and the attainable memory bandwidth. The former provides the peak floating point performance and L1 bandwidth while the latter is used to construct a set of local memory roofs (i.e. L2, L3, local DRAM bandwidths). The Locality-Aware Roofline model, adds the perspective dimension and rooflines for several memory access patterns which require the ability to detect and model the system topology. For this purpose, we leverage hwloc [9] hierarchical representation of the machine to automatically enrich the CARM with the herein proposed memory roofs.

The micro-architecture throughput can be obtained whether by relying on the theoretical hardware properties, or by extensively benchmarking the micro architecture. In the former case, the peak floating point performance can be computed as:

$$\underbrace{Fpeak}_{GFlop/s} = \underbrace{Throughput}_{Instructions/Cycle} * \frac{Flops}{Instruction} * N * \underbrace{Frequency}_{GHz}, \tag{1}$$

where the *Throughput* is the number floating point instruction retired per cycle by one core, *Flops/instruction* is the number of floating point operations performed in each instruction (*e.g.* 2 for `FMA` instruction and 1 for `ADD` instruction), and $N$ is the number of cores considered. Similarly, the peak bandwidth of the Level 1 cache can be computed as:

$$\underbrace{Bandwidth}_{GByte/s} = \underbrace{Throughput}_{Instructions/Cycle} * \frac{Bytes}{Instruction} * N * \underbrace{Frequency}_{GHz}. \tag{2}$$

Sometimes, theoretical throughput, provided by the constructor, and experimental throughput measured from highly tuned software do not match, or even the

former is not publicly available. For this reason, we use the prior CARM methodology to implement highly optimized micro-benchmarks and build the proposed roofs. Our methodology for NUMA-specific bandwidth evaluation relies on a hierarchical description of the system topology as provided by the hwloc library, to characterize the system bandwidth in a pertinent way. We focus on deep and heterogeneous memory level evaluation, rather than on micro-architecture throughput evaluation and caches already studied in [6]. Since the model needs to provide insights on possible bottlenecks of NUMA systems, the model includes the bandwidth roofs described in Sect. 2 and Fig. 2, *i.e.* local accesses, remote accesses, accesses with congestion and accesses with contention. In order to characterize local and remote bandwidths of a cluster in the model, a benchmark performs contiguous memory access, as in the CARM, but on each NUMA node individually. One thread per core of the target cluster is spawned, then for each roof (i.e. local and remotes), the workload is iteratively allocated on each NUMA node, as depicted in Figs. 2a and b. We do not look at individual links, but rather at pairs of cores+NUMA node, even though sometimes there are multiple (unknown) hops between clusters. The contended bandwidths are obtained similarly to the local and remote bandwidths, but loading the whole system cores with threads (Fig. 2c). Each cluster granted bandwidth is associated with the source contended node to build the contended roofs on each cluster chart. Finally, the congested bandwidth is obtained by doing memory access from all the cores, contiguous on the virtual address space, but with pages physically allocated in a round-robin fashion across the system NUMA nodes, and with a private data set for each thread. Once again, the bandwidth perceived by each cluster is modeled as the congested roof in its local CARM. Though we call it congestion, it differs from the official definition[4]. However it fits a more practical and easy to reproduce memory access pattern, i.e. the one implied by using the linux interleave memory allocation policy.

In this paper, we only show the bandwidth of `LOAD` instructions because it suits better our use cases, however we are able to also measure `STORE`, non-temporal `STORE` and mix of those for all memory levels with our tool.

## 4 Model Instantiation and Validation on Multi-socket System

In order to set up and validate the model, we use a dual-socket NUMA system named *Joe0*. It is composed with two Broadwell Xeon E5-2650L v4 processors (at 1.7 GHz), configured with the cluster-on-die mode and exposing the 4 NUMA nodes to the system. Each NUMA node of the system topology (Fig. 3) implement 7 cores, here with hyperthreading disabled, and is pictured on Fig. 3.

[4] Network congestion in data networking and queueing theory is the reduced quality of service that occurs when a network node is carrying more data than it can handle.

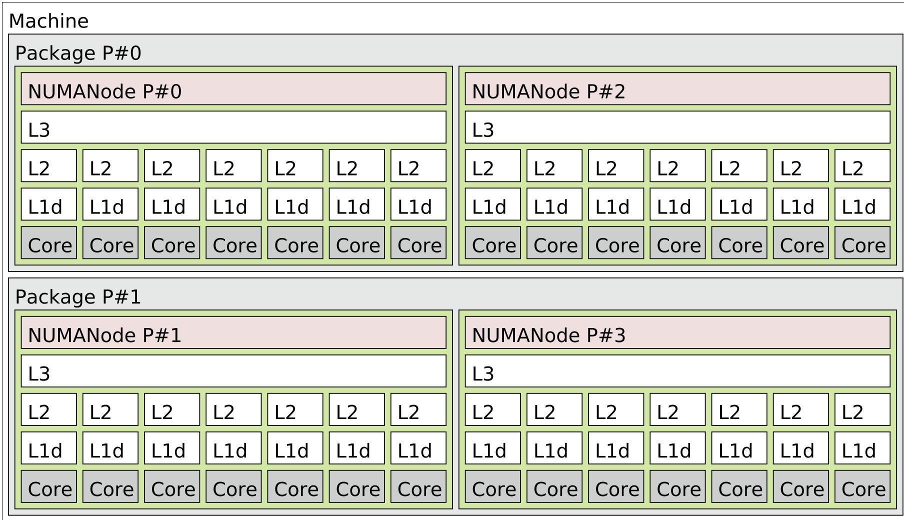



**Fig. 3.** hwloc topology representation of a dual-socket Xeon E5-2650L v4.

### 4.1 Platform Evaluation and Model Instantiation

By relying on the testing methodology proposed in [6], it was possible to reach near theoretical compute and L1 cache throughputs on the Intel Broadwell micro-architecture, as presented in Table 1. Each core throughput is derived using the number of operations per instruction and the processor frequency to obtain the peak FMA floating point performance (reaching 190 GFlop/s for a single cluster).

**Table 1.** Joe0 core instructions throughput (Instructions/Cycle)

| Instruction throughput | Load | Store | ADD | MUL | FMA |
|---|---|---|---|---|---|
| Theoretical | 2 | 1 | 1 | 2 | 2 |
| Experimental | 1.99 | 0.99 | 0.99 | 1.99 | 1.99 |

As presented in Sect. 3, the next evaluation aims to extensively benchmark the memory subsystem with several memory access patterns, *i.e.* the remote/local bandwidth between each pair (cluster, NUMA node), as well as the contended/congested ones. The results obtained are presented in Table 2 for the first NUMA cluster of the system. Unless specified, the model presented herein is restricted to a single NUMA cluster, due to the bandwidth symmetry between clusters.

The obtained measures (Tables 1 and 2) are then used to build the proposed model depicted in Fig. 4 for the first cluster of Joe0. Besides the roofs for local

**Table 2.** Joe0 bandwidth roofs to the first NUMA cluster, i.e. NUMANODE:0.

| Memory level | Bandwidth (GByte/s) |
|---|---|
| L1 | 760.1 |
| L2 | 309.2 |
| L3 | 154.0 |
| NUMANODE:0 (local) | 36.1 |
| NUMANODE:1 (remote) | 17.5 |
| NUMANODE:2 (remote) | 15.0 |
| NUMANODE:3 (remote) | 14.3 |
| NUMANODE:0 (contended) | 16.7 |
| NUMANODE:1 (contended) | 8.3 |
| NUMANODE:2 (contended) | 6.8 |
| NUMANODE:3 (contended) | 6.2 |
| All NUMANODES (congested) | 18.1 |

caches, this CARM chart also includes all the proposed memory roofs, namely local, remote, contended and congested roofs.

### 4.2 Model Validation

**With Micro-benchmarks.** This validation step consists in micro-benchmarking the system with several arithmetic intensities, *i.e.* interleaving the memory and compute instructions used in the above platform evaluation. It assesses code ability to reach measured roofs while performing both computations and memory accesses. We measure the roofs fitness as the relative root mean squared error[5] of validation points to the roof performance for a realistic range of arithmetic intensities. The errors and deviation (too small to be visible) for each validation point, and for each bandwidth roof of a single cluster of the system are presented in Fig. 4. As the error computed in the legend is small (less than 2% in average for every roof), the validation enforces that measured bandwidths are attainable by programs of various arithmetic intensities.

**With Synthetic Benchmarks.** Figure 5 shows the LARM instantiated on the first socket of Joe0. For each NUMA cluster a CARM chart includes local cache bandwidths, local node bandwidth, the bandwidth under congestion[6] and

[5] The error is computed as $\frac{100}{n} \times \sqrt{\sum_{i=1..n} \left(\frac{y_i - \hat{y}_i}{\hat{y}_i}\right)^2}$ where $y_i$ is the validation point at a given arithmetic intensity, and $\hat{y}_i$ is the corresponding roof.

[6] Remote memory bandwidths are very close to congested bandwidths on this system and we omit the former in the chart to avoid confusion.

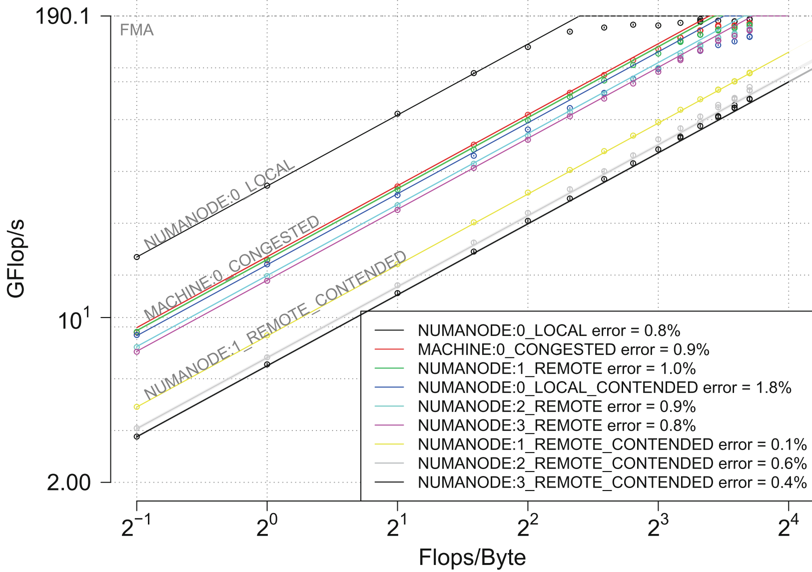
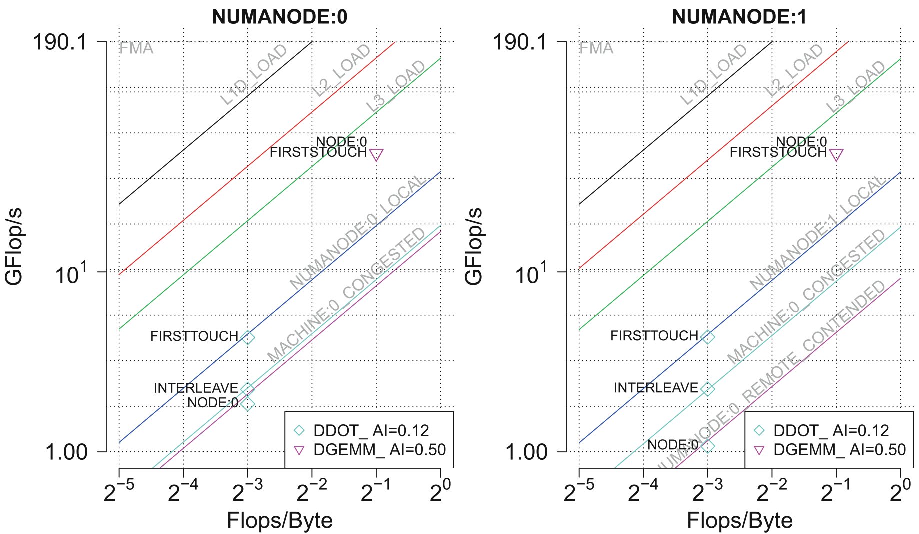



**Fig. 4.** CARM validation of one NUMA cluster of Joe0 platform. Validation points are visible along the roofs. Finally the model error for each roof is in the legend.




(a) CARM of the first NUMA cluster. (b) CARM of the second NUMA cluster.

**Fig. 5.** LARM chart of linear algebra kernels on one socket of Joe0 system.

the bandwidth of the first NUMA node under contention (which is different whether we see it from the first or the second cluster). Figure 5 also illustrates the memory-bound ddot kernel and the compute-bound dgemm kernel from the BLAS package, under several scenarios, showing the model ability to pinpoint locality issues. For each scenario, threads are bound in round-robin fashion and data allocation policy is one of: firsttouch (*i.e.* data in memory close to threads), interleave (*i.e.* data spread on all nodes), Node:0 (*i.e.* data on a single memory node). Each thread performs the same amount of work though the allocation policy on a single node may create an asymmetry when observing their performance across different NUMA nodes (Fig. 5). The modeled applications were run on the full system, *i.e.* 28 threads (1 thread per core), however, only the model for a single socket is presented to avoid redundancy. On the chart (Fig. 5), ddot and dgemm are represented each with their own constant arithmetic intensity (i.e. the code is unchanged between scenarios) but with several performances (changed runtime parameters).

The ddot case with allocation on Node:0 has a different performance whether we look at the first or the second cluster. Hence, the kernel characterization shows the model ability to spot asymmetries. Even if asymmetries do not originate from the instructions, they can come from the data distribution and significantly impact the performance [15]. Congested and contended roofs also successfully characterize similar bottlenecks in ddot application. Indeed, in Fig. 5, ddot kernel with interleaved access (*i.e.* inducing congestion) and access on a single node (*i.e.* inducing contention) match with appropriate memory bandwidths. Optimized compute-intensive applications do not suffer from locality issues. Indeed, as presented in Fig. 5, the dgemm execution is not affected by non-uniform memory access, since it achieves the same performance on each node even if data is allocated with different policies. This can be attributed the high cache efficiency of the kernel allowing the system to prefetch the required data into the cache before it is actually required, thus avoiding the local and remote memory access bottlenecks.

In a nutshell, data allocation policies applied to synthetic benchmarks affects the performance in a way that is foreseeable. It matches expectations from their characterization in the LARM, thus validating the proposed roofs relevance.

**With NAS MG Parallel Benchmark.** This step aims to show that the model insights can help to flush out performance bottlenecks, i.e. application characterization relatively to the new roofs can help to pinpoint potential execution bottlenecks. For this purpose, we ran a C version[7] of the NAS-3.0 MG benchmark with one thread per core, bound in a round-robin fashion on the system cores. On this system, we extract the LARM metrics with hardware counters at the core level, and aggregate the results at the Cluster level. As presented in Fig. 6, three functions from MG benchmark are characterized on the first cluster of the system with several memory allocation strategies. In the first scenario, the default linux policy *firsttouch* is used for data allocation. The characterization of

[7] https://github.com/benchmark-subsetting/NPB3.0-omp-C.

these functions (labeled with *firsttouch*) reach near contention roof performance, and suggest to use the *interleave* allocation policy to balance memory accesses over the NUMA nodes in order to decrease the contention. Indeed, the latter policy increases dramatically the performance above the congestion roof. However, it is unlikely that the *interleave* policy surpasses the *firsttouch* policy with such a significance. Hence this observation also suggests that *firsttouch* actually allocates memory on a single node. Indeed, once parallelized, the previously sequential memory allocations, enable the firsttouch policy to allocate data on all NUMA nodes near appropriate threads, and improve again the performance (labeled with *enhanced_firsttouch*) compared to the *interleave* policy.

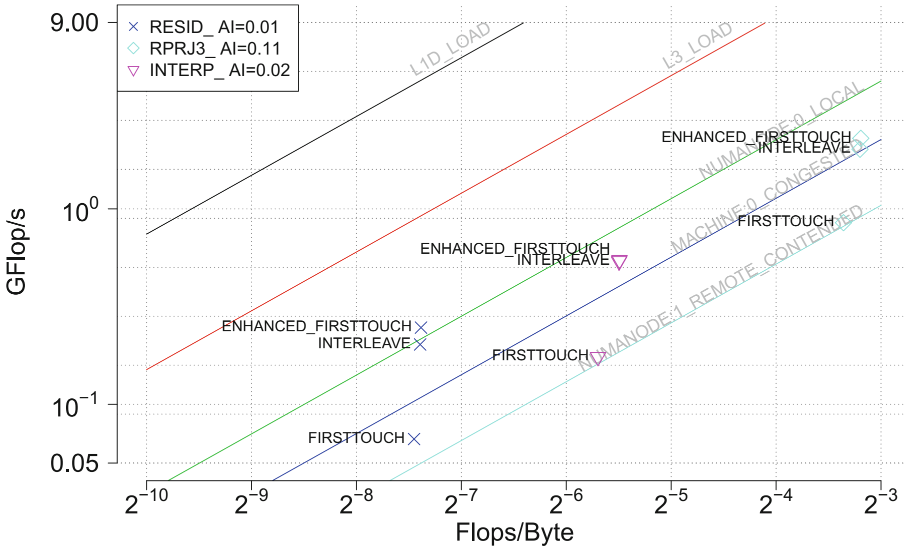



**Fig. 6.** NAS-3.0 MG functions characterization on Joe0 first cluster.

To sum up, the LARM characterization of the memory-bound MG benchmark, matches the contended roofs when data-allocation is serialized, i.e. data is allocated on a single contented node, and improves above the congestion roof once the contention issue is solved, validating hereby the proposed roofs.

## 5 Model Instantiation and Validation on Knights Landing Processor

When in SNC-4 mode [3], the KNL is a special case of a multi-Socket system where each socket has an additional fast memory (MCDRAM) to the conventional memory (DRAM also specified as NUMA:i), which is addressable in *Flat* mode or configurable as a last level cache in *Cache* mode. Whether the flat mode

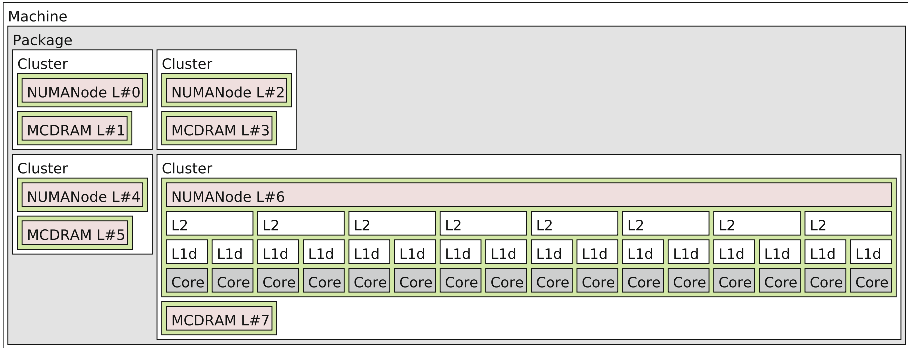



**Fig. 7.** hwloc model of KNL topology in SNC-4 flat mode. Only the fourth cluster is detailed, for clarity. Other clusters have a similar topology.

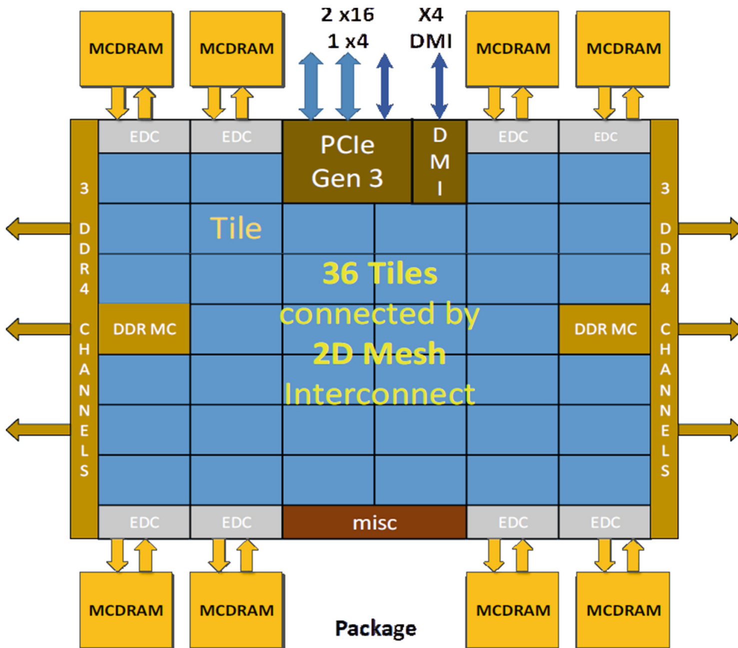



**Fig. 8.** Architecture of the Knights Landing mesh interconnect with DRAM and MCDRAM memory controllers (Source: Intel). Only 32 of these 38 tiles are actually enabled in our experimentation platform. The amount of tiles enabled can reach up to 36 tiles, though 38 are present.

or the cache mode is used, the system may yield different bandwidths, performance and execution time, and thus, the proposed model changes accordingly.

For our experiments, we used Knights Landing 7230 chips, with 64 cores at 1.3 GHz. The topology of the KNL with SNC-4 flat configuration is shown in Fig. 7 where the complete topology is provided for the last cluster. The mesh interconnection network (Fig. 8) between L2 tiles of the chip is widely different from conventional multi-socket system [16] and motivates additional observations compared to the previous system.

### 5.1 Platform Evaluation and Model Instantiation

By relying on the micro-architecture evaluation methodology from Sect. 3, the highest achievable throughput with carefully design micro-benchmarks is slightly lower than the theoretical values (see Table 3). However a performance of 2.2 TFlop/s for 64 cores is still achieved.

**Table 3.** Theoretical and experimental instruction throughput (in instructions per cycle) for a single core of the KNL platform.

| Instruction | Load | Store | ADD | MUL | FMA |
|---|---|---|---|---|---|
| Theoretical throughput | 2 | 1 | 2 | 2 | 2 |
| Experimental throughput | 1.66 | 0.96 | 1.70 | 1.70 | 1.70 |

Table 4 presents the bandwidth evaluation between clusters solo (*i.e.* by fully exercising memory units with a single cluster) for the flat mode. Only the evaluation for the first two clusters is presented since the others yield a similar bandwidth. Contrary to the multi-socket system, remote and local DRAM attain similar bandwidths which suggests high efficiency of the KNL interconnection network. However, significant and less predictable variations can be noticed for MCDRAM, which would require the disclosure of more architectural details to fully explain the mesh behavior.

**Table 4.** KNL load bandwidth (GByte/s) from first and second clusters memories to cores in flat mode. Other clusters are omitted because of similar results.

| | | From | | | | |
|---|---|---|---|---|---|---|
| | | NUMA:0 | MCDRAM:1 | NUMA:2 | MCDRAM:3 | |
| To | Cluster:0 | $38.1 \pm 0.1$ | $92.0 \pm 0.5$ | $38.0 \pm 0.8$ | $86.6 \pm 0.4$ | |
| | Cluster:1 | $38.1 \pm 0.1$ | $91.5 \pm 0.4$ | $38.2 \pm 0.1$ | $92.8 \pm 0.4$ | ... |
| | Cluster:2 | $37.8 \pm 0.1$ | $90.6 \pm 0.5$ | $38.1 \pm 0.1$ | $83.7 \pm 0.6$ | |
| | Cluster:3 | $38.0 \pm 0.2$ | $82.8 \pm 0.4$ | $38.0 \pm 0.1$ | $90.8 \pm 0.3$ | |

In Table 5, we also compare the load bandwidth granted to the first cluster when the data set is allocated into the first cluster DRAM and MCDRAM under several scenarios. The very first line is the reference when the cluster runs solo as in Table 4 but comparing the cache and flat modes. In the cache mode, the bandwidth of both types of memories (*i.e.* DRAM and MCDRAM) decreases, probably due to the overheads induced by the MCDRAM caching mechanism. In both modes, the DRAM bandwidth (NUMA:0) reduces when using all clusters simultaneously (*i.e.* local with 64 versus 16 threads), whereas this is less obvious for the MCDRAM. The presence of only two DRAM memory controllers shared among 4 clusters to access DRAM memory, whereas there are 8 EDC controllers (two per cluster) to access the MCDRAM (see Fig. 8), is a possible cause of this behavior.

**Table 5.** KNL load bandwidth (GByte/s) from first cluster memories.

| | | Flat | | Cache | | Threads |
|---|---|---|---|---|---|---|
| | | NUMA:0 | MCDRAM:1 | NUMA:0 | MCDRAM:1 | |
| Cluster:0 | Local | $38.1 \pm 0.1$ | $92.0 \pm 0.5$ | $22.9 \pm 0.7$ | $85.4 \pm 3.0$ | 16 |
| | Local | $21.7 \pm 0.7$ | $90.9 \pm 1.2$ | $20.0 \pm 0.7$ | $83.3 \pm 2.0$ | 64 |
| | Congested | $19.8 \pm 0.3$ | $77.6 \pm 2.0$ | $17.0 \pm 0.4$ | NA | 64 |
| | Contended | $10.7 \pm 0.0$ | $21.5 \pm 0.5$ | NA | NA | 64 |

In the cache mode, the bandwidth drop of DRAM memory when using all clusters simultaneously is not as high as the drop in the flat mode, probably because of data reuse in MCDRAM cache, which redirects a part of the traffic via the EDC channels and absorbs a part of the contention on DRAM memory controllers. Congestion already happens for interleaved memory access on DRAMs, provoking a further bandwidth reduction when compared to local memory accesses. As expected, contention is the worse case scenario, resulting in a dramatic bandwidth reduction for the cluster. Since congestion and contention are observable, they imply a need for locality to get good performance. Several Non-Achievable values stand in Table 4. One of them, i.e. contention on NUMA:0 in cache mode cannot be observed with the technique used in Sect. 3 methodology. Indeed, the private data accessed by each cluster in NUMA:0 memory would actually fit into the 4 MCDRAMs and result in MCDRAMs benchmark instead of NUMA:0 benchmark.

Based on the above characterization, the LARM is constructed for a single cluster and presented in Fig. 9, where the chip is configured in (SNC-4), flat mode. In contrast to the previous platform, it also includes the MCDRAM roofs siblings of the DRAM roofs. Bandwidths of remote nodes are hidden for clarity because they have the same order of magnitude as the local bandwidth and thus overlap on the chart.

## 5.2 Model Validation

**With Micro-benchmarks.** We use again the previous methodology to validate the model in Fig. 9, for one cluster (equivalent on the others). The micro-benchmark validation on KNL fits the model with an error below 5% in average for each roofs. Most of it is due to the points located near the ridge on L1 and MCDRAM bandwidths roofs. Otherwise it still fits nearly perfectly the roofs in the memory-bound and in the compute-bound regions.

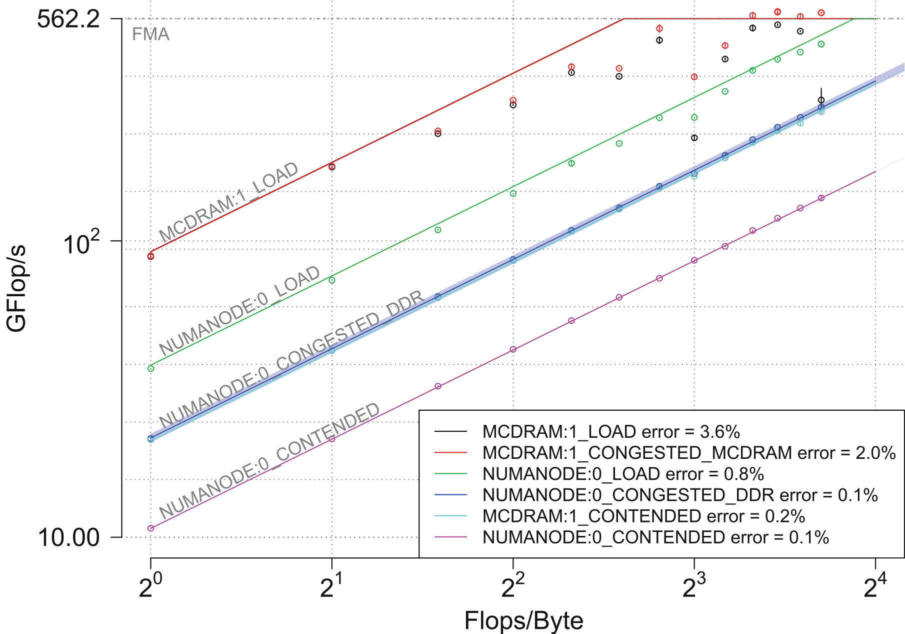
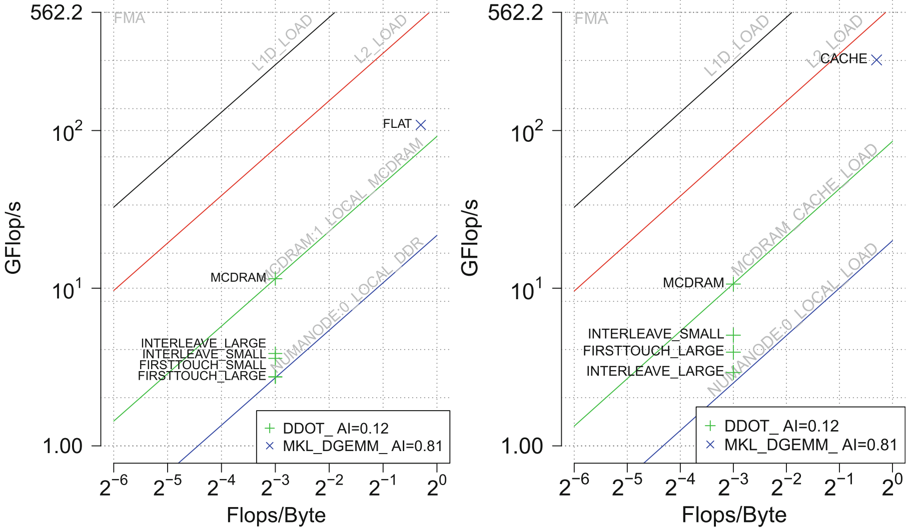



**Fig. 9.** CARM validation of one sub-NUMA cluster of KNL platform in SNC-4 flat mode.

**With Synthetic Benchmarks.** As previously referred, the validation with synthetic benchmarks aims to verify that well chosen causes lead to expected consequences, i.e. well chosen synthetic benchmarks are able to hit the roofs. For this purpose, we characterize again ddot and dgemm BLAS kernels in the CARM chart (Fig. 10) of the first cluster of the KNL. Here we focus on MCDRAM usage rather than the classical memory allocation policy already studied for the multi-socket system. Hence, we compare several data allocation strategies and sizes, as well as the flat and cache configurations of the chip.

The ddot function is compared under both flat and cache modes and by adopting various allocation strategies and data-set sizes. The small data-set (labeled *small* in Fig. 10 fits into the MCDRAM, whereas the large data-set




(a) CARM of the first cluster in flat mode. (b) CARM of the first cluster in cache mode.

**Fig. 10.** CARM of KNL first cluster with synthetic benchmarks running whether in flat mode or in cache mode.

(labeled *large*) does not. In the MCDRAM policy, we allocate the whole *small* data-set into MCDRAM. In the interleave policy, we use the Linux policy allocating pages of the data-set across all the nodes, *i.e.* MCDRAM and DRAM nodes. Finally, the Linux policy firsttouch allocates data on the DRAM near the first thread writing the corresponding page.

In flat mode, allocation into MCDRAM allows for the performance to reach near MCDRAM roof, and is visually assessed by the model. Unlike allocations into MCDRAM, large data-set with firsttouch policy are allocated into slow memory and also reach near DRAM roof performance. With the interleave policy, the data-set is mixed across memories and thus the performance stands in between the local DRAM roof and the local MCDRAM roof, with higher influence of the slower DRAM memory (the point is closer to this roof). In cache mode, the small data-set is cached in the MCDRAM, thus reaches a performance near MCDRAM bandwidth roof. Although the MCDRAM bandwidth is lower in cache mode, the performance of large data-set with firsttouch policy is higher than the DRAM roof. This is because of the large MCDRAM cache size that allows reusing a significant part of the data, thus improving the achievable performance. Interleaving memory accesses decreases the performance when compared to firsttouch policy, because of the congestion hereby created.

The dgemm function uses a data set too large to fit into MCDRAM and is compared in flat or cache mode. As expected, the intrinsic temporal locality of this kernel allows the cache mode to yield a better performance. When choosing the dataset size or location (DRAM or MCDRAM) the synthetic benchmark

performance still correspond to our expectation and validate the model insights on KNL system. When choosing the system configuration (flat or cache), the model also shows that kernels with good data reuse, i.e. with a performance over MCDRAM roofs, benefit from the hardware cache mechanism and also validates the model relevance.

**With Lulesh Proxy-Applications.** For this NUMA system, we focus on the Lulesh application because of it sensitivity to memory bandwidth. The aim of this validation step, is to show whether the model helps managing memory, i.e. whether performance can be improved with the chip configuration or a good memory allocation policy. From Lulesh, we pick the three greatest memory-bound hot spots of the application (*i.e.* the ones bounded below NUMANode:0 roof), namely CalcFBHourGlassForElems, IntegrateStressForElems functions, and a loop in the main function. Due to the lack of required hardware counters, arithmetic intensity, performance and application profile are collected with the Intel advisor tool[8]. In our experiments, we ran the application using a working set size large enough[9] not to fit into the MCDRAM. Hence, target memory needs to be carefully chosen to fulfill size constraints and a special care needs to be addressed when managing memory allocation to get good performance.

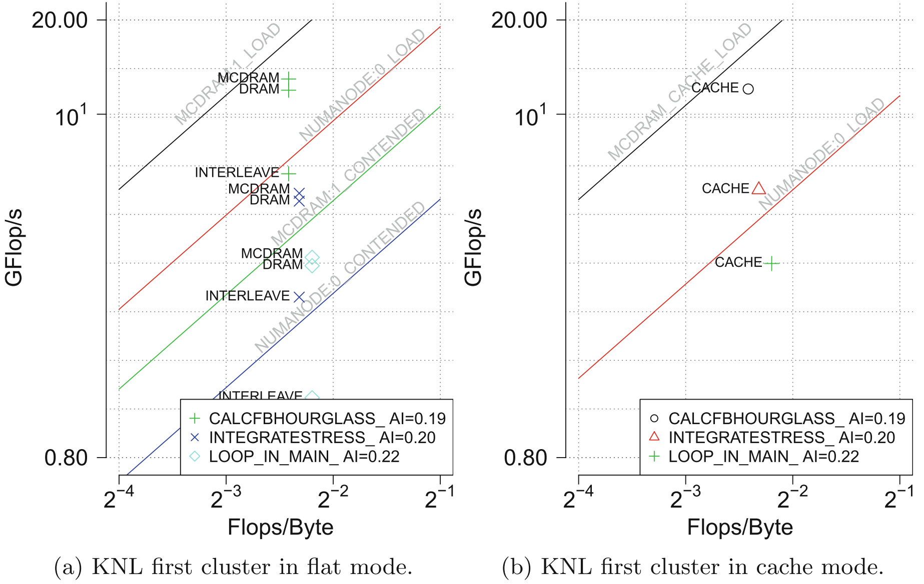
The first run allocates all data into regular DRAM memory (labeled DRAM), and aims at characterizing the application to find the potential allocation improvements. We then customize dynamic allocations in those hot spots by replacing the usual allocator from the standard C library with memkind [17] allocator to target the fast memory (labeled as *MCDRAM* in Fig. 11a) instead of the traditional DRAM (labeled as *DRAM*). Finally, instead of forcing MCDRAM allocations, we let the interleave policy (labeled as *interleave*) to choose data to put into MCDRAM for all allocations visible in the file lulesh.cc. Summarized, each hereby found hot spot is executed using three different policies, i.e. DRAM allocation (labeled DRAM), custom allocations (labeled MCDRAM), and interleave policy, in flat mode (see Fig. 11a).

The second chart in cache mode (see Fig. 11b), contrasts the performance of hand-tuned allocations into the MCDRAM with the hardware management of the fast MCDRAM cache. As expected MCDRAM allocations provide higher performance in flat mode. However, in cache mode the hot spots characterization reaches comparable performance to the top achieved performances in flat mode, denoting the hardware efficiency to manage data locality. In comparison, the interleave memory policy performs poorly maybe because of the spread allocations forcing threads to access remote nodes, and congesting the mesh.

To summarize the use of the LARM for data allocation policy choice with lulesh on the KNL, the cache mode brings no significant performance improvement, and the characterization laying below the MCDRAM roofs gave us this

[8] Product version: Update 2 (build 501009).

[9] Lulesh application run parameters: -i 1000 -s 60 -r 4. The application is compiled with ICC 17.0.2 and options: -DUSE_MPI=0 -qopenmp -O3 -xHost.




(a) KNL first cluster in flat mode. (b) KNL first cluster in cache mode.

**Fig. 11.** CARM chart of the KNL first cluster. The 3 main hot spots as detected by Intel Advisor are represented both in cache and flat modes.

insight. The data allocation policy however changes the performance dramatically between the interleave policy and other policies, but the cause is not clear in the model since the *interleave* points are widely spread on the performance axis which suggest another issue than ones the expected.

## 6 Related Works

To this date, there are two main approaches for Roofline modeling, namely: the Original Roofline Model (ORM) [7] and the Cache-Aware Roofline Model (CARM) [6]. Unlike the CARM that includes the complete memory hierarchy in a single plot, the ORM mainly considers the memory transfers between the last level cache and the DRAM, thus it provides fundamentally different perspective and insights when characterizing and optimizing applications [18]. Recently, the ORM was also instantiated on the KNL [19], without modifying the original model. The arithmetic intensity (AI) described in ORM is not to be confused with CARM AI because of the difference in the way how the memory traffic is observed. The bandwidth measured also differs from the one measured in this paper, the latter being explicitly load bandwidth. In [19], the authors present several ORM-based optimization case studies, and compare the performance improvements between Haswell processor and KNL, with data in DDR4 memory or MCDRAM, and finally KNL with data in MCDRAM memory. However, the authors do not show how the model can help choosing between memories when

working sets do not fit in the fastest one nor they provide a comparison with the cache mode.

An extension to the ORM, named 3DyRM [20], has been proposed to provide locality insights on NUMA systems. This model considers memory accesses from a single last level cache to any other memory, and not only local memory. It extends the ORM with a latency dimension to characterize the sampled memory access. Not only 3DyRM inherits the distorted perspective of the ORM, when characterizing real-world applications, but also it gives very limited insights on the distance of memory accesses to the NUMA thresholds considered in this paper. Moreover, 3DyRM characterizes applications with sampled memory accesses, without classifying them nor providing a methodology to get the first order insights, which is the main goal of the legacy model.

*Capability Model* [16] is recently proposed to evaluate KNL realistic upper-bounds and guide applications performance optimizations. The authors established a complex model mostly focusing on latency and bandwidth of the mesh interconnect. The Capability Model focuses on communication intensive algorithms (such as barrier synchronization, reduction, sorting, *etc.*), whereas the LARM has a throughput oriented approach, focusing on computational workloads stressing both compute and memory units. As such, the Capability models suits better message passing programming paradigms to enhance communication based algorithms, while the LARM suits better shared memory programming paradigms where communications are not explicitly expressed and mixed with computations.

*Execution Cache Memory* (ECM) [21] is also another insightful approach to model performance of memory-bound applications. This model is built under the similar assumptions as the CARM when modeling the performance of processing elements and memory levels, e.g., by considering their maximum throughput. However, the ECM aims at predicting the application runtime whereas the CARM aims at providing insights toward application characterization and optimization. Moreover, to the best of our knowledge, their are no studies demonstrating the usability of the ECM for NUMA and heterogeneous memory systems featuring emerging heterogeneous memory technologies.

Our contribution to the CARM also advances its current implementation in the Intel proprietary tool's, referred as Intel Advisor Roofline [22], and for which some author of this paper published concrete cases usage [23]. Unlike Intel Advisor Roofline, we keep track of the MCDRAM bandwidth in several aspects, and provide additional insights about potential bottlenecks and characteristics of NUMA systems. Indeed, we demonstrated that our model improvements can efficiently spot locality related issues, and provide more insights, especially in the case of traditional multi-socket systems.

## 7 Conclusions

The trend of increasing the number of cores on-chip is enlarging the gap between compute power and memory performance. This issue leads to design systems with

heterogeneous memories, creating new challenges for data locality. Before the release of those memory architectures, the Cache-Aware Roofline Model offered an insightful model and methodology to improve application performance with knowledge of the cache memory subsystem. With the help of hwloc library, we are able to leverage the machine topology to extend the CARM for modeling NUMA and heterogeneous memory systems, by evaluating the memory bandwidths between all combinations of cores and NUMA nodes.

Our contribution scopes most contemporary types of large compute nodes and characterizes three bottlenecks typical of those systems, namely contention, congestion and remote access. We showed that this additional information can help to successfully spot locality issues coming from parameters such as data allocation policy or memory configuration. To do so, we emphasized on several validation stages, ranging from micro-benchmarks to real-world applications on both a dual-Broadwell Xeon host and on an Intel Knight Landing processor. The LARM extension remains consistent with the traditional Cache-Aware Roofline Model while including a minimum of changes to the original methodology.

In the future we intend to validate the model also on larger systems embedding tens of NUMA nodes and probably yielding even more interest for locality aware modeling. It would also be interesting to investigate an extension of the model over the network in order to include distributed workloads characterization. Also, as mentioned in Sect. 3 footnote, we only consider the load bandwidth in the paper. However most applications mix load and store instructions and the top achievable roof in that case is neither the load bandwidth nor the store bandwidth but rather a combination of those. Additional constraints could also be added to the load/store mix in order to define a roof, but this would deserve a paper on its own. It could end up with an automatic roof matching features, which as for now, is left to the user.

**Acknowledgments.** We would like to acknowledge COST Action IC1305 (NESUS) and Atos for funding parts of this work, as well as national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013.

Some experiments presented in this paper were carried out using the PLAFRIM experimental testbed, being developed under the Inria PlaFRIM development action with support from Bordeaux INP, LaBRI and IMB and other entities: Conseil Régional d'Aquitaine, Université de Bordeaux and CNRS (and ANR in accordance to the programme d'investissements d'Avenirs, see https://www.plafrim.fr/).

## References

1. Blake, G., Dreslinski, R.G., Mudge, T.: A survey of multicore processors. IEEE Signal Process. Magaz. **26**(6), 26–37 (2009)
2. Blagodurov, S., Zhuravlev, S., Dashti, M., Fedorova, A.: A case for NUMA-aware contention management on multicore systems. In: 2011 USENIX Annual Technical Conference, Portland, OR, USA, 15–17 June 2011 (2011)
3. Reinders, J., Jeffers, J., Sodani, A.: Intel Xeon Phi Processor High Performance Programming Knights Landing Edition (2016)

4. Ziakas, D., Baum, A., Maddox, R.A., Safranek, R.J.: Intel® quickpath interconnect architectural features supporting scalable system architectures. In: 2010 IEEE 18th Annual Symposium on High Performance Interconnects (HOTI), pp. 1–6. IEEE (2010)
5. Bull atos technologies: Bull coherent switch. http://support.bull.com/ols/product/platforms/hw-extremcomp/hw-bullx-sup-node/BCS/index.htm
6. Ilic, A., Pratas, F., Sousa, L.: Cache-aware roofline model: upgrading the loft. IEEE Comput. Archit. Lett. **13**(1), 21–24 (2014)
7. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM **52**(4), 65–76 (2009)
8. Cantalupo, C., Venkatesan, V., Hammond, J., Czurlyo, K., Hammond, S.D.: Memkind: an extensible heap memory manager for heterogeneous memory platforms and mixed memory policies. Technical report, Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States) (2015)
9. Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., Namyst, R.: hwloc: a generic framework for managing hardware affinities in HPC applications. In: The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP 2010), Pisa, Italy. IEEE, February 2010
10. Kleen, A.: A NUMA API for LINUX. Novel Inc. (2005)
11. Lepers, B., Quema, V., Fedorova, A.: Thread and memory placement on NUMA systems: asymmetry matters. In: 2015 USENIX Annual Technical Conference (USENIX ATC 2015), Santa Clara, CA, pp. 277–289. USENIX Association, July 2015
12. Chou, C., Jaleel, A., Qureshi, M.K.: CAMEO: a two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. In: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47), Washington, DC, USA, pp. 1–12. IEEE Computer Society (2014)
13. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K.: The NAS parallel benchmarks. Int. J. Supercomput. Appl. **5**, 63–73 (1991). Technical report
14. Karlin, I., Keasler, J., Neely, R.: Lulesh 2.0 updates and changes. Technical report LLNL-TR-641973, August 2013
15. Lepers, B., Quéma, V., Fedorova, A.: Thread and memory placement on NUMA systems: asymmetry matters. In: USENIX Annual Technical Conference, pp. 277–289 (2015)
16. Ramos, S., Hoefler, T.: Capability Models for Manycore Memory Systems: A Case-Study with Xeon Phi KNL (2017)
17. The Memkind Library. http://memkind.github.io/memkind
18. Ilic, A., Pratas, F., Sousa, L.: Beyond the roofline: cache-aware power and energy-efficiency modeling for multi-cores. IEEE Trans. Comput. **66**(1), 52–58 (2017)
19. Doerfler, D., et al.: Applying the roofline performance model to the intel xeon phi knights landing processor. In: Taufer, M., Mohr, B., Kunkel, J.M. (eds.) ISC High Performance 2016. LNCS, vol. 9945, pp. 339–353. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46079-6_24
20. Lorenzo, O.G., Pena, T.F., Cabaleiro, J.C., Pichel, J.C., Rivera, F.F.: Using an extended roofline model to understand data and thread affinities on NUMA systems. Ann. Multicore GPU Program. **1**(1), 56–67 (2014)

21. Hofmann, J., Eitzinger, J., Fey, D.: Execution-cache-memory performance model: introduction and validation. CoRR abs/1509.03118 (2015)
22. Intel: Intel Advisor Roofline (2017)
23. Marques, D., Duarte, H., Ilic, A., Sousa, L., Belenov, R., Thierry, P., Matveev, Z.A.: Performance analysis with cache-aware roofline model in intel advisor. In: 2017 International Conference on High Performance Computing Simulation (HPCS), pp. 898–907, July 2017

# A Scalable Analytical Memory Model for CPU Performance Prediction

Gopinath Chennupati[1(✉)], Nandakishore Santhi[1], Robert Bird[1], Sunil Thulasidasan[1], Abdel-Hameed A. Badawy[2], Satyajayant Misra[3], and Stephan Eidenbenz[1]

[1] Los Alamos National Laboratory, SM 30, Los Alamos, NM 87545, USA
{gchennupati,nsanthi,bird,sunil,eidenben}@lanl.gov
[2] Klipsch School of Electrical and Computer Engineering, New Mexico State University, Las Cruces, NM 88003, USA
badawy@nmsu.edu
[3] Computer Science Department, New Mexico State University, Las Cruces, NM 88003, USA
misra@cs.nmsu.edu

**Abstract.** As the US Department of Energy (DOE) invests in exascale computing, performance modeling of physics codes on CPUs remain a challenge in computational co-design due to the complex design of processors including memory hierarchies, instruction pipelining, and speculative execution. We present Analytical Memory Model (AMM), a model of cache hierarchies, embedded in the Performance Prediction Toolkit (PPT) – a suite of discrete-event-simulation-based co-design hardware and software models. AMM enables PPT to significantly improve the quality of its runtime predictions of scientific codes.

AMM uses a computationally efficient, stochastic method to predict the reuse distance profiles, where reuse distance is a hardware architecture-independent measure of the patterns of virtual memory accesses. AMM relies on a stochastic, static basic block-level analysis of reuse profiles measured from the memory traces of applications on small instances. The analytical reuse profile is useful to estimate the effective latency and throughput of memory access, which in turn are used to predict the overall runtime of an application.

Our experimental results demonstrate the scalability of AMM, where we report the error-rates of three benchmarks on two different hardware models.

**Keywords:** Performance modeling · Cache hierarchies
Reuse distance · Probabilistic models · LLVM · Basic blocks

## 1 Introduction

The US DOE's exascale initiative demands a thousand-fold increase in supercomputing performance to meet the national needs in science, energy, and security.

© Springer International Publishing AG 2018
S. Jarvis et al. (Eds.): PMBS 2017, LNCS 10724, pp. 114–135, 2018.
https://doi.org/10.1007/978-3-319-72971-8_6

The transition to exascale computing poses hard challenges in the form of design of future architectures. Moreover, confining to modulate either of the software or hardware is insufficient to meet the design goals. Co-design helps to trade-off the hardware designs and code development. Most of the research in co-design has been aimed at getting cycle accurate simulations in exploring the design space. Recent developments encourage novel performance modeling frameworks due to the black-box nature of the cycle accurate simulators [3]. Especially, cycle accurate simulators are slow and hinder the factors that contribute to the design of processors. Apart from the speed/slowness, many of these simulators are old while the modern processors are far more advanced than many of those models. Furthermore, the validation of these simulators is not as exhaustive as it should be, yet they are accepted in the research community. With that motivation, rapid performance prediction of computational codes on potential hardware architectures is a crucial requirement for pushing forward towards the exascale era.

In co-tuning the hardware and software parameters for physics codes, we introduce a novel framework, *Analytical Memory Model* (AMM), to explore the design space. AMM contains a compiler-driven static analysis of applications and a hardware-driven performance model. The compiler-driven analysis identifies the basic blocks (contain no loops and branches with a single entry and exit points) of a program, for which, an off-line analysis calculates the exact probability of executing a basic block. The hardware model is unique among the family of exascale co-design models with its capability to scale while considering the hardware specific factors such as frequency, latency, throughput, and cache. For the execution time, we consider the reuse distance [24] (the number of unique memory references between two references to the same addresses) and the number of CPU operations. We measure the total execution time of CPU operations using the pre-calculated instruction latencies.

In measuring the memory access time, we estimate a distribution of reuse distances from the memory trace of an application at a smaller input size. We randomly sample for each basic block and measure the conditional reuse distance profiles. These profiles together with the probability of executing a basic block results in the overall reuse profile of a program. The resultant reuse profiles help us estimate the availability of data (conditional hit rates) for a processor through various cache hierarchies. With the hit-rates, we measure the effective latency and throughput per memory operation. With the latency and throughput at hand, we measure the total memory access time of a program. The predicted runtime of an application is the sum of the time required for CPU operations and the total memory access time.

We evaluate AMM on three benchmarks: STREAM [23], Matrix Multiplication [15], and BlackScholes [6], on two hardware models – Intel Xeon and Intel Core i7. The results show that the sampled reuse profiles are similar to the real profiles, while the characteristic behavior of predicted runtimes is similar to actual runtimes on all benchmarks. Using the predicted runtimes, AMM offers insights into the optimal combination of hardware models for software applications when run in serial mode.

The rest of the paper is organized as follows: Sect. 2 presents the background; Sect. 3 describes AMM, Sect. 4 shows the experiments and the results; Sect. 6 concludes and recommends future research.

## 2 Background

### 2.1 Performance Modeling

Although the question, *How much execution time and energy does my algorithm cost?* [10] is not entirely new, but it helps to justify the trade-offs of the design decisions (time, energy, power, throughput, and latency). Since performance modeling with cycle-accurate simulations is too slow and cannot scale to large core counts, the framework in [34] introduced scalable performance prediction on the then HPC systems. Their prediction contains the simulation of an interconnect and a single processor performance, but unfortunately that does not scale on modern HPC machines.

Bailey and Snavely [4] developed an approach for performance prediction, which helps the stakeholders (system designers, co-design centers, and computational scientists) to improve the performance of applications.

For an optimal design decision, İpek *et al.* [18] explored the design space using neural networks, where they devised a non-linear regression model for which the data points in the design space are sampled at regular intervals. A machine learning framework, VERITAS [19], used sparse coding [27], that identified the performance characteristics (efficiency and resource significance) of proxy applications on a node. VERITAS compared the performance of proxy and real codes, which identified the factors that contribute to loss of efficiency. Another machine learning attempt [20] employed decision-trees on communication data and network hardware counters. These trees derived a strong correlation among a set of network features that contribute to the runtime.

In contrast, AMM accounts for factors such as memory hierarchy, processor latency, and throughput. Our model is intertwined with the Performance Prediction Toolkit (PPT) in predicting the runtimes of physics codes.

Structural Simulation Toolkit (SST) [29], a complex code execution simulator, offers some similar functionality but with different goals; unlike Performance Prediction Toolkit (PPT), relies on replicating control flow (*i.e.,* dynamically executes the application), models messaging behavior, scalable unlike cycle-accurate simulators.

### 2.2 Performance Prediction Toolkit

Performance Prediction Toolkit (PPT) developed at Los Alamos National Laboratory (LANL), is a scalable co-design framework, that has parameterized hardware and middleware models, accepts stylized codes as input and predicts the runtimes. PPT relies on Simian [31], a parallel discrete event simulation engine written in Python, Lua, and JavaScript. In Simian, each computing unit (host,

compute node, CPU core) is an entity. Processes perform their tasks through message exchanges to remain active, sleep, wakeup, begin, and end. Simian advances the simulated time through a *time_compute*() function, that takes a *task list* – the number of CPU operations, memory usage, *etc.* The parameterized models of PPT use the task list to approximate the runtime. The hardware models – interconnects, compute nodes and CPU cores – mimic the lower level hardware processes using regression models resulting from PAPI [8] counters data.

The drawbacks of current PPT models are – regression often relies on inaccurate PAPI data; and the dependence on application developers expertise to explicitly specify the hit-rates. Alternatively, AMM predicts the hit-rates for a given input using an analytical reuse profile, we discuss the state-of-the-art in reuse distance calculation.

### 2.3 Reuse Distance

The reuse distance of a memory reference ($M$) is the number of distinct addresses in the trace after the most recent access to $M$. Memory traces were explored in a number of facets, including performance counters, reuse analysis, and cache behavior [26,33,35]. Our work differs in that, it improves concepts of *in-situ* reuse analysis from a memory trace. The reuse distances are used in defining a *reuse profile*, which is a distribution of reuse distances, that helps to estimate the availability of data in cache.

The compiler generated trace files for most scientific applications are often in tens and/or hundreds of gigabytes. Calculating reuse profiles from such large files is infeasible, moreover, the applications spend enormous amount of computational effort in generating these memory traces. Alternatively, synthetic traces [13] are used to estimate the reuse distributions. Partial Markov Model (PMM) [1] produced random memory references that rely on the existence of original trace and reported inaccuracies in the reuse profiles. Synthetic traces in [13] identified patterns in the memory references based on an analysis of instruction profiling, branches and dependencies. Attempts in [16] adapted least recently used stack models [7] over PMM states to accurately produce synthetic traces, their reuse profiles are accurate but unscalable.

Other attempts that sampled reuse profiles to study data locality include, StatCache [5], presented a probabilistic model that employs sampling to analyze the data locality on realistic workloads. Another sampling and parallelization attempt in [32] accelerated the reuse distance analysis on multi-cores. Unlike, these sampling attempts we use the memory trace of a single run of a program at smaller input size to estimate the reuse profiles at larger inputs. A recent approach [11] presented an analytical model to predict the performance and the energy consumption of a processor using architecture independent characteristics.

Of the attempts to approximate the reuse distance, Ding and Zhong [12] estimated the reuse patterns of a whole program based on training runs of a few small inputs. The model uses dependency analysis to estimate the cache misses

with poor accuracy. In a different attempt, Chatterjee *et al.* [9] applied a set of formulas to characterize the cache misses, which perfectly handles nested loops and non-linear array layouts. Their model lacks the runtime knowledge of loop bounds. Sahoo *et al.* [30] tried to accurately characterize the cache miss count using reuse distances in the context of tensor contraction computations. Recently, reuse distance analysis predicted miss-rate per instruction [14], however, such a fine grained miss-rate estimation fail to scale.

In contrast to the existing attempts, AMM is simple, scalable and relies on Low-Level Virtual Machine (LLVM) [21] basic blocks (BB). We calculate reuse profiles for each BB of a program. These profiles are used to measure the cache hit-rates at different levels, which are used in predicting the runtimes of scientific applications.

## 3 Analytical Memory Model

AMM is a parameterized model for performance prediction, the factors that we consider in the prediction are: reuse distance distribution, latency and throughput of a program. The reuse profile corresponds to modeling different cache hierarchies of a processor in an elegant and scalable manner. These reuse profiles are used in estimating the availability of data from main memory to the processor via different cache levels. Further, we use data availability in calculating the latency and throughput of a program.

Figure 1 shows different steps of AMM in predicting the runtime of a program. AMM accepts a computer program (written in FORTRAN or C/C++) as

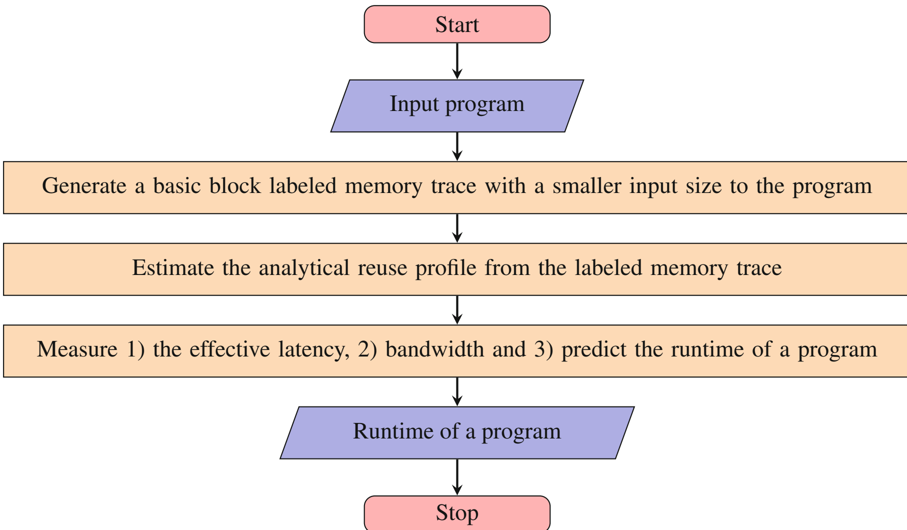



**Fig. 1.** Different steps in analytical memory model (AMM)

an input, which is transformed into an intermediate representation (IR) using the compilation framework, LLVM. The transformation and analysis process involves: *(a)* generating a memory trace with basic block labels produced with a smaller input size of a program, *(b)* estimating the analytical reuse profiles of a program from the labeled memory trace, and *(c)* measuring the effective latency and throughput, with which, program runtime prediction can be made. We describe each step in detail as follows.

### 3.1 Generate Memory Trace

The first step in AMM is to generate a memory trace that contains the LLVM basic blocks. When the source code is compiled to produce IR, the transformed code consists of basic blocks. A basic block is a straight-line code with single entry and exit, with no intermediate branches except a branch at the exit.

The basic block labels in the trace of a program are generated using an LLVM characterization tool, Byfl [28], developed at LANL. We extended Byfl to instrument the memory addresses with LLVM basic block names. Note that LLVM does not create a distinct basic block for the function calls. We resolve such an ineptitude through preprocessing the labeled trace, where we ensure to distinguish the function calls as a separate basic block. For example, the $i^{th}$ basic block ($BB_i$) of the labeled trace contains all the memory addresses that are generated as a result of executing the corresponding straight-line code of $BB_i$. Similar traces can be generated with Valgrind [25] and Pin [22], however, we use Byfl as it is developed using LLVM infrastructure. Like AMM, the attempts in [11] present a similar architecture independent performance and energy modeling.

### 3.2 Estimate Reuse Profile of a Program

The second step is to analytically estimate the reuse profile of a program ($Pr(D)$). The traditional methods of measuring the reuse profile are expensive due to large memory traces. Our technique promises to produce scalable memory traces at smaller inputs of a program, with which we estimate the reuse profiles at larger inputs. With the memory trace using smaller inputs, we estimate the reuse profile of a program as in Eq. 1

$$Pr(D) = \sum_{i=0}^{n(BB)} P(BB_i) \times P(D \mid BB_i) \tag{1}$$

where, $D$ is the reuse distance, $n(BB)$ is the number of basic blocks, $P(BB_i)$ is the apriori probability of executing a basic block and $P(D \mid BB_i)$ is the conditional reuse profile of $i^{th}$ basic block.

Algorithm 1 measures the conditional reuse profile of a basic block, $BB_i$. The algorithm takes the labeled trace as input, identifies all the instances of $BB_i$, from which, randomly select *sample_size* number of occurrences. For example, if a basic block appears hundred times in the trace, we randomly select $n\%$

**Algorithm 1.** Calculating the conditional reuse profile of a basic block ($BB_i$)

```
1: procedure reuse_profile_BB_i(BB_i, memory_trace)
2:     reuse_distances, sampled_wins ← [ ], [ ]
3:     sample_size ← x                                  ▷ x% of all the BB_i(s)
4:     for bb in all_BB_i do
5:         sampled_wins.append([BB_i_start, BB_i_end])
6:     end for
7:     windows ← random(sampled_wins, sample_size)
8:     for window in windows do
9:         reuse_dist ← get_rd(window, memory_trace)
10:        reuse_distances.append(reuse_dist)
11:    end for
12:    uniq_reuse_dist, counts ← unique(reuse_distances)
13:    prob_rd ← map(lambda x: x/len(reuse_distances), counts)
14:    r_prof_i ← zip(uniq_reuse_dist, prob_rd)
15:    return r_prof_i
16: end procedure
```

(typically 1%) of the samples from these occurrences. In fact, the reuse distance distributions are random due to uncertain memory mapping of program data. Therefore, it is important to randomly sample the trace, we term these random samples as *windows*. A window is a list that contains the start and the end indices of a sampled BB. We measure the reuse distances of all the memory addresses in a window, from which, calculate the corresponding probabilities.

**Algorithm 2.** Calculate the reuse distances

```
1: procedure get_rdist(window, memory_trace)
2:     reuse_dist ← [ ]
3:     for idx, addr in enumerate(window) do
4:         window_trace ← memory_trace[: idx]; dict_rd ← { }; addr_found ← False
5:         for addr_idx in range(len(window_trace)) do
6:             w_addr ← window_trace[−addr_idx − 1]
7:             if addr == w_addr then addr_found ← True;   break
8:             end if
9:             dict_rd[w_addr] = True
10:        end for
11:        if addr_found then reuse_dist.append(len(dict_rd))
12:        else reuse_dist.append(−1)
13:        end if
14:    end for
15:    return reuse_dist
16: end procedure
```

Algorithm 2 calculates the reuse distances memory addresses in a window. For each address in a window, we refer back in the trace from the current address

to the exact same address, termed as *max back reference*. Once we find the memory address at two different indexes, the reuse distance for that address is the *cardinality* of the unique addresses between the two indexes. If the second index is absent, the reuse distance is infinite ($\infty$). Similarly, the algorithm continues to measure the reuse distances for all the addresses in a basic block through a search for a *max back reference* in the original trace. At the end, the algorithm returns a list of all the reuse distances for that window.

Algorithms 1 and 2 calculate the reuse distances for all the addresses from all the sampled windows. Finally, we measure the frequency of each reuse distance, where the frequencies produce the respective probabilities. The reuse distances together with the corresponding probabilities form part of the *conditional reuse profile* of $BB_i$, $P(D \mid BB_i)$. The conditional reuse profiles are application dependent, for example, the conditional profiles of some applications may shift with input size. We extrapolate (see Sect. 4) these changes in conditional reuse profiles using polynomial regression techniques. Similarly, $P(BB_i)$ varies with the input size, measured as follows.

**Measure** $P(BB_i)$**:** Let us consider, $BB_1$, $BB_2$, ..., $BB_j$, ..., $BB_{n-1}$, $BB_n$ is a series of basic blocks, any BB can execute any other BB. For example, the basic blocks $BB_1$, $BB_2$, ..., $BB_k$ can execute $BB_j$, where, $BB_1$, ..., $BB_k$ are termed as the predecessors of $BB_j$. Therefore, the predecessor BBs satisfy the following linear recursive relation:

$$N_j = \sum_{i \in Pred(j)} \pi_{ij} \times N_i \tag{2}$$

where, $\pi_{ij}$ is the transition probability (measured off-line using compiler coverage analysis/application developer can identify manually) from predecessor block $BB_i$ to $BB_j$. $N_j$ is a homogeneous system of linear equations with many solutions. Since the entry basic block of most of the source codes is executed once, $N_1$ becomes 1.

Given $\pi_{ij}$, the apriori probability of a basic block ($P(BB_i)$) is defined as in Eq. 3:

$$P(BB_i) = \frac{N_i}{\sum_{k=0}^{n(BB)} N_k} \tag{3}$$

where, $N_i$ and $N_k$ are the number of calls to the $i^{th}$ and $k^{th}$ basic blocks respectively.

$P(BB_i)$ changes with respect to the input size, however, we use the same labeled memory trace at smaller inputs to estimate the reuse profiles for larger instances of the program. We repeat our off-line analysis on $P(BB_i)$ in order to generate the apriori probabilities of basic blocks at bigger inputs. Note, the basic blocks with no memory access in their trace has no contribution towards the final reuse distribution.

### 3.3 Predict Runtime

The final step in AMM is to predict the runtime of an application. In runtime prediction, we measure latency and throughput using the reuse profile. The reuse profile calculates the availability of the data (hit-rates) from main memory to processor via different cache levels. The total predicted runtime of a program is the sum of the average memory access time ($T_{avg_mem}$) and the average time taken for the CPU operations ($T_{CPU_ops}$). The application characterization tool, Byfl is useful in counting the total memory required for the program and the number of CPU operations.

Therefore, the predicted runtime is measured with Eq. 4:

$$T_{pred} = T_{avg_mem} + T_{CPU_ops} \tag{4}$$

**Probability of a Cache-Hit:** In predicting the runtime, identifying the data availability at different cache levels is essential. With the analytical reuse profiles ($Pr(D)$), we measure the cache hit-rates (data availability) employing a *stack distance based cache model* (SDCM) [7], which helps to estimate the probability of a hit at any cache hierarchy ($L_1$, $L_2$, or $L_3$) for a given memory reference with a specific reuse distance. The following formula represents the probability of a hit for an *n-way* associative cache at a given reuse distance ($P(h \mid D)$):

$$P(h \mid D) = \sum_{a=0}^{A-1} \binom{D}{a} \left(\frac{A}{B}\right)^{a} \left(\frac{B-A}{B}\right)^{(D-a)} \tag{5}$$

where $D$ is the reuse distance, $A$ is the associativity and $B$ is cache size in terms of number of blocks (which is cache size over cache line size). For example, an $L_1$ cache of size $64K$ with line size 64 has $B = 1024$ blocks. For a direct-mapped cache, $P(h \mid D)$ is $((B-1)/B)^D$ [7]. Therefore, the unconditional probability of a hit *P(h)* for the entire program can be approximated as in Eq. 6

$$P(h) = \sum_{i=0}^{N} P(D_i) \times P(h \mid D_i) \tag{6}$$

where, $P(D_i)$ is the probability of $i^{th}$ reuse distance ($D$) in a reuse distribution $Pr(D)$. Herein, we investigate two variations (contiguous and non-contiguous) of runtime prediction with respect to the availability of data on memory and/or cache.

**Case 1 (Contiguous): Memory Runtime Prediction.** Assuming the contiguous availability of memory, the average memory access time is measured as in Eq. 7:

$$T_{avg_mem} = \frac{\lambda_{avg} + (b-1) \times \beta_{avg}}{b} \times total_mem \tag{7}$$

where $\lambda_{avg}$ is average latency, $\beta_{avg}$ is average reciprocal throughput, $b$ is block size and $total_mem$ is the total memory required by the program. The latency

and throughput are per memory access, while the block size is considered as word size with the assumption of the availability of contiguous memory. Dividing the first term with block size will result in the average memory access time per byte, multiplying with $total_mem$ results in the total memory access time of a program.

The hit-rates (Eq. 6) at different cache levels estimate the average latency and throughput of a given program. The average latency for a three-level cache is in Eq. 8

$$\lambda_{avg} = P_{L_1}(h) \times \lambda_{L_1} + (1 - P_{L_1}(h)) \Big[ P_{L_2}(h) \times \lambda_{L_2} + (1 - P_{L_2}(h)) \big[ P_{L_3}(h) \times \lambda_{L_3} + (1 - P_{L_3}(h)) \times \lambda_{RAM} \big] \Big] \quad (8)$$

where, $\lambda_{L_1}$, $\lambda_{L_2}$, $\lambda_{L_3}$ and $\lambda_{RAM}$ are the hardware specific measured latencies of $L_1$, $L_2$, $L_3$ caches and $RAM$ respectively; $P_{L_1}(h)$, $P_{L_2}(h)$ and $P_{L_3}(h)$ are the probabilities of a hit for $L_1$, $L_2$ and $L_3$ caches respectively, that are calculated using Eq. 6. Similarly, we measure the average throughput, $\beta_{avg}$ (replace $\lambda$s in Eq. 8 with $\beta$).

**Case 1 (Contiguous): Measure.** $T_{CPU_ops}$ Byfl and/or a simple off-line analysis helps to identify the number of CPU operations (ADD, SUB, and DIV, etc.) of a program. We measure the time required for CPU operations using the hardware specific instruction latencies and the operations count, thus, the total runtime is predicted as $T_{pred}$ (Eq. 4).

**Case 2 (Non-contiguous): Memory Runtime Prediction.** In measuring the average memory access time, as opposed to the previous consideration, we consider the non-contiguous alignment of memory, as is the case in reality. There will be gaps ($v$) in between the required program data, therefore, the new block size ($b$ in Eq. 7 becomes $b^{new}$): $b^{new} = b + v$. However, the entire block may not always be transferred from main memory to different cache levels due to the dependence on factors such as data bus width, and cache size, etc. Therefore, we model such a unique behavior of cache as follows. Let us consider, $b_1^{new}$, $b_2^{new}$, $b_3^{new}$, ..., $b_i^{new}$, ..., $b_n^{new}$ are the blocks of data on main memory, while $C$ be the amount of data transferred on to a cache from main memory at any given time. Thus, the new block size at a given cache size ($B$) can be re-written as:

$$b^{new} = \begin{cases} C & : \textbf{if} \quad b_i^{new} \leq C \\ \left\lceil \dfrac{b_i^{new}}{C} \right\rceil \times C & : \textbf{if} \quad B \geq b_i^{new} \geq C \\ B & : \textbf{if} \quad b_i^{new} \geq B \end{cases}$$

**Case 2 (Non-contiguous): Time for CPU Operations ( $T_{CPU_ops}$).** In case of the time taken for CPU operations, there is a large difference in the

instruction latencies between DIV and the rest of the instructions. Moreover, the time required for CPU operations is dependent on program characteristics, where some applications are instruction latency dependent while others are throughput reliable. Thus, the time for the resultant CPU operations is:

$$T_{CPU_ops} = \begin{cases} \lambda_{in} + (N_{in} - N_{in_div} - 1) \times \beta_{in} + \lambda_{div} + (N_{in_div} - 1) \times \beta_{div} & : \textit{throughput} \\ (N_{in} - N_{in_div} - 1) \times \lambda_{in} + (N_{in_div} - 1) \times \lambda_{div} & : \textit{latency} \end{cases}$$

where $\lambda_{in}$, $\lambda_{div}$, $\beta_{in}$ and $\beta_{div}$ are latencies and throughputs of instructions, ADD/SUB, MUL and DIV respectively, while $N_{in}$ and $N_{in_div}$ are the number of instructions.

## 4 Experiments

In this section, we describe the target architectures and the benchmark applications used in validating our model.

### 4.1 Target Architectures

We use AMM to validate three different benchmark applications on two hardware architectures. Table 1 presents the two processor architectures, each of which uses three cache levels with different sizes. The $L_3$ cache of Intel Xeon processor is shared among the available cores on the chip while that of the Intel Core i7 is unshared.

In predicting the runtimes, we build the hardware models for the two experimental processors (Table 1) along with AMM in Performance Prediction Toolkit (PPT). PPT has parametrized hardware models and software proxy applications. The hardware parameters of PPT are: cache latencies, cache sizes, cache line sizes, associativity, and memory bandwidth (throughput) at different cache levels (we consider the reciprocal throughput), RAM latency, and data bus width. The hardware parameters are measured values for a given processor, reasonably reliable sources include Agner Fog's manual [2], Intel and others[1] present these parameter values for a number of hardware architectures. We can measure

**Table 1.** The target architectures and their parameters

| # | Processor | Speed | Cache size (bytes) | | | Shared |
|---|---|---|---|---|---|---|
| | | (GHz) | $L_1$ | $L_2$ | $L_3$ | L3? |
| 1 | Intel Xeon E5-2695 | 2.10 | 64K | 256K | 45M | Yes |
| 2 | Intel Core i7-4770HQ | 2.20 | 256K | 1M | 6M | No |

[1] http://www.7-cpu.com/cpu/Haswell.html.

these parameters using standard benchmarks, nevertheless, the objective in this paper is performance modeling rather parameter calibrations. The latencies and throughputs used in the hardware model include both at the cache hierarchies and the instructions such as *ADD/SUB*, *MUL*, and *DIV*. The software parameters are: total memory of an application, the number of integer and floating point operations (add, mul, etc.), and the block size (Eq. 7 in Sect. 3.3), measured using Byfl.

### 4.2 Benchmarks

The three benchmark applications we used are: STREAM [23], matrix-matrix multiplication (MM) [15], and BlackScholes [6].

**STREAM** is a memory benchmark with vectors of floating point operations. STREAM contains four kernels: *ADD* performs the sum of two vectors; *SCALE* multiplies a vector with a floating-point scalar; *COPY* assigns one vector into another and *TRIAD* performs the above three operations. We execute all the above four kernels.

**MM** is a naive implementation (*ijk* method that has 3 nested loops) of floating-point matrix-matrix multiplication. MM, in this paper, is defined as $R = \alpha P \times Q + \beta R$, where P, Q and R are m × k, k × n and m × n matrices respectively while $\alpha$ and $\beta$ are floating-point scalars.

**BlackScholes** is a PARSEC benchmark, partial differential model used to predict the European stock option prices. BlackScholes functions within two nested loops, where the outer-loop stands for the number of iterations of the algorithm and the inner loop performs the floating-point operations needed for option prices.

## 5 Results

We implemented the respective proxy application in PPT for all the three benchmarks. We validate AMM for these three applications as follows: (1) compare the real and predicted reuse profiles, and (2) compare the real and predicted runtimes. Both the simulation and actual runs are computed on a single core of a CPU.

### 5.1 Validate Reuse Profile

Our goal is to validate the analytical reuse profiles with that of the actual profiles. The reuse profiles are discrete, in general, they are architecture independent due to which the reuse profiles are same across the two experimental hardware architectures.

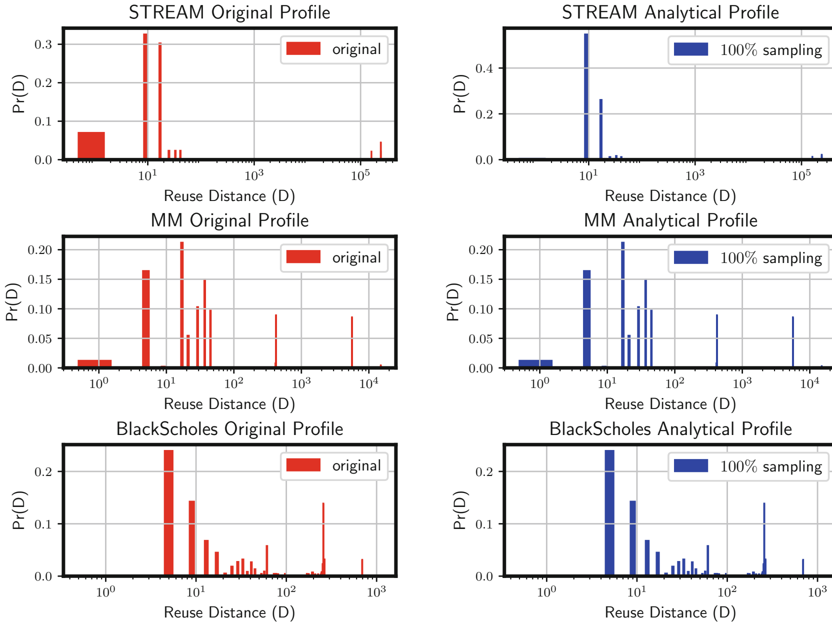



**Fig. 2.** Compare the original (left) and analytical (right) reuse profiles of STREAM (top), MM (middle), and BlackScholes (bottom) at input sizes of 10000 floating-points, matrix of size $25 \times 25$ and 16 data points respectively. Original distribution is measured using a stack based algorithm, while that of the analytical is measured using AMM with 100% sampling. The reuse distance (D) is in *log* scale while Pr(D) is in decimal scale.

Figure 2 compares the actual and the analytical reuse profiles of both the benchmarks. The analytical reuse profiles are prepared with 100% sampling. For example, if a basic block contains ten occurrences, all of them contribute to calculate the conditional reuse profiles before multiplying the probability $(P(BB_i) \times P(D|BB_i))$ of execution of that basic block. We adopted 100% sampling in order to validate the actual and analytical profiles, in the runtime prediction, we consider 1% sampling, which guarantees scalability. On all the three benchmarks, AMM calculated reuse distances (D, on X-axis) are identical to that of the actual reuse distances, so does their number of occurrences. The corresponding probabilities (Pr(D), on Y-axis) are approximately similar, the analytical probabilities are slightly higher at a few reuse distances because of their dependence on the accuracy of $P(BB_i)$. Nonetheless, these inaccuracies have insignificant impact on the final cache hit-rate, therefore, the analytical reuse profiles are similar to the actual.

The original reuse profiles are measured using a stack [24] based implementation that has a time complexity of $O(NM)$. The analytical reuse profiles are measured using Algorithm 1, which has a computational complexity of $O(NSB)$ $\sim O(N)$, since the number of samples $(S)$ and size of the basic block $(B)$ are

constant. The worst case complexity is $O(NM)$, in the case of 100% *sampling*, which will never happen.

### 5.2 Validate Runtime

We validate the AMM predicted runtimes with that of the actual for all the three benchmark applications at different input sizes on both the target architectures. Table 2 presents four different input sizes for each of the three benchmarks. For example, STREAM has three floating-point vectors, all of which are initialized with same input size. The inputs for each run of STREAM varies from 10000, 20000, 30000 to 40000 elements. Similarly, MM and BlackScholes have four square matrix sizes and four datasets (16, 32, 64, and 128 data-points) respectively. We report both the actual and predicted runtimes at four different input sizes on each benchmark.

**Table 2.** Benchmarks with different input sizes.

| # | Program | Input sizes |
|---|---|---|
| 1 | STREAM | {10000, 20000, 30000, 40000} |
| 2 | MM | {25 × 25, 50 × 50, 100 × 100, 200 × 200} |
| 3 | BlackScholes | {16, 32, 64, 128} |

In predicting the runtimes, we analytically estimate the reuse profiles at each input using the memory trace (1% sampling) for the smaller input size of the respective benchmark. For example, in the case of MM, we use the memory trace at an input size of 25 × 25 as the base to estimate the reuse profiles at 50 × 50, 100 × 100 and 200 × 200. The probabilities of basic blocks ($\mathrm{P}(BB_i)$) change as the input size changes.

Figure 3 shows the analytical (1% sampling) reuse profiles of both the benchmarks at different input sizes. The sampled reuse profiles are approximately similar to that of the original, however, some large but relatively rare reuse distances disappear due to random sampling. For example, if a basic block occurrence appears at the bottom of the memory trace, there is a chance to omit such occurrences due to 1% random sampling, thereby, the larger reuse distances disappear. These large values may have significant impact on the cache hit-rates, thus, we propose to extrapolate these reuse distances, similar to Zhong et al. [36], where the prediction of program locality with respect to inputs identifies the data access patterns and builds a parametrized model for extrapolation. In contrast to Zhong et al., we extrapolate the conditional reuse distances of basic blocks (instead of the whole program) at larger input sizes of a program using the reuse distances at a few smaller inputs. In fact, extrapolating the conditional reuse profiles of basic blocks using small input reuse distances preserves our promise of scalable AMM.

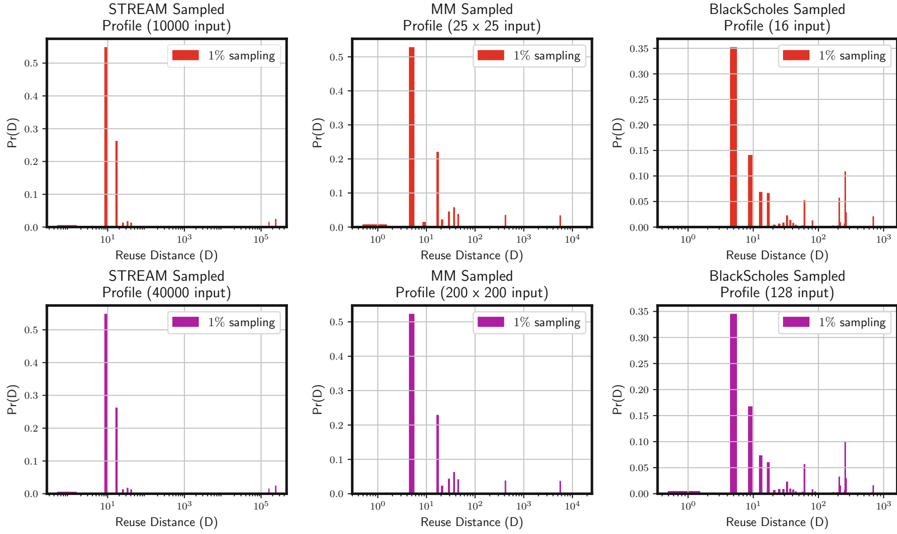



**Fig. 3.** Sampled analytical reuse profiles (reuse distance (D) is on *log scale*) of the three benchmarks: STREAM (left), matrix multiplication (middle), and BlackScholes (right) at different input sizes. The rate of sampling is 1%, while the base memory traces for the three benchmarks are at input sizes of 10000, 25 × 25, and 16 respectively.

**Extrapolate the Conditional Reuse Distances of a BB.** From the probability distribution of executing the basic blocks, we observe that a few number of the total basic blocks of a program have significant impact on the reuse profiles. Figure 4 (left) shows the probability of executing each basic block ($P(BB_i)$) at different small input sizes (10, 12, 15, 17, and 20) of MM. Of all the *twenty two* basic blocks of a MM program, $BB15 - BB17$ have relatively significant contribution over the remaining basic blocks. Empirically, the number of entries in the conditional reuse profiles of these three basic blocks grow with the input size, while that of the remaining basic blocks remain consistent irrespective of the input size. Therefore, extrapolating the conditional reuse distances of these significant BBs helps in identifying the missing large reuse distances.

We explain the extrapolation strategy on one of the three BBs, $BB15$, where we find that the first few (seven for MM) reuse distance entries of the distribution remain unchanged irrespective of the inputs. Probability of these reuse distances contribute 75% of the distribution, while the other growing reuse distances contribute the remaining 25%. Since these initial entries are consistent, what the following linear relation (Eq. 9) predicts is useful in estimating the reuse distances at any input size ($x$).

$$D_i^{'}|x = D_i \quad \forall\, i = 1 \dots 7 \tag{9}$$

where, $D_i^{'}|x$ is new reuse distance at an input, $D_i$ is the reuse distance of a basic block.

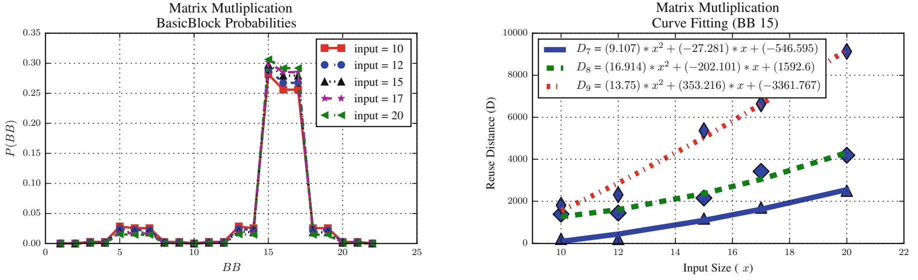



**Fig. 4.** Probabilities (left) of all basic blocks (BBs) of MM at multiple small inputs. Extrapolation of reuse distances (right) as a function of input size ($x$) for matrix multiplication using the data from five small runs at five different input sizes.

We extrapolate the remaining reuse distance entries that grow with the input size, where the number of these entries are inconsistent at each input size. In order to regulate these inconsistencies, we apply a *fixed-binning* strategy, in which, we use a constant number of bins, each of which represents an average of the subset-of-reuse-distances. The number of entries within a bin changes while the total number of bins remain same.

Figure 4 (right) shows the extrapolation of three bins using five small input sizes. The points represent the average reuse distances for each bin while the curves represent the predicted polynomial fit for each bin as a function of input size ($x$). Note, the input ($x$) in the extrapolated curves is on one-dimension of MM. We observe that the predicted average reuse distances grow in polynomial fashion. Similarly, n we can estimate the respective probabilities of these reuse distances, together forms the extrapolated conditional reuse profile of a BB. We can increase the number of bins, however, estimating the hit-rates relies on the magnitude of the reuse distances rather the distinct number of reuses alone. That way, we extrapolate the conditional reuse profiles of the most significant basic blocks of an application and combine the reuse profiles of all the basic blocks to produce the complete reuse profile of a program.

Our prediction strategy does not incur the extra computational overhead of extrapolating the reuse profiles on the whole program (as opposed to Zhong et al., therefore, AMM is scalable), while approximates the hit-rates with reasonably good accuracy.

**Is the Data Available for Use by the Processor?** Given the reuse profiles, it is essential to analyze the availability of data for the processor. Figure 5 shows the conditional cache hit-rates at a given reuse distance for three different cache sizes ($L_1$, $L_2$, and $L_3$). The results are for the input sizes of 10000, $25 \times 25$ and 16 of STREAM, MM and BlackScholes respectively. Since the reuse distances are independent of the underlying hardware, we use the same reuse profile (with respect to the benchmark) to measure the cache hit-rates at different cache sizes. However, the conditional hit-rates at a given stack distance are calculated

on *Intel Xeon E5-2695* architecture. The reuse distance ($D$) is on a *log scale*, whereas $B_1$, $B_2$ and $B_3$ are cache sizes measured in terms of the number of blocks (cache-size/cache-line-size). $P_{L_1}(h|D)$, $P_{L_2}(h|D)$ and $P_{L_3}(h|D)$ are conditional hit-rates at three cache levels $L_1$, $L_2$ and $L_3$ respectively. On all the benchmarks, the cache hit-rate at a reuse distance ($P_{L_1}(h|D)$) suddenly drops for $L_1$ cache after the cache size ($B_1$), which confirms that the application data exceeds the $L_1$ cache of Intel Xeon processor. A similar behavior is found on $L_2$ cache in the case of STREAM and matrix multiplication, while for BlackScholes the data exists on $L_2$ cache. STREAM data slightly exceeds the $L_3$ limits, while the data of the remaining two benchmarks is available on $L_3$. We found that the probability of the corresponding large reuse distances ($P(D_i)$ in Fig. 3) is approximately zero.

However, for *Intel Core i7-4470HQ* – BlackScholes data exists in $L_1$ cache and the remaining two benchmarks data does not exist; on $L_2$, STREAM data is not present while the remaining two benchmarks data does exist; $L_3$ can hold the data for all the three benchmarks. Since Intel Core i7 has relatively large $L_1$ and $L_2$ cache sizes, the data is readily available for the processor. Intel Core i7 have relatively smaller $L_3$ cache size compared with that of Intel Xeon. Intel Core i7 processors $L_3$ capacity is insufficient for large input sizes of a program. In addition, $L_3$ cache of Intel Xeon is shared among the available cores while that is not the case with Core i7. With these characteristics, reuse distances that exceed the cache sizes are always a miss. These observations (Fig. 5) suggest that the availability of data in the cache depends on the target architectures and the application data requirements.

A discussion on the locality of data is out of the scope. However, this study shows that the 1% sampled reuse profiles are reasonably better approximations in estimating the runtime of an application. Therefore, for better availability of data, we suggest to design a processor with the $L_1$ and $L_2$ caches of Intel Core i7 and the $L_3$ of Intel Xeon.

**Prediction of Run-Times:** We validate the predicted runtimes, Fig. 6 presents the error-rates of the AMM predicted runtimes when compared wxith that of the actual for all the three benchmarks at different input sizes on the two target architectures. We assume that the processor executes one application at a given time, so that the cache and RAM are available for the application. We observe that Intel Core i7 error-rates of STREAM are significantly higher than the Intel Xeon due to small $L_3$ cache size of Core i7. For the remaining two benchmarks (MM and BlackScholes), the difference in the predicted error-rates across both the target architectures is insignificant, The reason being the fact that the application data for these two benchmarks fits in the cache hierarchy.

We observe that AMM over-predicts the runtimes when compared to the actual runtimes, especially, at larger input sizes. Although we over-predict, the characteristic behavior of the runtimes with respect to the input remains in coherence with the actual runtimes. The reason behind the over-prediction is due to the fact that AMM is purely a *memory model*. We can reduce such over-prediction through a model for pipelines along with the memory model. AMM

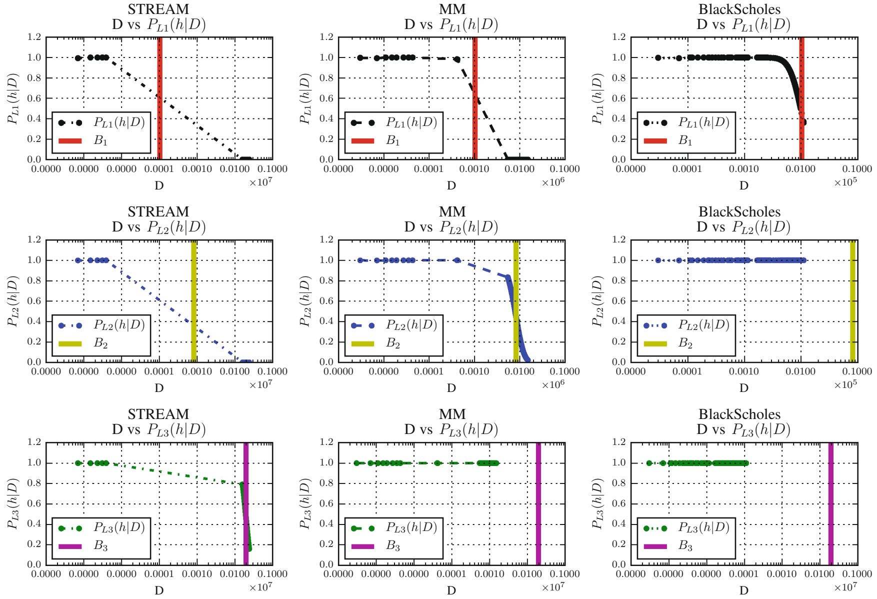



**Fig. 5.** Conditional cache hit-rates at a given reuse distance for all the three benchmarks – STREAM (left), matrix multiplication (middle) and BlackScholes (right). $B_1$, $B_2$ and $B_3$ are the cache sizes in terms of number of blocks (see Sect. 3.2) for $L_1$, $L_2$ and $L_3$ caches respectively.

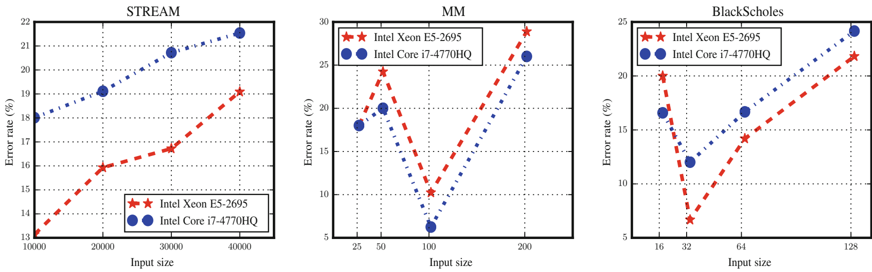



**Fig. 6.** Error-rates of predicted runtimes (with respect to the actual runtimes) for the three benchmark applications (STREAM, matrix multiplication (MM) and BlackScholes) on both the target architectures (Intel Xeon E5-2695, Intel Core i7-4770HQ).

assumes the execution of the program in complete sequential mode, whereas the actual CPU core executes the independent tasks simultaneously through pipelines. In addition to pipelines, factors such as prefetching, replacement strategy, TLB, vector operations, micro-architecture, and etc. can have higher dividends in performance prediction. Building a parameterized model (one of our future directions to investigate) using these factors that work hand-in-hand with AMM would reduce the over-prediction in runtimes. Although the pipeline effect

is not present in AMM, the predicted runtimes are reasonably abreast with that of the actual while we also claim that the characteristic behavior of the predicted runtimes is akin to the actual runtimes of all the benchmarks on both the target architectures.

Between Intel Xeon and Core i7, the latter is much faster than the former on these set of benchmarks due to higher clock speed. Our observations (Fig. 5) in cache sizes play a significant role in making the data available for processor, which obviously impacts the performance of an application. Intel Core i7 clearly has larger $L_1$, $L_2$ caches and a smaller $L_3$ cache, which in fact, is insufficient for large applications that might have adverse effects on performance despite processor speed. Intel states that the *Broadwell* family of Xeon processors are less powerful and energy efficient compared to the *Haswell* of Intel Core i7. With our study, we believe that increasing the $L_1$ and $L_2$ cache sizes of Xeon processors might further boost the performance with little/minimum effect on energy consumption, especially, when the execution of an application becomes concurrent/parallel.

## 6 Conclusion

We presented a novel analytical memory model (AMM) that produces basic block labeled memory traces using LLVM instrumentation. The memory traces at smaller inputs are randomly sampled to produce the reuse distance distributions at larger inputs for scientific applications. Using the smaller input memory traces, reuse distance profiles of the applications are estimated at larger input sizes. The analytically measured reuse profiles are similar to the actual reuse profiles. Further, the estimated reuse profiles are used to predict the runtimes of the applications. Our hardware model consists of low-level details such as latency, throughput of different hardware components (cache levels, RAM, etc.) and CPU instructions (*add*, *sub*, *mul*, etc.). The runtime results are consistent with the real runtimes while the characteristic behavior of the predicted runtimes is similar to that of the actual runtimes. We observed that AMM over-predicted the runtimes due to nonexistence of pipeline, cache prefetching, hardware threads, and TLB in the hardware model. Developing and integrating these missing models would guarantee a close prediction, therefore, is one of our future directions. With the addition of pipelines in AMM, similar to [4,17,20], we aim to predict the performance of MPI aware applications. Nevertheless, having AMM like fine-grained hardware model is essential for accurate and scalable performance prediction in distributed environments.

## References

1. Agarwal, A., Hennessy, J., Horowitz, M.: An analytical cache model. ACM Trans. Comput. Syst. **7**(2), 184–215 (1989)
2. Agner, F.: Instruction tables: lists of instruction latencies, throughputs and micro-operation breakdowns for intel, AMD and VIA CPUs. Technical University of Denmark, Copenhagen, Denmark (2016)

3. Austin, T., Larson, E., Ernst, D.: Simplescalar: an infrastructure for computer system modeling. Computer **35**(2), 59–67 (2002)
4. Bailey, D.H., Snavely, A.: Performance modeling: understanding the past and predicting the future. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 185–195. Springer, Heidelberg (2005). https://doi.org/10.1007/11549468_23
5. Berg, E., Hagersten, E.: StatCache: a probabilistic approach to efficient and accurate data locality analysis. IEEE Int. Symp. ISPASS Perform. Anal. Syst. Softw. **2004**, 20–27 (2004)
6. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The parsec benchmark suite: characterization and architectural implications. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, PACT 2008, New York, NY, USA, pp. 72–81. ACM (2008)
7. Brehob, M., Enbody, R.: An analytical model of locality and caching. Technical report MSU-CSE-99-31 (1999)
8. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. Int. J. High Perform. Comput. Appl. **14**(3), 189–204 (2000)
9. Chatterjee, S., Parker, E., Hanlon, P.J., Lebeck, A.R.: Exact analysis of the cache behavior of nested loops. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, PLDI 2001, New York, NY, USA, pp. 286–297. ACM (2001)
10. Choi, J.W., Vuduc, R.W.: How much (execution) time and energy does my algorithm cost? XRDS **19**(3), 49–51 (2013)
11. den Steen, S.V., Eyerman, S., Pestel, S.D., Mechri, M., Carlson, T.E., Black-Schaffer, D., Hagersten, E., Eeckhout, L.: Analytical processor performance and power modeling using micro-architecture independent characteristics. IEEE Trans. Comput. **65**(12), 3537–3551 (2016)
12. Ding, C., Zhong, Y.: Predicting whole-program locality through reuse distance analysis. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, PLDI 2003, pp. 245–257. ACM (2003)
13. Eeckhout, L., de Bosschere, K., Neefs, H.: Performance analysis through synthetic trace generation. In: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2000, Washington, DC, USA, pp. 1–6. IEEE (2000)
14. Fang, C., Carr, S., Önder, S., Wang, Z.: Reuse-distance-based miss-rate prediction on a per instruction basis. In: Proceedings of the 2004 Workshop on Memory System Performance, MSP 2004, New York, NY, USA, pp. 60–68. ACM (2004)
15. Gunnels, J.A., Henry, G.M., van de Geijn, R.A.: A family of high-performance matrix multiplication algorithms. In: Alexandrov, V.N., Dongarra, J.J., Juliano, B.A., Renner, R.S., Tan, C.J.K. (eds.) ICCS 2001. LNCS, vol. 2073, pp. 51–60. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45545-0_15
16. Hassan, R., Harris, A., Topham, N., Efthymiou, A.: Synthetic trace-driven simulation of cache memory. In: 21st International Conference on Advanced Information Networking and Applications Workshops, vol. 1 of AINAW 2007, pp. 764–771 (2007)
17. Ipek, E., de Supinski, B.R., Schulz, M., McKee, S.A.: An approach to performance prediction for parallel applications. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 196–205. Springer, Heidelberg (2005). https://doi.org/10.1007/11549468_24

18. Ipek, E., McKee, S.A., Caruana, R., de Supinski, B.R., Schulz, M.: Efficiently exploring architectural design spaces via predictive modeling. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XII, New York, NY, USA, pp. 195–206. ACM (2006)
19. Islam, T.Z., Thiagarajan, J.J., Bhatele, A., Schulz, M., Gamblin, T.: A machine learning framework for performance coverage analysis of proxy applications. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Piscataway, NJ, USA, pp. 46:1–46:12. IEEE (2016)
20. Jain, N., Bhatele, A., Robson, M.P., Gamblin, T., Kale, L.V.: Predicting application performance using supervised learning on communication features. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2013, New York, NY, USA, pp. 95:1–95:12. ACM (2013)
21. Lattner, C., Adve, V.: Llvm: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, CGO 2004, Washington, DC, USA, pp. 75–87. IEEE (2004)
22. Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2005, New York, NY, USA, pp. 190–200. ACM (2005)
23. Luszczek, P.R., Bailey, D.H., Dongarra, J.J., Kepner, J., Lucas, R.F., Rabenseifner, R., Takahashi, D.: The hpc challenge (hpcc) benchmark suite. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC 2006, New York, NY, USA. ACM (2006)
24. Mattson, R.L., Gecsei, J., Slutz, D.R., Traiger, I.L.: Evaluation techniques for storage hierarchies. IBM Syst. J. **9**(2), 78–117 (1970)
25. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2007, New York, NY, USA, pp. 89–100. ACM (2007)
26. Nguyen, A.T., Bose, P., Ekanadham, K., Nanda, A., Michael, M.: Accuracy and speed-up of parallel trace-driven architectural simulation. In: Proceedings 11th International Parallel Processing Symposium, pp. 39–44. IEEE (1997)
27. Olshausen, B.A., Field, D.J.: Sparse coding with an overcomplete basis set: a strategy employed by v1? Vis. Res. **37**(23), 3311–3325 (1997)
28. Pakin, S., McCormick, p.: Hardware-independent application characterization. In: International Symposium on Workload Characterization (IISWC), Portland, Oregon, USA, pp. 111–112. IEEE (2013)
29. Rodrigues, A.F., Murphy, R.C., Kogge, P., Underwood, K.D.: The structural simulation toolkit: exploring novel architectures. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC 2006, New York, NY, USA, p. 157. ACM (2006)
30. Sahoo, S.K., Panuganti, R., Sadayappan, P., Krishnamoorthy, P.: Cache miss characterization and data locality optimization for imperfectly nested loops on shared memory multiprocessors. In: Proceeding of the 19th IEEE International Parallel and Distributed Processing Symposium, pp. 44–53 (2005)

31. Santhi, N., Eidenbenz, S., Liu, J.: The simian concept: parallel discrete event simulation with interpreted languages and just-in-time compilation. In: Proceedings of the 2015 Winter Simulation Conference (WSC), pp. 3013–3024. IEEE (2015)
32. Schuff, D.L., Kulkarni, M., Pai, V.S.: Accelerating multicore reuse distance analysis with sampling and parallelization. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT 2010, New York, NY, USA, pp. 53–64. ACM (2010)
33. Sherwood, T., Perelman, E., Hamerly, G., Calder, B.: Automatically characterizing large scale program behavior. In: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS X, New York, NY, USA, pp. 45–57. ACM (2002)
34. Snavely, A., Carrington, L., Wolter, N., Labarta, J., Badia, R., Purkayastha, A.: A framework for performance modeling and prediction. In: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, SC 2002, Los Alamitos, CA, USA, pp. 1–17. IEEE (2002)
35. Weinberg, J., McCracken, M.O., Strohmaier, E., Snavely, A.: Quantifying locality in the memory access patterns of hpc applications. In: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, SC 2005, Washington, DC, USA, pp. 50–61. IEEE (2005)
36. Zhong, Y., Shen, X., Ding, C.: Program locality analysis using reuse distance. ACM Trans. Program. Lang. Syst. **31**(6), 20:1–20:39 (2009)

# Modeling UGAL on the Dragonfly Topology

Md Atiqul Mollah[1](✉), Peyman Faizian[1], Md Shafayat Rahman[1], Xin Yuan[1], Scott Pakin[2], and Michael Lang[2]

[1] Florida State University, Tallahassee, FL, USA
{mollah,faizian,rahman,xyuan}@cs.fsu.edu
[2] Computer, Computational, and Statistical Sciences Division, Los Alamos National Laboratory, Los Alamos, NM, USA
{pakin,mlang}@lanl.gov

**Abstract.** The Dragonfly topology has been proposed and deployed as the interconnection network topology for next-generation supercomputers. Practical routing algorithms developed for Dragonfly are based on a routing scheme called *Universal Globally Adaptive Load-balanced routing with Global information* (UGAL-G). While UGAL-G and UGAL-based practical routing schemes have been extensively studied, all existing results are based on simulation or measurement. There is no theoretical understanding of how the UGAL-based routing schemes achieve their performance on a particular network configuration as well as what the routing schemes optimize for. In this work, we develop and validate throughput models for UGAL-G on the Dragonfly topology and identify a robust model that is both accurate and efficient across many Dragonfly variations. Given a traffic pattern, the proposed models estimate the aggregate throughput for the pattern accurately and effectively. Our results not only provide a mechanism to predict the communication performance for large scale Dragonfly networks but also reveal the inner working of UGAL-G, which furthers our understanding of UGAL-based routing on Dragonfly.

## 1 Introduction

The Dragonfly topology features a cost-effective interconnect design. It is scalable and supports high aggregate throughput capacity at a lower cost in comparison to other alternatives such as fat-trees [1]. Dragonfly has been deployed in the Cray Cascade architecture [2] and in current supercomputers such as Cori [3] and Trinity [4].

To achieve high performance in the Dragonfly topology, different routing schemes must be used for different traffic patterns [1]. In particular, minimal routing (MIN) is better suited to uniform traffic while non-minimal Valiant Load-balanced routing (VLB) is essential for achieving good performance on adversarial traffic patterns. To unify the two routing schemes in one system, the Universal Globally Adaptive Load-balanced routing (UGAL) [1] was developed to adapt the routing decision for each packet between MIN and VLB paths based

© Springer International Publishing AG 2018
S. Jarvis et al. (Eds.): PMBS 2017, LNCS 10724, pp. 136–157, 2018.
https://doi.org/10.1007/978-3-319-72971-8_7

on the occupancy of packet queues [5]. The theoretical UGAL with perfect global link state information (UGAL-G) achieves high performance on Dragonfly [1], and performs similarly as MIN for uniform traffic and as VLB for adversarial traffic.

While UGAL-G is an ideal scheme that cannot be perfectly implemented, it is the foundation of practical routing schemes developed for Dragonfly [2,6]. These practical adaptive routing schemes, including the one used in Cray Cascade [2], are based on UGAL and approximate the performance of UGAL-G. As such, the performance characteristics of UGAL-G is representative of all UGAL-based adaptive routing schemes.

Although UGAL-G and UGAL-based routing schemes have been extensively studied, all existing results are obtained through simulation and measurement. To the best of our knowledge, no theoretical model for UGAL-based routing has been developed. As such, the theoretical understanding of UGAL is lacking. For example, it is unclear how effectively these routing schemes can utilize the path diversity of a given network configuration and how sensitive the routing performances are to any change in local as well as in global network connectivity. An analysis of UGAL-G along this direction provides useful information to the problem of provisioning links and bandwidths on different Dragonfly designs.

In this work, we develop effective throughput models using linear programming (LP) for UGAL-G on the Dragonfly topology and identify a robust model for many Dragonfly variations that is both accurate and efficient. There are several theoretical as well as practical implications of our contribution. First, our proposed theoretical throughput models can accurately and efficiently predict the aggregate throughput for large scale Dragonfly networks. Second, the models reveal the implicit rate allocation in UGAL-G and thus, further our understanding of UGAL-based routing schemes. Third, the proposed models can be applied in many practical situations. For example, the models allow for efficiently exploring the design space of potential Dragonfly configurations and thus, enabling faster design prototyping before a detailed simulation on selected designs is performed. The models also give rate allocation that is competitive with UGAL-G. They can be applied to solve traffic engineering optimization problems in Software Defined Networking (SDN) architectures [7] to find rate allocation schemes that are competitive to adaptive routing in the SDN environment.

Given a traffic pattern and a Dragonfly topology, our models estimate the aggregate throughput for the pattern under the *maximum concurrent flow (MCF) model*, which is commonly used to model the throughput performance of interconnects [8–11]. The models are validated through simulations with a flit-level simulator, Booksim [12]. The results demonstrate that to accurately model UGAL-G, the LP formulations need only a small number of variables per flow. This enables the models to be used for large-scale systems with tens of thousands of flows. The study also reveals that even with the precise global network state information, UGAL-G does not have effective control over all the paths that are available and does not allocate rates to individual paths to maximize

its performance. Instead, for the general cases when the numbers of MIN and VLB paths are sufficiently large, UGAL-G effectively allocates rates to groups of paths instead of individual paths.

The rest of the paper is structured as follows. Section 2 discusses the background of this work, describing the Dragonfly topology, its variation in Cray Cascade, UGAL-G routing, and the MCF throughput model. Section 3 introduces our performance models for UGAL-G on Dragonfly. Section 4 presents the results of a set of experiments used to validate the models. Section 5 discusses related work. Finally, in Sect. 6 we draw some conclusions from our work.

## 2 Background

### 2.1 Dragonfly Topology

We will briefly introduce the Dragonfly topology. More details about the topology can be found in Kim et al.'s original paper [1]. The Dragonfly topology has a 2-layer structure. A group of low-radix routers/switches are interconnected with an intra-group topology into a *group* that works as a single virtual router with a very high radix. In this paper, the terms router and switch will be used interchangeably. The groups are then connected with some inter-group topology. Figure 1 shows an example of the 2-layer Dragonfly topology. In this example, each group consists of 4 switches; there are a total of 9 groups in the system.

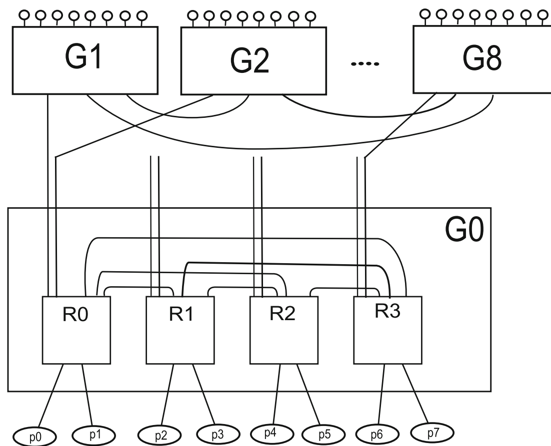



**Fig. 1.** Dragonfly architecture (p=h=2, a=4, g=9)

Various topologies can be used to form the intra-group connectivity. A typical intra-group topology is a fully connected graph where all pairs of switches are directly connected [1]. An example of such an intra-group topology is shown in the G0 group in Fig. 1. The groups in a Dragonfly are also fully connected where

there is at least one *global* link connecting each pair of groups. Such a topology is uniquely defined by four parameters: the number of links per switch connecting to local compute nodes $p$, the number of switches in each group $a$, the number of global links per switch connecting to switches in other groups $h$, and the number of groups $g$. In a fully connected Dragonfly group, the number of links per switch connecting to local switches is $a-1$. We will use the 4-tuple notation $dfly(p, a, h, g)$ to denote such a topology and 3-tuple notation $group(p, a, h)$ to denote an individual Dragonfly group. By definition, the number of ports in each switch in $dfly(p, a, h, g)$ is $p + a - 1 + h$; the number of global links from each group is $a \times h$, and the number of groups, $g$, is thus at most $a \times h + 1$. The number of global links between each pair of groups is $a \times h/(g - 1)$. The total number of switches and the total number of compute nodes in $dfly(p, a, h, g)$ is $a \times g$ and $p \times a \times g$ respectively. As discussed in [1], a load-balanced Dragonfly system should have $a = 2p = 2h$. Figure 1 illustrates a balanced system with the largest possible group count $dfly(p = 2, a = 4, h = 2, g = 9)$. In this case, each group has $a = 4$ switches and $a \times h = 8$ global links with $a \times h/(g - 1) = 1$ global link connecting to each of other groups.

### 2.2 Cray Cascade Topology

The Cray Cascade architecture employs Dragonfly as its topology [2]. It has a well-defined structure for each group, but allows a variable number of groups to form a system.

Unlike $dfly(p, a, h, g)$, switches in a Cray Cascade group are not fully connected. Every group in Cascade is formed of a pair of cabinets. Each cabinet houses three chassis. Each chassis contains 16 blades. Each blade connects a single *Aries* router and four compute nodes. Each chassis backplane provides all-to-all connections among sixteen Aries routers. Each router is also connected to five other routers in the remaining five chassis within the same group. Each inter-chassis link is equivalent to three intra-chassis links in terms of bandwidth. Each Aries router has a total of 48 ports: 8 ports for local compute nodes, 15 ports connecting to 15 routers in the same chassis, 15 ports to 5 routers in the same slot but different chassis, and 10 ports to other groups. Figure 2 shows the interconnect topology of a single Cascade group. Logically, a cascade group consists of a $6 \times 16$ mesh with fully connected X and Y dimensions. Each pair in the same row is connected by one link while each pair in the same column is connected by three links.

In practice, the number of global links connecting a pair of groups in Cascade can be configured. For example, in the NERSC Edison supercomputer, there are 24 global links (spreading among multiple pairs of switches) connecting each pair of groups [13]. The details about how the global links are connected can be quite involved. The Cascade topology that we consider in this paper is a six-group system whose connectivity is directly read from the connectivity dump file for the first 6 groups of the Edison supercomputer [13].

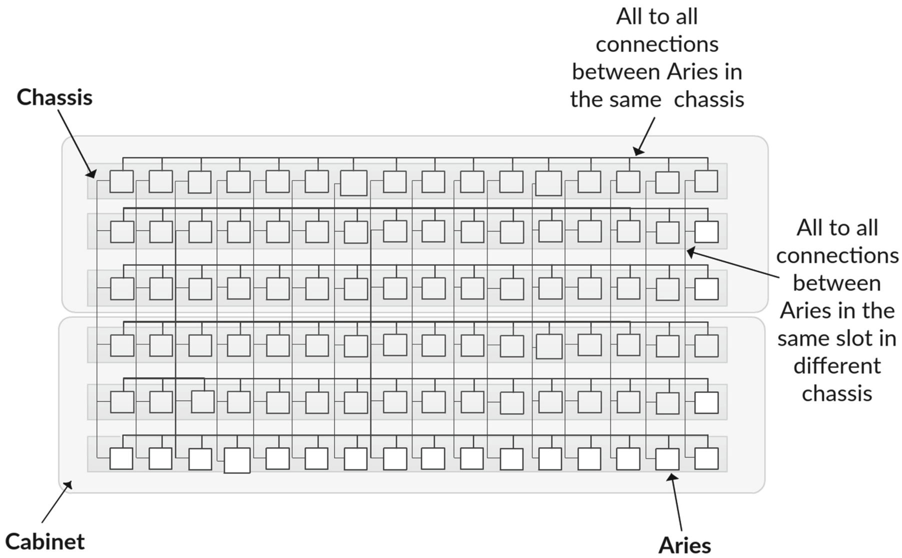



**Fig. 2.** Cray cascade intra-group topology

### 2.3 Routing in Dragonfly and UGAL

The following terminology will be used to describe routing in Dragonfly. Packets are routed from a *source compute node* to a *destination compute node*. The switch that the source compute node connects to is called the *source switch*. The switch that the destination compute node connects to is called the *destination switch*. The group that the source compute node is in is called the *source group*; the group that the destination compute node is in is called the *destination group*. We will describe routing for a generic Dragonfly topology. The routing can also be applied to the Cascade Dragonfly variation.

In a Dragonfly topology, packets are routed along either a *minimal* or a *non-minimal path*. The minimal path is the shortest path from the source compute node to the destination compute node that contains at most one global link. The thick segmented line in Fig. 3 shows a typical minimal path from $s$ to $d$, where the path takes one local hop in the source group from the source switch to the switch that has a global link to the destination group, then the global link to the destination group, and finally a local link at the destination group to the destination switch. Depending on the positions of the source and the destination, the minimal path may have fewer hops. In *dfly*$(p, a, h, g)$, two routers belonging to different groups may be connected through one of the $(a \times h)/(g-1)$ global links between the two groups. Thus, there are $(a \times h)/(g-1)$ minimal paths between such router pairs.

The Minimal routing (MIN) scheme routes packets only with minimal paths. It minimizes the resource usage and works well for traffic patterns where MIN

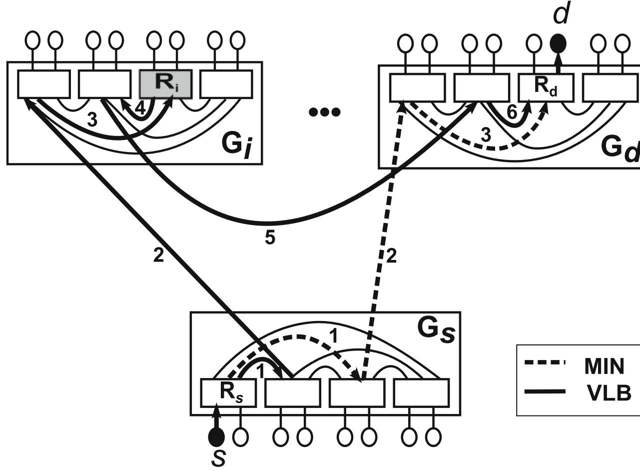



**Fig. 3.** MIN and VLB routing on Dragonfly

can evenly distribute the load such as the random uniform traffic. However, since the number of links between each pair of groups is typically small, for traffic patterns where many nodes in one group must communicate to many nodes in another group, the MIN routing will perform poorly since all of the traffic from one group to another must use the small number of links between the two groups. Such traffic patterns are considered adversarial.

To avoid congestion on global links for an adversarial traffic pattern, Valiant Load-balanced routing (VLB) [14] can be used to spread non-uniform traffic evenly over the set of available links. A VLB path can be considered as using MIN to find a path from the source to a randomly selected intermediate switch that is not in the source and destination groups, and then, from the intermediate switch to the destination. A VLB path is thus non-minimal. Figure 3 shows a 6-hop VLB path in solid thick lines. With a VLB route, a packet is first sent to an intermediate router ($R_i$ in this example) and then to the destination. We note that the initial works on Dragonfly routing [1,6] consider randomly selecting an intermediate group to obtain VLB paths. However, it is shown by Garcia et al. [15] that the randomly choosing a group leads to local link congestion at the intermediate group and instead, random selection of an intermediate switch is preferred. In $dfly(p, a, h, g)$, there are a total of $a \times (g-2)$ intermediate switches, $(a \times h)/(g-1)$ minimal paths from the source to each intermediate switch and again, $(a \times h)/(g-1)$ minimal paths from intermediate switch to destination. Therefore, the total number of VLB paths between two nodes of a $dfly(p, a, h, g)$ that are not in the same group is given by

$$\frac{a^3 \times h^2 \times (g-2)}{(g-1)^2} \tag{1}$$

The Universal Globally Adaptive Load-balanced routing (UGAL) selects among MIN and VLB paths for each packet based on the traffic condition. The traffic condition is inferred from the occupancy of packet queues of the network sensed at the source switch. For each packet, UGAL first randomly selects a small number of candidate MIN and VLB paths from all possible MIN and VLB paths for further consideration. In the original UGAL proposal and its Dragonfly adaptation, the number of MIN paths is 1 and the number of VLB paths is 1 [1,5]; in Cascade, 2 MIN paths and 2 VLB paths are chosen as candidates [2]. Then, UGAL selects a path from among the candidate paths for routing that would achieve the smallest packet delay. In contrast, UGAL-G assumes that the precise global network state information is available, and uses the total queue length on all links along the path to estimate the packet delay. Let $TQ_{MIN}$ be the smallest path queue length for all MIN paths considered, and $TQ_{VLB}$ be the smallest path queue length for all VLB paths considered. UGAL-G selects the MIN path if $TQ_{MIN} \leq TQ_{VLB}$, and the VLB path otherwise. Other UGAL-based schemes [2,6] rely on some practically measurable quantities such as credit-round-trip latency and piggybacked link-state information broadcast on source group to estimate the actual packet delay and approximate UGAL-G.

### 2.4 Maximum Concurrent Flow

Given a traffic pattern, there are various models to quantify the aggregate throughput performance. Among the throughput models, the maximum concurrent flow model is one of the commonly used models [8–11]. The Maximum Concurrent Flow (MCF) can informally be described as the maximum attainable throughput by all flows for a traffic pattern in a given network. In other words, MCF is the single largest rate that can be assigned to all flows without violating any capacity constraints. It is therefore the lower bound of the flow rates for all flows in the traffic pattern.

Without the routing constraint, the MCF rate for a given pattern on a given topology can be computed using the linear programming (LP) formulation given by Shahrokhi and Matula [8]. LP is an approach to minimize an objective function subject to a set of linear inequalities. Their linear-programming formulation considers all possible paths to route each flow. The models proposed in this work not only consider the specific UGAL routing on Dragonfly, which is constrained, but also how the paths are selected in UGAL. This allows us to develop more accurate and efficient models for UGAL on Dragonfly.

## 3 Performance Models for UGAL-G on Dragonfly

### 3.1 Notation

Let $A$ be a set and $|A|$ be the size of the set. Let a Dragonfly network be represented as a graph $G = (V, E)$, where $V$ is the set of nodes and $E$ is the set of links in the network. $V = PE \cup S$ contains two types of nodes. $PE$ is the set

of compute nodes; and $S$ is the set of switches. The nodes are numbered from 0 to $|V| - 1$. For each link $e \in E$, $C_e$ is the link capacity.

Let $s \in PE$ and $d \in PE$. A flow from $s$ to $d$ is denoted as $(s, d)$. A traffic pattern $F$ is a set of flows. The traffic in a flow is carried over a set of paths for the flow. Each path $p$ is represented as a set of links. For each flow, UGAL-G considers all MIN paths and all VLB paths. For a flow $(s, d)$, $P_{s,d}^{MIN,L}$ is the set of MIN paths with path length $L$; $P_{s,d}^{VLB,L}$ is the set of VLB paths with path length $L$; $P_{s,d}^{MIN}$ is the set of all MIN paths for the flow; $P_{s,d}^{VLB}$ is the set of all VLB paths and $P_{s,d}$ is the set of all considered paths. Clearly, $P_{s,d}^{MIN} = \cup_L P_{s,d}^{MIN,L}$; $P_{s,d}^{VLB} = \cup_L P_{s,d}^{VLB,L}$; and $P_{s,d} = P_{s,d}^{MIN} \cup P_{s,d}^{VLB}$. Let $e \in E$ be a link. If a path $p$ uses a link $e$, we say that $e \in p$. Given a set of paths $P$, $P(e)$ returns a subset of $P$ only containing paths that use link $e$. Table 1 summarizes the notations.

**Table 1.** Notation used in the models

| | |
|---|---|
| $G = (V, E)$ | the topology with node set $V$ and edge set $E$ |
| $C_e$, $e \in E$ | link capacity |
| $(s, d)$ | a flow from $s$ to $d$ |
| $P_{s,d}$ | the set of all MIN and VLB paths for $(s, d)$ |
| $P_{s,d}^{MIN}$ | the set of MIN paths for $(s, d)$ |
| $P_{s,d}^{MIN,L}$ | the set of MIN paths of length $L$ for $(s, d)$ |
| $P_{s,d}^{VLB}$ | the set of VLB paths for $(s, d)$ |
| $P_{s,d}^{VLB,L}$ | the set of VLB paths of length $L$ for $(s, d)$ |
| $P(e)$ | $\{p \mid e \in p \text{ and } p \in P\}$ |

### 3.2 Performance Models

We use linear programming (LP) to model UGAL-G performance as an optimization problem. For accuracy, our models consider the following UGAL-G features.

- **Feature 1**: UGAL-G considers all MIN and VLB paths.
- **Feature 2**: UGAL-G randomly selects a small number of MIN and VLB paths as candidate paths for each packet.
- **Feature 3**: UGAL-G implicitly differentiates paths of different lengths. UGAL-G selects paths based on the path latency. As a result, it biases towards using shorter paths: if the queue length is the same for all links, shorter paths will have smaller aggregate queue length and are more likely to be selected by UGAL-G.

The challenge to develop accurate performance models is to capture the dominating factors in the UGAL-G routing process. UGAL-G uses an identical process to select between MIN and VLB paths. Thus, the spectrum of UGAL-G's control over MIN and VLB paths is the same. Next, we will use VLB paths to describe the potential control that UGAL-G has on paths. Consider the spectrum of UGAL-G's control over VLB paths. At one end, since UGAL-G considers all VLB paths (Feature 1), if it may have a fine-grain control at the path level, it could allocate rates for individual paths so as to maximize the aggregate throughput for a pattern. This level of control will be referred to as **individual** control. On the other end, UGAL-G randomly selects a small number of VLB paths as candidate paths for each packet (Feature 2). If the random selection dominates the performance, the routing essentially treats all VLB paths the same as a group and uniformly distribute the load to each of the paths. This level of control will be referred to as **all random** control. In general, the level of control falls in between the two extremes. Feature 3 states that UGAL-G differentiates paths of different lengths. This gives another potential level of control in between the two extremes, which we call **path-length-based random** control. In this control, the VLB paths are grouped based on their lengths. The routing scheme may allocate rates differently for different groups, but will treat paths in the same group the same. Further refinement of the levels of control is possible. However, it will be shown later that the combination of these three levels of control already yields accurate modeling.

The level of control that UGAL-G has would depend on the number of MIN and VLB paths, which is determined by the Dragonfly topology. When the number of MIN (VLB) paths is small, each MIN (VLB) path is likely considered as a candidate path for each packet; and UGAL-G can have a high level of control over the rate allocation over the MIN (VLB) paths. On the other hand, when the number of MIN (VLB) paths is very large, the chance for each MIN (VLB) path to be selected as the candidate path by UGAL-G is very small. As a result, UGAL-G will have a low level of control of the rate allocation over such paths. In between these two extremes, path-length-based random control may be more appropriate.

**Table 2.** Summary of models (Model No. 3 is a robust and efficient model for different topologies)

| Model | MIN | VLB |
|---|---|---|
| No. 0 | individual | individual |
| No. 1 | individual | path-length-based random |
| No. 2 | individual | all random |
| No. 3 | path-length-based random | path-length-based random |
| No. 4 | path-length-based random | all random |
| No. 5 | all random | all random |

Given a Dragonfly topology, it is unclear which level of control UGAL-G has for the MIN and VLB paths. In general, the number of MIN paths is significantly smaller than the number of VLB paths. As such, UGAL-G will have more control over MIN paths than over VLB paths. To find a robust model that is both accurate and efficient, we develop a set of six models that applies each of the three levels of control on the two types of paths (MIN and VLB) with the assumption that UGAL-G will have an equal or higher level of control over MIN paths than over VLB paths. The models are summarized in Table 2. Our experiments indicate that Model No. 3 with path-length-based random control for both MIN and VLB paths is a robust and efficient model across many variations of Dragonfly including the Cascade topology, achieving accurate modeling results and low modeling complexity.

**Model No. 0 (the upper bound, individual control on both MIN and VLB paths)**

For each flow, UGAL-G considers all MIN and VLB paths. Model No. 0 assumes that UGAL-G has individual control over both MIN and VLB paths so that it can allocate the rate for each path to maximize the throughput. To model the individual control over each MIN and VLB path, each MIN or VLB path can have a different rate, which is represented as one variable in the LP formulation. Our linear programming formulation uses the edge-path formulation assuming that each path considered by UGAL-G can be assigned a different rate to maximize the MCF rate.

The LP formulation is shown in Fig. 4. In this model, one variable $x^p_{s,d}$ is assigned to each path $p$ considered by UGAL-G for a flow $(s, d)$ in the pattern. The variable $x^p_{s,d}$ represents the rate allocated for the path. Hence, for flow $(s, d)$, the sum of the rates allocated to all of its paths, $\sum_{p\in P_{s,d}} x^p_{s,d}$, is the flow rate. The variable $\alpha$ is the MCF rate for the pattern. By MCF definition, the rates for all flows must be no less than the MCF rate. The constraints in (1) ensure that the rates for all flows are no less than the MCF rate. Constraints (2) are link capacity constraints that state that for each link, the total rates for all paths that use the link, $\sum_{e\in p, p\in P_{s,d}, (s,d)\in F} x^p_{s,d}$, do not exceed the link capacity.

1 Maximize $\alpha$
2 Subject to:
3 $$\alpha - \sum_{p\in P_{s,d}} x^p_{s,d} \leq 0, \ \forall (s,d) \in F \quad (1)$$
4 $$\sum_{e\in p, p\in P_{s,d}, (s,d)\in F} x^p_{s,d} \leq C_e, \forall e \in E \quad (2)$$

**Fig. 4.** Model No. 0: the upper bound MCF rate for all UGAL-based schemes (individual control over MIN paths and individual control over VLB paths)

The formulation in Fig. 4 assumes that the rate for each path can be tuned to maximize the MCF throughput, which provides an upper bound for all UGAL-based algorithms. This formulation, however, has two issues. First, solving the

problem on reasonably sized networks becomes computationally infeasible due to the use of a large number of variables. In practical Dragonfly networks, the number of minimal paths is usually not very large, while the number of VLB paths can easily approach tens of thousands to millions. See Table 3 for Dragonfly examples with the numbers of MIN and VLB paths. This formulation can easily introduce more than one million variables for some topology. Solving LP problems of such sizes is computationally infeasible with today's technology. The second issue is that this formulation does not consider the inner working of UGAL-G such as Features 2 and 3. Thus, it may not yield accurate estimation results for UGAL-G.

**Model No. 1 (individual control on MIN paths and path-length-based random control on VLB paths)**

Model No. 0 would yield an accurate modeling result only if UGAL-G were capable of tuning the rate for each available MIN and VLB path in the most effective manner. In the Dragonfly topology, the number of MIN paths for each flow is usually small while the number of VLB paths can be much larger. For example, in *dfly*$(3, 6, 3, 10)$, the number of VLB paths between two nodes that are not in the same group is 192 as calculated from Formula 1, while the number of MIN paths for each flow is 2. In such a situation, considering a small number of (1 or 2) VLB paths for each packet is not likely to result in effective use of VLB paths while the routing may have individual control over MIN paths since the MIN path is considered for every packet. Model No. 1 assumes individual control over MIN paths and path-length-based random control over VLB paths and targets Dragonfly networks with a small number of MIN paths and a reasonably large number of VLB paths per flow.

The LP formulation for Model No. 1 is shown in Fig. 5. In this model, for each flow $(s, d)$, a variable $x^p_{s,d}$ is assigned to each MIN path $p \in P^{MIN}_{s,d}$. In addition, another variable $x^{VLB,L}_{s,d}$ is assigned for all VLB paths of length $L$ ($P^{VLB,L}_{s,d} \neq \emptyset$) of a given flow $(s, d)$: each of the VLB paths of length $L$ will have the same rate, $x^{VLB,L}_{s,d}$, while VLB paths of different lengths may have different rates. The LP formulation of Model No. 1 is basically the same as that of Model No. 0 except that all VLB paths of the same length $L$ for each flow is assumed to have the same rate. $\sum_{p \in P^{MIN}_{s,d}} x^p_{s,d} + \sum_{P^{VLB,L}_{s,d} \neq \emptyset} |P^{VLB,L}_{s,d}| \times x^{VLB,L}_{s,d}$ is the rate allocated for flow $(s, d)$; and Constraints (1) ensure that the rates for all flows are no less than the MCF rate. $\sum_{p \in P^{MIN}_{s,d}(e),(s,d) \in F} x^p_{s,d} + \sum_{P^{VLB,L}_{s,d}(e) \neq \emptyset,(s,d) \in F} |P^{VLB,L}_{s,d}(e)| \times x^{VLB,L}_{s,d}$ is the total rate allocated over link $e$; and Constraints (2) are link capacity constraints that ensure that the rate allocated over each link is no more than its capacity.

The Model No. 1 in Fig. 5 will be accurate when the random selection of VLB paths (Feature 2) and the path length preferences (Feature 3) have impacts on the throughput performance. Since VLB paths have similar path lengths in Dragonfly, Model No. 1 only needs a small number of variables for VLB paths, which significantly reduces the number of variables over Model No. 0. For

1 Maximize $\alpha$
2 Subject to:
3 $$\alpha - (\sum_{p \in P^{MIN}_{s,d}} x^p_{s,d} + \sum_{P^{VLB,L}_{s,d} \neq \emptyset} |P^{VLB,L}_{s,d}| \times x^{VLB,L}_{s,d}) \leq 0,\ \forall (s,d) \in F \quad (1)$$
4 $$\sum_{p \in P^{MIN}_{s,d}(e),(s,d) \in F} x^p_{s,d} + \sum_{P^{VLB,L}_{s,d}(e) \neq \emptyset,(s,d) \in F} |P^{VLB,L}_{s,d}(e)| \times x^{VLB,L}_{s,d} \leq C_e, \forall e \in E \quad (2)$$

**Fig. 5.** Model No. 1: Maximize the MCF rate with the assumption that VLB paths of the same length for a flow have the same rate (individual control over MIN paths and path-length-based random control over VLB paths)

example, the longest VLB path in $dfly(p,a,h,g)$ is 6 hops, as shown in Fig. 3. Therefore, there could be at most 6 different path lengths for all VLB paths and thus, only up to 6 variables corresponding to VLB routing is required per flow in the model LP formulation. This reduction in the number of variables enables Model 1 to be used to solve much larger problems in much larger systems.

### Model No. 2 (individual control on MIN paths and all random control on VLB paths)

Model No. 1 considers the three features of UGAL-G: (1) the routing considers all MIN and VLB paths, (2) the large number of VLB paths is randomly selected for consideration for each packet, and (3) UGAL-G inherently differentiates between paths of different lengths. When the number of VLB paths is very large, the random selection of VLB paths to be considered for each packet may be the dominating factor. In this case, UGAL-G may only have the all random control over VLB paths. Model No. 2 that assumes individual control of MIN paths and all random control of VLB paths is designed for such cases.

The LP formulation for Model No. 2 is shown in Fig. 6. In this model, for each flow $(s,d)$, a variable $x^p_{s,d}$ is assigned to each MIN path $p \in P^{MIN}_{s,d}$. In addition, another variable $x^{VLB}_{s,d}$ is assigned for all VLB paths, that is, each of the VLB paths is assumed to have the same rate $x^{VLB}_{s,d}$. Model No. 2 is basically the same as Model No. 1 except that all VLB paths for each flow are assumed to have the same rate. Constraints (1) ensure that the rates for all flows are no less than the MCF rate. $\sum_{p \in P^{MIN}_{s,d}(e),(s,d) \in F} x^p_{s,d} + \sum_{P^{VLB}_{s,d}(e) \neq \emptyset,(s,d) \in F} |P^{VLB}_{s,d}(e)| \times x^{VLB}_{s,d}$ is the

1 Maximize $\alpha$
2 Subject to:
3 $$\alpha - (\sum_{p \in P^{MIN}_{s,d}} x^p_{s,d} + |P^{VLB}_{s,d}| \times x^{VLB}_{s,d}) \leq 0,\ \forall (s,d) \in F \quad (1)$$
4 $$\sum_{p \in P^{MIN}_{s,d}(e),(s,d) \in F} x^p_{s,d} + \sum_{P^{VLB}_{s,d}(e) \neq \emptyset,(s,d) \in F} |P^{VLB}_{s,d}(e)| \times x^{VLB}_{s,d} \leq C_e, \forall e \in E \quad (2)$$

**Fig. 6.** Model No. 2: Maximize the MCF rate with the assumption that all VLB paths for a flow have the same rate (individual control for MIN paths and all random control for VLB paths)

total rate allocated over link $e$ which must not exceed the link capacity. Such capacity constraints are summarized in Constraints (2).

The Model No. 2 in Fig. 6 will be accurate when the random selection of VLB paths dominates the performance. It further reduces the number of variables for each flow in comparison to Model No. 1.

#### Model No. 3 (path-length-based random control on MIN paths and path-length-based random control on VLB paths)

Although the number of VLB paths is always significantly larger than the number of MIN paths for each flow in a Dragonfly topology, some Dragonfly topologies can have a significant number of MIN paths. Variants of Dragonfly such as the Cascade topology that do not have a fully connected intra-group network and have high number of global links between all group pairs, fall into this category. For such topologies, UGAL-G may not have the individual control over each MIN path. Model No. 3 assumes that the control over MIN paths as well as VLB paths is path-length-based random.

The LP formulation for Model No. 3 is shown in Fig. 7. In this model, for each flow $(s, d)$, a variable $x_{s,d}^{MIN,L}$ is assigned to each group of MIN paths of length $L$ ($P_{s,d}^{MIN,L} \neq \emptyset$). For VLB paths, a variable $x_{s,d}^{VLB,L}$ is assigned for each group of VLB paths of length $L$ ($P_{s,d}^{VLB,L} \neq \emptyset$). $\sum_{P_{s,d}^{MIN,L} \neq \emptyset} |P_{s,d}^{MIN,L}| \times x_{s,d}^{MIN,L} + \sum_{P_{s,d}^{VLB,L} \neq \emptyset} |P_{s,d}^{VLB,L}| \times x_{s,d}^{VLB,L}$ is the rate allocated for flow $(s, d)$. Constraints (1) describe the MCF rate constraints. $\sum_{P_{s,d}^{MIN,L}(e) \neq \emptyset, (s,d) \in F} |P_{s,d}^{MIN,L}(e)| \times x_{s,d}^{MIN,L} + \sum_{P_{s,d}^{VLB,L}(e) \neq \emptyset, (s,d) \in F} |P_{s,d}^{VLB,L}(e)| \times x_{s,d}^{VLB,L}$ is the total rate allocated over link $e$; and the same expression is used in Constraints (2) to summarize capacity constraints on all links.

1 Maximize $\alpha$
2 Subject to:
3 $$\alpha - (\sum_{P_{s,d}^{MIN,L} \neq \emptyset} |P_{s,d}^{MIN,L}| \times x_{s,d}^{MIN,L} + \sum_{P_{s,d}^{VLB,L} \neq \emptyset} |P_{s,d}^{VLB,L}| \times x_{s,d}^{VLB,L}) \leq 0, \forall (s,d) \in F \quad (1)$$
4 $$\sum_{P_{s,d}^{MIN,L}(e) \neq \emptyset, (s,d) \in F} |P_{s,d}^{MIN,L}(e)| \times x_{s,d}^{MIN,L} + \sum_{P_{s,d}^{VLB,L}(e) \neq \emptyset, (s,d) \in F} |P_{s,d}^{VLB,L}(e)| \times x_{s,d}^{VLB,L} \leq C_e, \forall e \in E \quad (2)$$

**Fig. 7.** Model No. 3: Maximize the MCF rate with the assumption of path-length based control for both MIN and VLB paths

#### Model No. 4 and Model No. 5

Model No. 4 assumes path-length-based random control on MIN paths and all random control on VLB paths. Model No. 5 assumes all random control on both VLB and MIN paths. These two models uses less variables than all of the earlier models. Their LP formulations are straight-forward extensions of those for Models No. 1, 2, and 3, and are omitted.

## 4 Model Validation

We implemented the six models for the general Dragonfly topology as well as for the Cascade topology. Each implemented model takes in a topology, a routing scheme and a traffic pattern as inputs and generates an LP formulation file. The LP formulation is then fed into IBM's CPLEX optimizer [16] to find the maximum MCF rate for each of our experiment instances.

We have also extended Booksim [12] to support UGAL-G for $dfly(p, a, h, g)$ and the Cascade topology. Then, simulation results on the same network configurations are obtained to validate the models. We assume single-flit packets and a 2.5x speedup for router crossbar over network links. The latency of each network link is set to 10 cycles. To ensure deadlock-free routing, we allocate three virtual channels for the Dragonfly topology in the same way as described in [1], and ten virtual channels for the Cascade topology. The buffer size of each virtual channel is set to 256 flits. For each data point, the network is warmed-up for 40,000 cycles and network statistics are collected for another 10,000 cycles. In Booksim, all processing nodes inject traffic to the network at a same *injection rate*. During each simulation run, We gradually increment the injection rate until the packet queues across the network becomes saturated. Once the network is saturated, we record the corresponding injection rate as the maximum concurrent throughput of that run.

**Table 3.** Topologies used in the validation

| Topology | # of switches | # of PEs | # of MIN | # of VLB |
|---|---|---|---|---|
| $dfly(2,4,2,9)$ | 36 | 72 | 1 | 28 |
| $dfly(3,6,3,19)$ | 114 | 342 | 1 | 102 |
| $dfly(4,8,4,33)$ | 264 | 1,056 | 1 | 248 |
| $dfly(5,10,5,51)$ | 510 | 2,550 | 1 | 490 |
| $dfly(5,10,5,26)$ | 260 | 1,300 | 2 | 960 |
| $dfly(5,10,5,11)$ | 110 | 550 | 5 | 2250 |
| $dfly(5,10,5,6)$ | 60 | 300 | 10 | 4000 |
| Cascade | 576 | 2,304 | 96 | 3,538,944 |

The topologies considered are summarized in Table 3. Two types of topologies are used: the load-balanced Dragonfly with fully connected intra-group topology described in $dfly(p, a, h, g)$ denotation, and the 6-group Cascade topology. The difference between these two topologies is in the number of MIN and VLB paths that are available. The number of MIN and VLB paths in $dfly(p, a, h, g)$ is $(a \times h)/(g-1)$ and $(a^3 \times h^2 \times (g-2))/(g-1)^2$ respectively, as shown in Sect. 2. In the Cascade topology, a packet can go in either X or Y dimension first within each group and there are 24 global links between each group pair. Hence, the number

of MIN paths between two nodes in different groups can be up to $2 \times 24 \times 2 = 96$. The number of VLB paths in Cascade is much larger. Using $4 \times 96 = 384$ potential intermediate switches, the number of VLB paths for each flow can be up-to $96 \times 96 \times 384 = 3,538,944$. As discussed earlier, the number of MIN and VLB paths affects how UGAL-G controls the paths.

In the experiments on $dfly(p, a, h, g)$, one MIN path and one VLB path are randomly chosen as candidate paths for each packet, same as in the original UGAL proposal [6]. On the Cascade topology, we consider 2 MIN and 2 VLB candidate paths in consistency with the current Cascade routing scheme [2].

The results for two types of traffic patterns are reported, the random permutation patterns where each node sends to and receives from at most one other destination and source respectively, and the random shift pattern where compute node $i$ sends to compute node $(i + x) \bmod |PE|$ where $x$ is a random number. Results for other patterns yield similar trends.

The general observations in the experiments include the following: individual control in general overestimates the throughput; all random control in general underestimates the throughput; and the path-length-based random control gives good estimation for a wide range of Dragonfly variations. In particular, Model No. 3 that assumes path-length-based random control for both MIN and VLB paths, which has a low complexity with a small number of variables for each flow, achieves good prediction for a wide range of Dragonfly topologies (within 10% of prediction errors in all cases in our study).

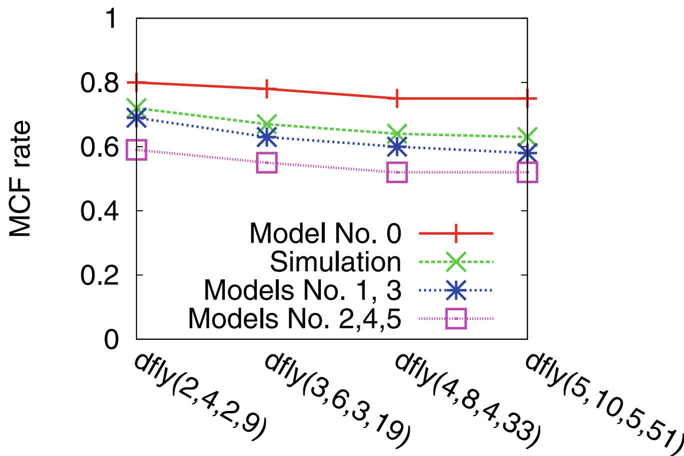



**Fig. 8.** The modeling and simulation results for random permutation patterns on $dfly(p, a, h, a \times h + 1)$

Figure 8 shows the average modeling and simulation results for five random permutation patterns on maximum size $dfly(p, a, h, a \times h + 1)$ networks of different sizes. For these topologies, since the number of MIN paths for each flow

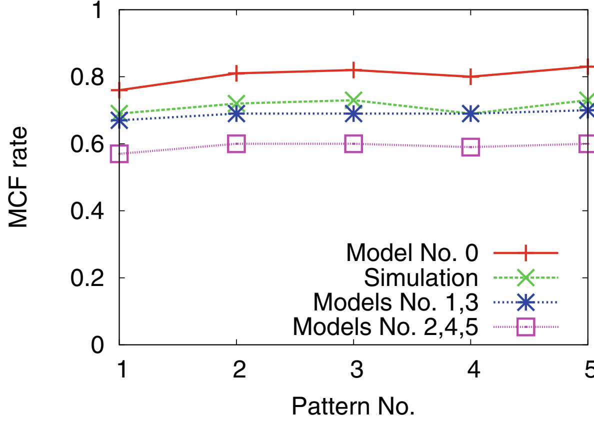



**Fig. 9.** The modeling and simulation results for individual random permutation patterns on *dfly*$(2, 4, 2, 9)$

is only 1, Model No. 1 is equivalent to Model No. 3, and Models No. 2, 4, and 5 are equivalent. As can be seen from the figure, the throughput with UGAL-G across all topologies is significantly worse than the throughput predicted by Model No. 0. This indicates that for these topologies, UGAL-G cannot fully control the MIN and VLB paths to maximize its throughput. The figure also shows that the throughput with UGAL-G is significantly better than that predicted with Model No. 2. This indicates that UGAL-G has better control than all random over VLB paths. Across all topologies, the throughput predicted by Models No. 1 and No. 3 closely matches the simulation with the prediction errors ranging from 4.3% to 8.6%. Figure 9 shows prediction and simulation results for each individual random permutation on *dfly*$(2, 4, 2, 9)$. As can be seen from the figure, the trend for the prediction with each model is exactly the same as that in Fig. 8. Results on other similar *dfly*$(p, a, h, g)$ instances are similar.

Figure 10 shows the average modeling and simulation results for five random permutation patterns on Dragonfly topologies with the same group $group(5, 10, 5)$, but different numbers of groups: *dfly*$(5, 10, 5, 6)$ with 6 groups, *dfly*$(5, 10, 5, 11)$ with 11 groups, and so forth. These topologies have the same structure with different numbers of global links connecting each pair of groups, which affects the number of MIN and VLB paths as shown in Table 3. Results for Model No. 4, which are in-between the results for Models No. 3 and No. 5, are omitted to make the figure less dense. From the figure, it is evident that individual control overestimates the throughput when the number of paths in a group (MIN or VLB) is sufficiently large, while the all random control underestimates the throughput. The overall throughput estimation is a combination of the estimation of VLB paths and MIN paths. Thus, Model No. 0 overestimates the throughput for both VLB and MIN paths, resulting in consistent over-estimation of throughput for all cases. Similarly, Model No. 5 consistently

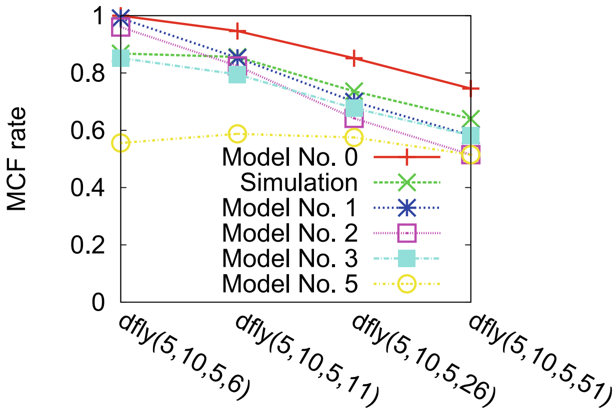



**Fig. 10.** The modeling and simulation results for random permutation patterns on different number of $group(5, 10, 5)$ groups

underestimates the throughput for all cases. Model No. 3 consistently tracks the throughput obtained from simulation for different topologies. Notice that the overall throughput estimation is the combination of the estimation for MIN and VLB paths: over-estimating or under-estimating either MIN or VLB performance can sometimes dominate the overall prediction, resulting in prediction errors. For example, for $dfly(5, 10, 5, 6)$ with 10 MIN paths per flow, Models No. 1 and No. 2 both overestimate the throughput for MIN by assuming individual control, resulting large overall prediction errors.

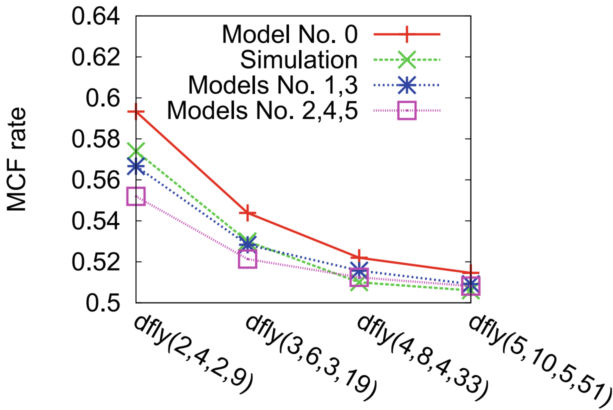



**Fig. 11.** The modeling and simulation results for random shift patterns on $dfly(p, a, h, a \times h + 1)$

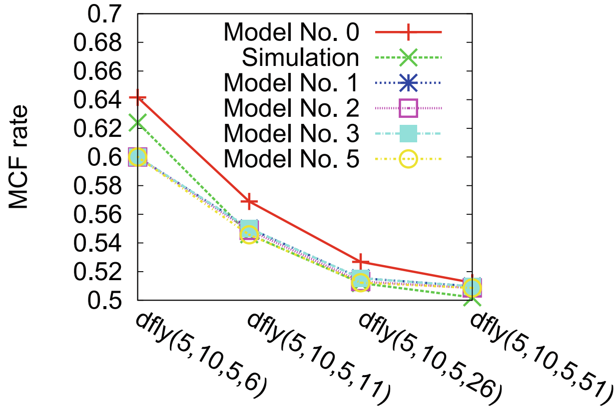



**Fig. 12.** The modeling and simulation results for random shift patterns on different number of $group(5, 10, 5)$ groups

Figure 11 shows the average modeling and simulation results for five random shift patterns on the largest Dragonfly of different sizes $dfly(p, a, h, a \times h + 1)$. This is one of the adversarial traffic patterns for Dragonfly. From the rate allocation perspective, however, it is clear what needs to happen to achieve high performance: use the VLB paths uniformly. As can be seen from the figure, even with the full control of the rate allocation for the patterns, the throughput is not much higher than treating all VLB paths the same. For this pattern, Model No. 0 only slightly overestimates the throughput while Models No. 2, 4, 5 only slightly underestimates the throughput. Models No. 1 and No. 3, nonetheless, produces the most accurate prediction. Figure 12 compares modeling and simulation results on Dragonfly topologies with the same group $group(5, 10, 5)$, but different number of groups. Very similar results to those in Fig. 11 are observed.

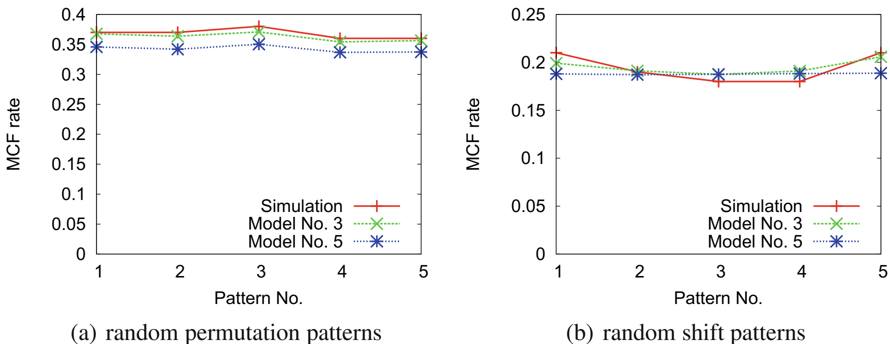



**Fig. 13.** The modeling and simulation results on the 6-group Cascade topology

Figure 13(a) shows modeling and simulation results for five different random permutation patterns on the 6-group Cascade topology. We recall that the LP formulation given by Model No. 0 requires a unique variable for each unique path. Due to the large number of VLB and MIN paths in this topology, calculating the performance upper bound of UGAL-G in Cascade would then require solving LP with several billions of variables which is not computationally feasible. We, therefore, omit considering Model No. 0 on the Cascade system and compare UGAL-G performance with the remaining five models. In the experiments, Models No. 1 and No. 3 result in almost the same values while Models No. 2, No. 4, and No. 5 yield almost the same value. We only show the results for Models No. 3 and No. 5 in the figure for clarity. For this topology, the number of MIN and VLB paths are both very large. Models No. 1 and No. 3 differ in how the MIN paths are controlled: Model No. 1 assumes individual control of MIN paths while Model No. 3 assumes path-length based control. The fact that Models No. 1 and No. 3 yield similar results for the random permutation patterns indicates that fine-grain control of the MIN paths does not yield better throughput performance for this topology, which is likely due to the large number of links between each pair of groups. Models No. 2, No. 4 and No. 5 also only differ in how the MIN paths are controlled. Thus, similar logic applies. It is evident from Fig. 13(a) that Model No. 3 and Model No. 1 predict the throughput performance on this topology very accurately. The prediction errors for the five random permutation patterns range from 0.0% to 2.6%. In fact, even Model No. 5 (as well as Models No. 2 and No. 4) has good prediction accuracy with errors up-to 7.0%. These results confirm that when the number of MIN and VLB paths are large, the control of UGAL-G over the MIN and VLB paths is group-based. Figure 13(b) shows modeling and simulation results for five random shift patterns on the same Cascade topology. The trend is very similar: UGAL-G performance is almost perfectly approximated by Model No. 3 and can be reasonably approximated with Model No. 5.

Other patterns and other Dragonfly topologies have also been studied. The results have the similar trend: individual control consistently overestimates the performance although the level of over-estimation differs based on the topology; all random control consistently underestimates the performance; and the path-length-based random control, which takes the three distinguished features of UGAL-G described in Sect. 3 into consideration, consistently tracks the performance across a wide range of topologies. These results have two indications. First, UGAL-G has group-based control when the number of MIN and VLB paths is sufficiently large. Second, path-length-based control for both MIN and VLB paths (Model No. 3) is sufficient to model UGAL-G accurately on different Dragonfly topologies. As a result, the LP formulation only needs a small number of variables (at most 6 for $dfly(p, a, h, g)$ and 12 for Cray Cascade) to model each flow; and the models can be used to obtain throughput performance for large systems with tens of thousands of flows.

## 5 Related Work

Since the Dragonfly network was first introduced, it has been clear that a globally adaptive routing scheme is needed. In the seminal work by Kim et al. [1], the authors propose selecting a random intermediate group to route non-minimally in order to load-balance adversarial traffic patterns over global channels. Jiang proposes several adaptive routing heuristics that approximate UGAL-G [6]. Improvements over the original UGAL-based scheme have been developed. Garcia et al. [15] are the first to address local congestion inside Dragonfly groups and proposed allowing non-minimal routing on both intra- and inter-group communication in their OFAR routing scheme. OFAR-CM [17] proposes throttling packet injection at local nodes as well as routing through an escape subnetwork to mitigate congestion on OFAR routing at the cost of additional hops. Opportunistic Local Misrouting (OLM) [18] allows non-minimal routing on both local and global levels of the Dragonfly hierarchy and the routing decision may be updated at any hop. Improvements for load estimation with UGAL-based routing scheme have also been developed [19,20]. Existing research on UGAL-based routing mainly focuses on improving the effectiveness of the routing scheme. Jain et al. [21] provide an iterative model to predict the link utilization and thus, estimate throughput of UGAL-G routing on large-scale Dragonfly networks. Their model uses a bandwidth approximation scheme assuming all flows have a fair of bandwidth on each link, which is known to underestimate throughput with a multi-path routing. Our work is different from the existing research in that we develop efficient throughput performance models using linear programming that give more insights about rate allocation control of UGAL on Dragonfly designs.

## 6 Conclusion

We develop a set of throughput models for UGAL-G on the Dragonfly topology based on the level of control that UGAL-G has on the MIN and VLB paths, and identify a robust model that is both accurate and efficient for a large number of Dragonfly variations. The model not only provides a mechanism to predict the aggregate throughput performance for large scale Dragonfly networks, but also reveals (1) that even with the precise global information, UGAL-G is unable to achieve a fine-grain control over individual paths that are available, and (2) that UGAL-G in general allocates rates to groups of paths.

The Dragonfly topology has a large number of variants. The level of control that UGAL has over its paths is largely determined by the number of MIN and VLB paths, which in turn is decided by the topology. This work in general indicates that higher level of control can be achieved by UGAL-G when the number of MIN (VLB) paths is small, and that the level of control decreases as the number of MIN (VLB) paths increases. More research is necessary to determine the relationship between the number of available MIN and VLB paths and the level of control that UGAL has over the paths.

## References

1. Kim, J., Dally, W.J., Scott, S., Abts, D.: Technology-driven, highly-scalable dragonfly topology. In: ACM SIGARCH Computer Architecture News, vol. 36, pp. 77–88. IEEE Computer Society (2008)
2. Faanes, G., Bataineh, A., Roweth, D., Froese, E., Alverson, B., Johnson, T., Kopnick, J., Higgins, M., Reinhard, J., et al.: Cray cascade: a scalable HPC system based on a dragonfly network. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, p. 103. IEEE Computer Society Press (2012)
3. NERSC Cori supercomputer. http://www.nersc.gov/users/computational-systems/cori/
4. Archer, B.J., Vigil, M.: The trinity system. In: Nuclear Explosive Code Development Conference (NECDC), Los Alamos, New Mexico, 20–24 October 2014. Also appears as Los Alamos Technical Report LA-UR-15-20221
5. Singh, A.: Load-balanced routing. In: Interconnection Networks. Ph.D. thesis, Stanford University (2005)
6. Jiang, N., Kim, J., Dally, W.J.: Indirect adaptive routing on large scale interconnection networks. SIGARCH Comput. Archit. News **37**(3), 220–231 (2009)
7. Open networking foundation. Sdn architecture. White Paper, ONF TR-502, June 2014. https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_SDN_ARCH_1.0_06062014.pdf
8. Shahrokhi, F., Matula, D.W.: The maximum concurrent flow problem. J. ACM **37**(2), 318–334 (1990)
9. Jyothi, S.A., Singla, A., Godfrey, P.B., Kolla, A.: Measuring and understanding throughput of network topologies. In: The International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2016), November 2016
10. Singla, A., Godfrey, P.B., Kolla, A.: High throughput data center topology design. In: 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI), April 2014
11. Faizian, P., Mollah, M.A., Yuan, X., Pakin, S., Lang, M.: Random regular graph and generalized De Bruijn graph with k-shortest path routing. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 103–112, May 2016
12. Jiang, N., Balfour, J., Becker, D.U., Towles, B., Dally, W.J., Michelogiannakis, G., Kim, J.: A detailed and flexible cycle-accurate network-on-chip simulator. In: 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 86–96, April 2013
13. NERSC Edison supercomputer. http://www.nersc.gov/users/computational-systems/edison/
14. Valiant, L.G.: A scheme for fast parallel communication. SIAM J. Comput. **11**(2), 350–361 (1982)
15. Garcia, M., Vallejo, E., Beivide, R., Odriozola, M., Camarero, C., Valero, M., Rodríguez, G., Labarta, J., Minkenberg, C.: On-the-fly adaptive routing in high-radix hierarchical networks. In: 2012 41st International Conference on Parallel Processing (ICPP), pp. 279–288, September 2012
16. IBM CPLEX optimizer. https://www.ibm.com/us-en/marketplace/ibm-ilog-cplex/

17. Garcia, M., Vallejo, E., Beivide, R., Valero, M., Rodríguez, G.: OFAR-CM: efficient dragonfly networks with simple congestion management. In: 2013 IEEE 21st Annual Symposium on High-Performance Interconnects (HOTI), pp. 55–62, August 2013
18. Garcia, M., Vallejo, E., Beivide, R., Odriozola, M., Valero, M.: Efficient routing mechanisms for dragonfly networks. In: 2013 42nd International Conference on Parallel Processing (ICPP), pp. 582–592, October 2013
19. Won, J., Kim, G., Kim, J., Jiang, T., Parker, M., Scott, S.: Overcoming far-end congestion in large-scale networks. In: 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), pp. 415–427, February 2015
20. Fuentes, P., Vallejo, E., Garcia, M., Beivide, R., Rodríguez, G., Minkenberg, C., Valero, M.: Contention-based nonminimal adaptive routing in high-radix networks. In: 2015 IEEE International Conference on Parallel and Distributed Processing Symposium (IPDPS), pp. 103–112, May 2015
21. Jain, N., Bhatele, A., Ni, X., Wright, N.J., Kale, L.V.: Maximizing throughput on a dragonfly network. In: SC14: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 336–347, November 2014

# Resilient N-Body Tree Computations with Algorithm-Based Focused Recovery: Model and Performance Analysis

Aurélien Cavelan[1,2(✉)], Aiman Fang[3], Andrew A. Chien[3,4], and Yves Robert[2,5]

[1] University of Basel, Basel, Switzerland
aurelien.cavelan@unibas.ch
[2] Laboratoire LIP, ENS Lyon and Inria, Lyon, France
yves.robert@inria.fr
[3] University of Chicago, Chicago, USA
{aimanf,achien}@cs.uchicago.edu
[4] Argonne National Laboratory, Lemont, USA
[5] University of Tennessee Knoxville, Knoxville, USA

**Abstract.** This paper presents a model and performance study for Algorithm-Based Focused Recovery (ABFR) applied to N-body computations, subject to latent errors. We make a detailed comparison with the classical Checkpoint/Restart (CR) approach. While the model applies to general frameworks, the performance study is limited to perfect binary trees, due to the inherent difficulty of the analysis. With ABFR, the crucial parameter is the detection interval, which bounds the error latency. We show that the detection interval has a dramatic impact on the overhead, and that optimally choosing its value leads to significant gains over the CR approach.

## 1 Introduction

Future large-scale systems are projected to have higher error rates, with MTBFs (Mean Time Between Failures) as low as 20 min [1]. We focus on latent errors, that are not detected immediately after their occurrence. Such errors escape simple system level detection and can only be exposed by sophisticated application checks [2,3]. We use the term "detection latency" to denote the time from error occurrence to detection. Such latency may be thousands ($10^3$) to billions ($10^9$) cycles, corrupting a range of computational data. Without support to detect and recover from latent errors, applications will suffer silent data corruption, producing invalid scientific results.

In previous work [4], we proposed a new approach, Application-Based Focused Recovery (ABFR), that exploits application data flow and intermediate states to focus recovery on an accurate estimate of potentially corrupted data. Our study on stencil computations demonstrated ABFR reduces recovery cost by up to 400×. This paper investigates the use of ABFR for N-body

© Springer International Publishing AG 2018
S. Jarvis et al. (Eds.): PMBS 2017, LNCS 10724, pp. 158–178, 2018.
https://doi.org/10.1007/978-3-319-72971-8_8

computations in the presence of latent errors. N-body computations are much more challenging, as information is exchanged along time-varying patterns that progress up and down the computation tree.

The first contribution of this paper is to propose a detailed model to enable the comparison of ABFR with the classical Checkpoint/Restart (CR) approach. The model is valid for arbitrary N-body trees, which can be either binary trees, or quad-trees-, or oct-trees, and which are locally imbalanced to account for specific simulation requirements. The second and major contribution is to provide a comprehensive performance study for perfect binary trees. While the scenario of perfect binary trees is not the most general, it encompasses the intrinsic complexity of the whole model while being amenable to an exact analytical evaluation. In particular, setting the value of the detection interval, which bounds the error latency, is crucial to minimize the overhead incurred by ABFR to detect, and recover from, a latent error. We show how to compute the optimal value of this key parameter, and that the optimal value leads to significant savings over the CR approach. This result is an important step towards a full understanding of the potential impact of ABFR to N-body computations.

The rest of the paper is organized as follows. We start with background material in Sect. 2: in Sect. 2.1, we introduce Global View Resilience (GVR), the execution framework for resilient computing which is used as the support to deploy ABFR, and in Sect. 2.2, we briefly review N-body computations. Next, we outline the general principles of the ABFR approach in Sect. 3, and describe how to apply ABFR for N-body tree simulations. Then we provide a detailed formulation of the performance model in Sect. 4. Sections 5 and 6 are devoted to the performance study, and show how to compute the expected cost (Sect. 5) and expected overhead (Sect. 6) of the CR and ABFR approaches. Section 7 shows how to compute the optimal detection interval for ABFR. We report simulation results corresponding to a broad range of scenarios in Sect. 8. Section 9 presents related work. Finally, we give concluding remarks and hints for future directions in Sect. 10.

## 2 Background

### 2.1 Global View Resilience (GVR)

We use the GVR library to preserve application data and enable flexible recovery. GVR provides a global view of array data, enabling an application to easily create, version and restore (partial or entire) arrays. In addition, GVR's convenient naming enables applications to flexibly compute across versions of single or multiple arrays. GVR users can control where (data structure) and when (timing and rate) array versioning is done, and tune the parameters according to the needs of the application. The ability to create multi-version array and partially materialize them, enables flexible recovery across versions. GVR has been used to demonstrate flexible multi-version rollback, forward error correction, and other creative recovery schemes [5,6]. Demonstrations include high-error rates, and results show modest runtime cost (<1%) and programming effort in

full-scale molecular dynamics, Monte Carlo, adaptive mesh, and indirect linear solver applications [7,8].

GVR exploits both DRAM and high bandwidth and capacity burst buffers or other forms of non-volatile memory to enable low-cost, frequent versioning and retention of large numbers of versions. As needed, local disks and parallel file system can also be exploited for additional capacity. For example, NERSC Cori [9] supercomputer provides 1.8 PB SSDs in the burst buffer, with 1.7 TB/s aggregate bandwidth (6 GB/s per node). The JUQUEEN supercomputer at Jülich Supercomputing Center [10] is equipped with 2 TB flash memory, providing 2 GB/s bandwidth per node. Multi-versioning performance studies on JUQUEEN [10] showed GVR is able to create versions at full bandwidth, demonstrating low cost versioning is a reality [11]. In this paper, GVR's low-cost versioning enables flexible recovery for ABFR.

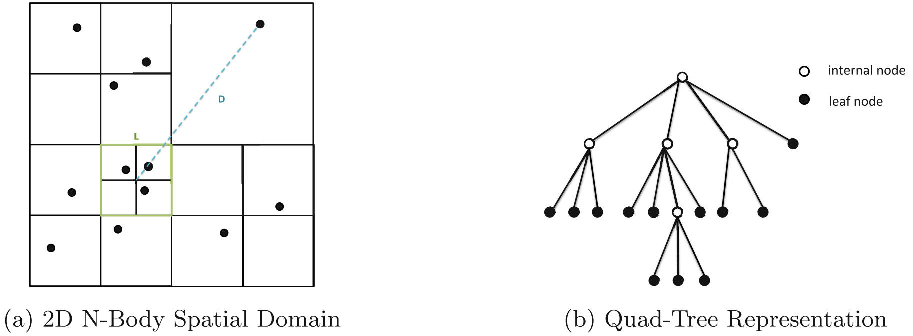
### 2.2 N-Body Computations

The N-body problem is the problem of predicting the motions of a dynamical system of objects, under the influence of physical forces, e.g. gravity. N-body simulations are a fundamental tool in the study of physical systems, from investigating three-body systems like the Earth-Moon-Sun to understanding the evolution of star clusters [12].

Over the past years, a number of methods have been introduced to solve N-body problem. The direct-summation method computes and integrates the pairwise forces on each particle with all others, in which the computation increases as $\mathcal{O}(N^2)$. Much effort [13–16] has been expended to reduce the complexity by approximating the contribution of many particles with a single interaction, resulting in complexity of $\mathcal{O}(N \log N)$. Among them, "tree codes" [15,17,18] are widely deployed, which use a tree structure to organize particles and group distant particles into one larger cell, allowing their gravity to be accounted for a single force. Barnes-Hut [17] is a commonly used tree algorithm, consisting of two major steps: first construct the tree and then compute the force of each particle by walking the tree.

**Tree construction:** A root node is used to encompass the full mass distribution. In 2D simulation, the space is repeatedly subdivided into four daughter nodes of half the side length each, until one ends up with single particles (see Fig. 1). After the topology of the tree has been constructed, the contents (mass, position) of each node are initialized by a post-order tree traversal.

**Force computation:** For each particle, forces are obtained by traversing the tree, i.e. starting at the root node, a decision is made whether or not to open a node (i.e. continue the tree walk) to provide an accurate enough partial force. Thus the error is controlled conveniently by the opening criterion, because higher accuracy is obtained by walking the tree to lower levels. The Barnes-Hut opening criterion determines if a node is sufficiently far away by computing $l/D$, where $l$ is the length of the region represented by the node, and $D$ the distance between the node's center-of-mass and the particle. If $l/D < \theta$ (i.e. opening criterion),




(a) 2D N-Body Spatial Domain (b) Quad-Tree Representation

**Fig. 1.** Barnes-Hut quad-tree computation for 2D N-body simulation

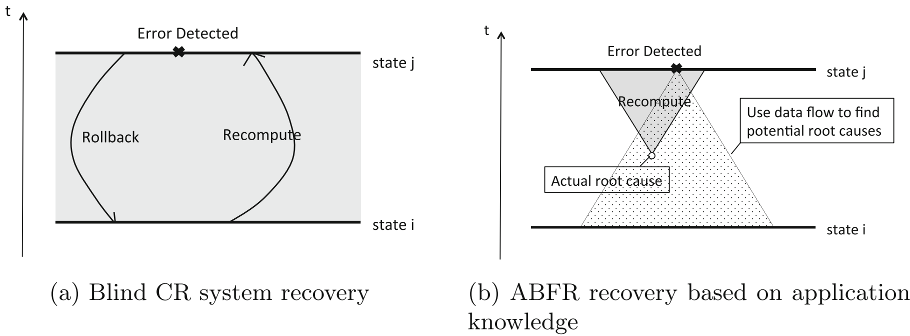
then approximate the particles in the node by their center of mass. Otherwise, continue the tree walk. The typical value of $\theta$ ranges from 0.3 to 0.8.

The reconstruction of full tree at each step can lead to significant overhead. As a result, the dominant time of simulation is spent on tree construction rather than force computation. McMillan and Aarseth [19] first discussed that the geometric structure of tree evolves slowly in time, therefore it is sufficient to reconstruct the tree once in a while to take into account the slow changes in the tree hierarchy. Gadget [20] proposed a dynamic tree update scheme, in which the tree node is updated without reconstructing the full tree. The tree reconstruction frequency can be controlled to improve computation efficiency. In our study, we adopt the dynamic tree update scheme and allow tree nodes to be updated with tunable frequency.

## 3 Algorithm-Based Focused Recovery (ABFR)

We propose to use the Algorithm-Based Focused Recovery (ABFR) approach [4] for N-body computations. ABFR exploits application semantics and versioned states to bound error impact and further localize recovery. ABFR exploits application algorithmics and data flow to identify potential root causes of a latent error and focus recovery effort on a small subset (see Fig. 2b). ABFR allows recovery to be overlapped with computation, reducing recovery overhead and enabling tolerance of high error rates. In contrast, checkpoint-restart (CR) (Fig. 2a) blindly rolls back the entire computation to the last verified checkpoint and recomputes everything.

We assume that a latent error detector (or "error check") is available. Such detectors are application-specific and computationally expensive. In order to keep the model general, we make the following assumptions:




(a) Blind CR system recovery

(b) ABFR recovery based on application knowledge

**Fig. 2.** Checkpoint restart (CR) vs. Algorithm-based focused recovery (ABFR).

- The error detector has 100%[1] coverage, finding some manifestation whenever there is an error, but not precisely identifying all manifestations.
- The error check detects error manifestations in the data, namely, corrupted values and their locations.
- Because latent ("silent") errors are complex to identify, the detector is computationally expensive.[2]

The interval between two consecutive error detections bounds the error latency. Given the error location and timing, three steps are performed to correct the state of corrupted data.

1. **Inverse propagation:** application logic and dataflow is used to inverse error propagation, identifying all data points in past that could have contributed to this error manifestation. These data points are called potential root causes (**PRC**). For N-Body tree computations, the errors may reside in leave nodes (i.e. no propagation) or propagate to some up-level nodes depending on the latency. Nodes that have interacted with the detected erroneous node in the latency bound are considered as PRCs. Therefore the tree structure and the error latency bound are used to invert error propagation and identify PRCs.
2. **Diagnosis:** to bound error impact more precisely, PRCs can be tested (diagnosis), eliminating many of the initial PRCs. For N-Body tree computations, this can be accomplished by recomputing intermediate states from versions (courtesy of GVR) and comparing to previously saved results. If the values match, the PRC can be pruned.
3. **Recovery:** recovery is applied to the reduced set of PRCs and their downstream error propagation paths. For instance, recovery can be recomputing PRCs and particles that have interacted with PRCs in the latency bound.

[1] Errors that cannot be detected are beyond the ability of any error recovery system to consider.

[2] Assuming expensive checks means that any improvements in checking can be incorporated – cost is not a disqualifier.

## 4 Analytical Performance Model

In this section, we introduce the model. We start with the application framework before detailing all error-related and fault-tolerance parameters. Table 1 summarizes main notations.

**Table 1.** Summary of main notations.

| Definitions | |
|---|---|
| $n$ | Height of tree |
| $K$ | Number of iterations performed at level $n$ (tree leaves) |
| Error rate | |
| $\lambda$ | Errors per second per leaf |
| Time | |
| $c$ | Time to compute one leaf |
| $d$ | Time to detect errors on one leaf |
| $v$ | Time to version one leaf |
| $r$ | Time to recover one leaf |
| Tree-wise | |
| $T_c$ | Time to compute the tree without errors |
| $T_d$ | Time for detection the tree without errors |
| $T_v$ | Time for versioning the tree without errors |
| Frequency | |
| $D$ | Detection interval of the form $2^x \cdot K$ |

**Application model.** We consider a perfect binary tree $\mathcal{T}_n$ of depth $n$. Leaves at the bottom of the tree hold the original data and perform computations, while internal nodes operate by aggregating the data of their two children and keeping a *summary.* The root is at level 0, and leaves are at level $n$. Nodes at different levels are updated with different rates: these rates are decreasing from bottom to top, so that leaves are updated the most frequently, while the root is updated the least frequently. The execution proceeds through iterations with global period $2^n$. Each iteration consists of $K$ computing steps at leaf level $n$, plus some information propagation, first bottom-up and then top-down, to exchange summary data. The scope of the propagation across the tree varies as follows. Every odd iteration is limited to level $n$ nodes (leaves), without any propagation. Iteration number $2j$ with $j$ odd is a depth-1 propagation that goes up to level $n-1$ nodes and then back to the leaves. Iteration number $4j$ with $j$ odd is a depth-2 propagation that goes up to level $n-2$ nodes and then back to the leaves. More generally, iteration number $2^i j$ with $j$ odd is a depth-$i$ propagation that goes up to level $n-i$ nodes and then back to the leaves. Hence the root is

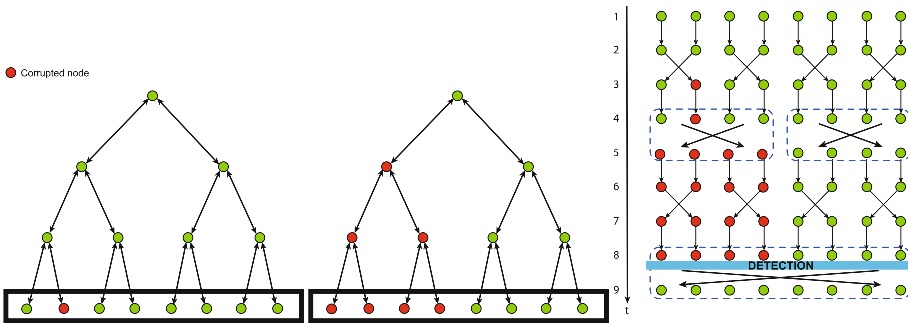
**Table 2.** Iterations and number of nodes updated each level for $\mathcal{T}_3$, with $n = 3$.

| Level | Iterations | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | | | | | | | | $2^0$ |
| 1 | | | | $2^1$ | | | | $2^1$ |
| 2 | | $2^2$ | | $2^2$ | | $2^2$ | | $2^2$ |
| 3 | $K \cdot 2^3$ | $K \cdot 2^3$ | $K \cdot 2^3$ | $K \cdot 2^3$ | $K \cdot 2^3$ | $K \cdot 2^3$ | $K \cdot 2^3$ | $K \cdot 2^3$ |

first updated at iteration $2^n$. Note that the root is updated only once per global period of $2^n$ iterations, which represent $2^n K$ computing steps at leaf level. The value of parameter $K$ is application-dependent. See an illustration with $n = 3$ on Table 2, which also shows how many nodes per iteration are updated at the different levels.

During a global period of $2^n$ iterations, the $2^n$ leaves execute $K$ computation steps every iteration, hence the total computing cost is $T_c = 4^n \cdot K \cdot c$, where $c$ denote the time of a computing step at the bottom level.

**Error model.** An error can strike at any time during the computations of the leaves. When an error occurs, it produces a localized error on a leaf, as shown in Fig. 3a. This error then spreads to other nodes every time data is exchanged, i.e., during iterations $1, 4, 6, 8$ in Table 2 and in Fig. 3c. This results in several leaves being corrupted after a few iterations (as shown by Fig. 3b). In particular, the whole tree will be corrupted after the root has been updated and the data has been sent back to all nodes, which happens every $2^n$ iterations (this corresponds to iteration 8 in Table 2). We assume that errors strike following an




(a) Tree at iteration $iter = 3$ (b) Tree at iteration $iter = 5$ (c) Leaves at each iteration

**Fig. 3.** Computation of a tree with $x = n = 3$. An error strikes a node at $t = 3$ (a). The error propagates to neighboring nodes following the communication pattern and four leaves are corrupted at $t = 5$ (b). Detection is done on the leaves at $t = 8$ after computation but before propagation (c).

Exponential probability distribution. Let $\lambda$ denote the error rate per leaf node, so that $1 - e^{-\lambda c}$ is the probability of having an error during the computation of one leaf, and $1 - e^{-\lambda T_c}$ the probability of having an error during the computation of the whole global period. We also assume that at most one error strikes during the execution of one period.

**Versioning.** Versioning a leaf consists in saving its current state. We assume that the cost of versioning is low in front of the detection cost. For simplicity, we version the state of the leaves every $K$ steps. Therefore the total time needed for versioning $2^n$ leaves, for a period of $2^n$ iterations, is $T_v = 2^n \cdot 2^n \cdot v$, where $v$ is the time to version a leaf.

**Detection.** Let $D$ denote the detection interval, i.e. the number of steps between two consecutive error checks. $D$ will be chosen as $D$ has a multiple of $K$. Detection is performed at level $x$ every $D$ time-steps. Finding the optimal value of $x$ is part of the optimization problem to be solved.

The detector is applied after computations at leaf nodes and before propagation to upper level nodes. $D$ is of the form $D = 2^x \cdot K$, where $x$ is an arbitrary integer between 0 and $n$. Therefore the detection is performed $2^{n-x}$ times during a global period. If $x = n$, detection occurs only once while if $x = 0$, detection occurs every $K$ steps, just as versioning. The total time for detection is $T_d = 2^{n-x} \cdot 2^n \cdot d$, where $d$ is the time to apply the detector at a leaf.

We assume that the detector is perfect: it always detects the manifestation of the error if one has struck. Finding how many leaves have been affected by the error (after its striking and until detection), and how many nodes must be recomputed, is performed through *diagnosis* and *recomputation*, respectively.

## 5 Performance Study: Expected Cost

In this section, we derive exact formulas for the expected total cost of the Checkpoint-Restart (CR) and Application-Based Focused Recovery (ABFR) approaches.

### 5.1 CR

**Theorem 1.** *The expected total cost for executing a global period with a binary tree of depth n using the CR approach is given by:*

$$\mathbb{E}(T_{CR}) = (2 - e^{-\lambda 4^n Kc}) \cdot 4^n \cdot K \cdot c + 2^n \cdot (d + v). \quad (1)$$

*Proof.* Let $\mathbb{E}(T_{CR})$ denote the expected cost for executing the entire period before checkpointing, using the standard CR approach. We first need to account for the cost of computation $T_c$. Detection and checkpointing are done at the end of the period, with cost $2^n d$ and $2^n s$ respectively, where $s$ denotes the time to save the state of a leaf onto global storage. If an error occurs, with probability

$(1-e^{-\lambda 2T_c})$, all nodes need to be recomputed, with cost $T_c$ again, from the last correct version. We can write:

$$\mathbb{E}(T_{CR}) = T_c + 2^n \cdot (d+v) + (1-e^{-\lambda T_c})T_c.$$

Then, setting $T_c = 4^n \cdot K \cdot c$ and simplifying, we retrieve Eq. 1.

### 5.2 ABFR

**Theorem 2.** *The expected total cost for executing a global period with a binary tree of depth $n$ using the ABFR approach is given by:*

$$\begin{aligned}\mathbb{E}(T_{ABFR}) &= 4^n \cdot K \cdot c + 2^{n-x} \cdot 2^n \cdot d + 4^n \cdot v \\ &+ (1-e^{-\lambda 4^n Kc})\left(\frac{1}{4}(4^x+2^x)(Kc+r) + \frac{1}{6}(4^x-1)(Kc+v)\right). \end{aligned} \quad (2)$$

*Proof.* Let $\mathbb{E}(T_{ABFR})$ denote the expected cost for executing the entire period using the ABFR approach. We first need to account for the cost of computation $T_c$, the cost of detection $T_d$ and the cost of versioning at every step $T_v$. Then, we need to account for the cost of diagnosis and recomputation in case of error. Let $T_{diag}$ and $T_{recomp}$ denote the time for diagnosis and recomputation, respectively. By definition, the probability that an error strikes during the period is given by $(1-e^{-\lambda T_c})$, therefore we can write:

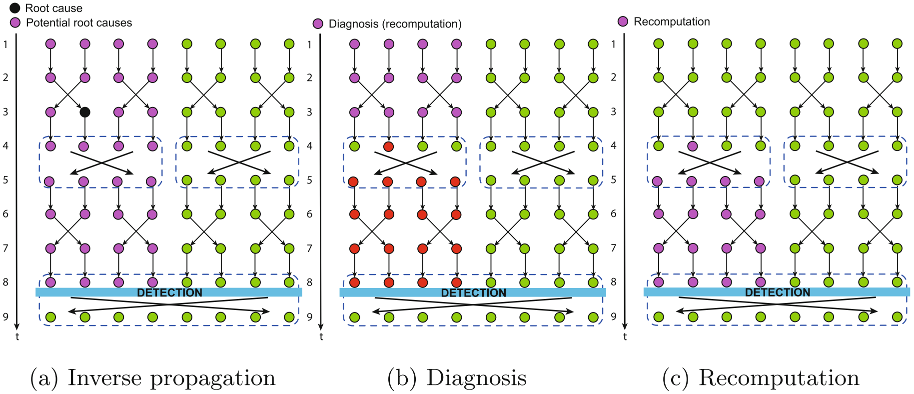
$$\mathbb{E}(T_{ABFR}) = T_c + T_d + T_v + (1-e^{-\lambda T_c})(T_{diag} + T_{recomp}).$$

Note that diagnosis and recomputation are random variables, because they depend upon when the error strikes. We take expectations and write:

$$\mathbb{E}(T_{ABFR}) = T_c + T_d + T_v + (1-e^{-\lambda T_c})\left(\mathbb{E}(T_{diag}) + \mathbb{E}(T_{recomp})\right) \quad (3)$$

**Inverse Propagation.** When an error is detected, we can use inverse error-propagation to identify the set of potential root causes. The number of potential root causes depends on the detection interval $D = 2^x K$. Indeed, the error can only be located in the $2^{x-1}$ nodes connected to the manifestation of the error, as shown in Fig. 4a.

With $x = n = 3$, this means that there is exactly one detection during the execution of the entire tree. Remember that detection is done after computations, but before propagation. Therefore in this example the number of potential root causes can be restricted to the 4 leaves (out of 8 leaves) that are directly linked to the manifestation of the error. Similarly, setting $x = 2$ and $n = 3$ means two detections during the execution of the tree and at most 2 potential root causes (out of 8 leaves).




(a) Inverse propagation (b) Diagnosis (c) Recomputation

**Fig. 4.** Computation of the leaves with $x = n = 3$. (a) After an error has been detected, we use inverse propagation to identify the set of potential root causes; (b) Then we use diagnosis, i.e. we recompute potential root causes from the last correct version and check again old versions, to locate the root cause; (c) Finally, we recompute all corrupted nodes.

**Diagnosis.** There are $2^x$ versions in one detection interval. Knowing that an error has occurred, the probability of having an error in each version is uniformly distributed. Thus the probability of having an error in version $i$ (resp. iteration $i$) is $\frac{1}{2^x}$. Diagnosis is done by recomputing all potential root causes and comparing the result with previous versions (as shown in Fig. 4b). We need to recompute (and reload) $2^{x-1}$ nodes $i$ times in order to locate the root cause of the error. Thus the expected time for diagnosis is given by:

$$\mathbb{E}(T_{diag}) = \sum_{i=1}^{2^x} \frac{1}{2^x} \cdot i \cdot 2^{x-1}(Kc + r)$$
$$= 2^{x-2}(2^x + 1)(Kc + r) = \frac{1}{4}(4^x + 2^x)(Kc + r). \quad (4)$$

**Recomputation.** Recomputation follows diagnosis. The root cause of the error has been localized and we know which node must be recomputed, as shown in Fig. 4c. As seen in diagnosis, when an error is detected, we only need to consider the $2^{x-1}$ nodes that are directly linked to the manifestation of the error. Now, depending on the actual location of the root cause.

The probability that an error strikes a node is uniformly distributed in space and time. Therefore, we start with two cases:

1. With probability $\frac{1}{2}$, the error has struck during the first $2^{x-1}$ iterations. This means that the error has propagated to all of the $2^{x-1}$ nodes in the last $2^{x-1}$ iterations, and that we must recompute *at least* $2^{x-1}$ nodes $2^{x-1}$ times.
2. With probability $\frac{1}{2}$ the error has struck in the other $2^{x-1}$ nodes and we don't need to recompute any of the first $2^{x-1}$ nodes.

We can write

$$\mathbb{E}(T_{recomp}) = \frac{1}{2} 2^{x-1} \cdot 2^{x-1} \cdot (Kc + v) + \dots$$

Then, we have four cases:

1. With probability $\frac{1}{4}$, the error has struck in the *first* $2^{x-2}$ nodes of the *first* $2^{x-1}$ iterations.
2. With probability $\frac{1}{4}$, the error has struck in the *last* $2^{x-2}$ nodes of the *first* $2^{x-1}$ iterations.
3. With probability $\frac{1}{4}$, the error has struck in the *first* $2^{x-2}$ nodes of the *last* $2^{x-1}$ iterations.
4. With probability $\frac{1}{4}$, the error has struck in the *last* $2^{x-2}$ nodes of the *last* $2^{x-1}$ iterations.

In cases 1 and 3, we need to recompute *at least* $2^{x-2}$ nodes $2^{x-2}$ times. However, in cases 3 and 4, we do not need to recompute these nodes. We can write:

$$\mathbb{E}(T_{recomp}) = \frac{1}{2} 2^{x-1} \cdot 2^{x-1} \cdot (Kc + v) + \frac{2}{4} (2^{x-2} \cdot 2^{x-2} \cdot (Kc + v) + \dots$$

This approach can be used recursively to compute the probability and cost of all possible scenarios (i.e. for all possible error locations). See Fig. 5 for an example with $x = n = 3$. We derive that:

$$\begin{aligned} \mathbb{E}(T_{recomp}) &= \sum_{i=1}^{x} \frac{2^{i-1}}{2^i} 2^{x-i} \cdot 2^{x-i} \cdot (Kc + v) \\ &= \sum_{i=1}^{x} \frac{1}{2} 4^{x-i} \cdot (Kc + v) = \frac{1}{6} (4^x - 1)(Kc + v) \ . \end{aligned} \tag{5}$$

$$\mathbb{E}(T_{recomp}) = \frac{1}{2} \begin{cases} 2^{n-1} 2^{n-1} (K \cdot c + v) + \frac{1}{2} \begin{cases} 2^{n-2} 2^{n-2} (K \cdot c + v) + \frac{1}{2} \begin{cases} (K \cdot c + v) \\ 0 \end{cases} \\ \frac{1}{2} \begin{cases} (K \cdot c + v) \\ 0 \end{cases} \end{cases} \\ \frac{1}{2} \begin{cases} 2^{n-2} 2^{n-2} (K \cdot c + v) + \frac{1}{2} \begin{cases} (K \cdot c + v) \\ 0 \end{cases} \\ \frac{1}{2} \begin{cases} (K \cdot c + v) \\ 0 \end{cases} \end{cases} \end{cases}$$

**Fig. 5.** Computation of the expected recomputation cost for $x = n = 3$ considering all 8 possible scenarios. The error can hit any one of the 8 iterations with uniform probability.

**Expected Cost.** Altogether, putting expressions for diagnosis (see Eq. 4) and recomputation (see Eq. 5) back into Eq. 3, we retrieve Eq. 2.

## 6 Performance Analysis: Expected Overhead

In this section, we derive exact formulas for the overhead incurred by using the CR or ABFR approach. For either method, the expected overhead is defined as $\mathbb{E}(H_X) = \frac{\mathbb{E}(T_X)}{T_c} - 1$, where $T_X$ denotes the cost for method X. Recall that $T_c$ is the baseline cost, so that the overhead measures extra the fraction of work spent to mitigate the impact of errors.

### 6.1 CR

Let $\mathbb{E}(H_{CR})$ denote the expected overhead for CR. We can write:

$$\mathbb{E}(H_{CR}) = \frac{\mathbb{E}(T_{CR})}{T_c} - 1.$$

Taking Eq. 1 for $\mathbb{E}(T_{CR})$ and setting $T_c$ to $4^n \cdot K \cdot c$, we obtain:

$$\mathbb{E}(H_{CR}) = 1 - e^{-\lambda 4^n Kc} + \frac{d+v}{2^n \cdot K \cdot c}. \tag{6}$$

### 6.2 ABFR

Let $\mathbb{E}(H_{ABFR})$ denote the expected overhead for ABFR. We can write:

$$\mathbb{E}(H_{ABFR}) = \frac{\mathbb{E}(T_{ABFR})}{T_c} - 1.$$

Taking Eq. 2 for $\mathbb{E}(T_{ABFR})$ and setting $T_c = 4^n \cdot K \cdot c$, we obtain:

$$\begin{aligned}\mathbb{E}(H_{ABFR}) = & \frac{4^n \cdot K \cdot c + 2^{n-x} \cdot 2^n \cdot d + 4^n \cdot v}{4^n Kc} \\ & + \frac{(1 - e^{-\lambda 4^n Kc})}{4^n Kc} \left( \frac{1}{4}(4^x + 2^x)Kc + \frac{1}{6}(4^x - 1)(Kc) \right) \\ = & \frac{2^{-x} d + v}{Kc} + \frac{(1 - e^{-\lambda 4^n Kc})}{4^n Kc} \left( \frac{1}{4}(4^x + 2^x)(Kc + r) + \frac{1}{6}(4^x - 1)(Kc + v) \right).\end{aligned} \tag{7}$$

## 7 Optimal Detection Interval for ABFR

We now show how to derive the optimal detection interval for ABFR. Recall that the detection interval is of the form $D = 2^x \cdot K$. Our goal is to find the optimal value for $x$, denoted by $x^*$.

First, we use Taylor series to approximate $1 - e^{-\lambda 4^n Kc}$ to $\lambda 4^n Kc + O(\lambda^2)$, and derive that:

$$\mathbb{E}(H_{ABFR}) = \frac{2^{-x}d + v}{Kc} + \lambda\Big(\frac{4^x + 2^x}{4}(Kc + r) + \frac{4^x - 1}{6}(Kc + v)\Big).$$

Then, in order to get the optimal value for $x$, denoted by $x^*$, we need to solve the following equation:

$$\frac{\partial \mathbb{E}(H_{ABFR})}{\partial x} = 0, \tag{8}$$

Note that letting $y = 2^x$ in Eq. (8) leads to solving a third-degree equation in $y$, so it is possible to obtain a closed-form expression for the optimal value $y^*$, and hence for $x^*$. In the following, we simply solve Eq. (8) numerically, obtain the optimal solution as a real variable, and using nearest rounding to retrieve the optimal integer value. Finally, plugging $x^*$ back into Eq. 7, we obtain $\mathbb{E}(H_{ABFR}^{opt})$.

### 7.1 Limits of the Analysis

For the sake of simplicity, we have made the assumption that only one error can strike during the computation of a tree, meaning that (1) re-execution after an error always succeeds, and (2) diagnosis only needs to find one root cause. While this makes for a good approximation with large MTBE, the error rate can only get so small in the analysis. In particular, we must ensure that MTBE $>> T_c$ in order to keep the probability of having more than one error as low as possible.

Note that this is a common assumption when dealing with CR models. However there are several possible ways the model could be extended to handle multiple errors. First, multiple errors within a detection interval could trigger multiple ABFR responses. Alternatively, diagnosis and recovery could be extended to deal with multiple errors concurrently. These are promising directions for future work.

## 8 Simulations

In this section, we run a set of simulations whose goal is twofold: (1) show the accuracy of the theoretical analysis; and (2) assess the performance of the proposed ABFR approach against the standard CR approach. We describe the settings of the simulations in Sect. 8.1 and we present the results in Sect. 8.2.

### 8.1 Settings

We target large platforms subject to silent errors. Such platforms can handle large simulations with millions of nodes, and we set $n = \lceil log_2(10^6) \rceil = 20$. The time needed to compute one node in N-Body computation is typically measured at around $c = 10^{-5}$ s and we set the number of iterations at the bottom level to $K = 100$. In addition, we assume that ABFR can take advantage of high bandwidth, high capacity burst buffers or other form of non-volatile memory to

perform low-cost, frequent versioning, and we set cost to version and recover a node to $r = v = \frac{c}{100}$. Detection, on the opposite, is assumed to be expensive and we set the detection cost for one node to $d = 100 \cdot c$. Finally, we set the error rate to $\lambda = 1.15 \cdot 10^{-10}$, which corresponds to a MTBE of 275 years for a single processor (or one day on a platform with 100000 of such processors).

Simulations are based on the model and we instantiate the model using the above values by default. Errors are injected into the computation following the error rate $\lambda$. Note that at most one error is injected into the computation of a tree and that errors can strike any node with uniform probability. When an error strikes a node, diagnosis and recomputation are computed according to the exact number of nodes that need to be recomputed for diagnosis and recomputation, with respect to the error location. The overhead of the simulation is obtained by averaging the results of 1000 runs.

### 8.2 Results

In this section, we present the results of the simulations for different scenarios.

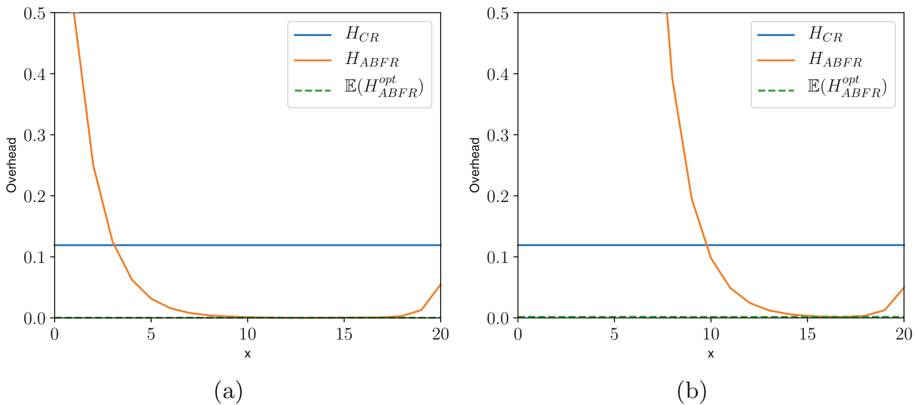



**Fig. 6.** Overhead of CR and ABFR approaches for different values of $x$ and with detection cost $d = 100 \cdot c$ (a) and $d = 10000 \cdot c$ (b).

**Impact of Detection Interval.** Figure 6 shows the expected overhead obtained using the CR and ABFR approach, denoted by $H_{CR}$ and $H_{ABFR}$, respectively, for all possible values of $x$ between 0 and $n$. We show the results for the default detection cost $d = 100 \cdot c$ (a) and for a much larger detection cost $d = 10000 \cdot c$ (b). In addition, we plot the theoretical optimal expected overhead for ABFR, denoted by $\mathbb{E}(H^{opt}_{ABFR})$, which is obtained with Eq. 7 for the optimal value of $x$. By solving Eq. 8 numerically, we find that $x^* = 14$ for $d = 100 \cdot c$ and $x^* = 17$ for $d = 10000 \cdot c$.

First, we observe that, in both cases, the optimal overhead obtained with the simulations closely matches the optimal theoretical overhead, which confirms the accuracy of the analysis. Then, we can see that both figures show a dramatic increase of the overhead for small values of $x$. This is because the detection interval $D$ is of the form $2^x \cdot K$. This means that decreasing $x$ (and therefore the detection interval $D$) causes an exponential increase in the number of detections, which in turns increases the overhead. In addition, we note a slight increase of the overhead for large values of $x$. Indeed, when the detection interval is too large, (e.g. only one detection at the end of the computation when $x = n = 20$), errors have more time to propagate and more nodes need to be recomputed as a result, which increases the recovery cost, and therefore the overhead.

While the optimal overhead is very sensitive to the detection interval, we observe (by comparing both scenarios) that it does not vary much as a function of the detection cost. This is because the detection cost only represents a small part of the total computation. Overall, we show that ABFR is able to improve the overhead by several orders of magnitude compared to the standard CR approach, and is up to 120 times more efficient with this setting.

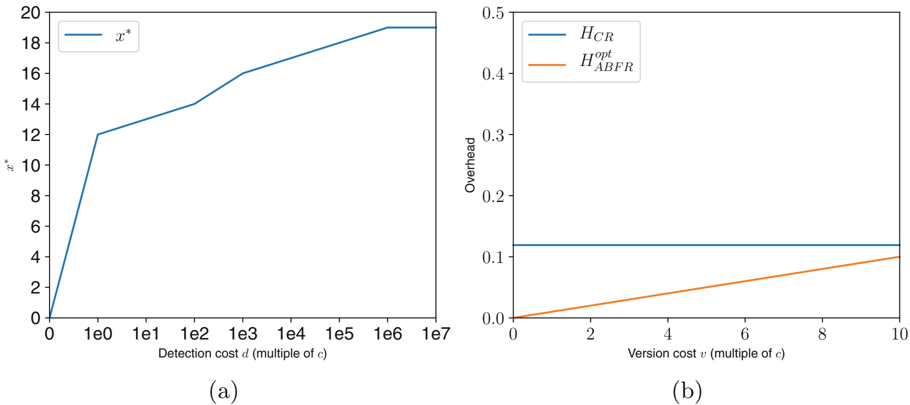



**Fig. 7.** Impact of the detection cost on the optimal $x$ (a) and impact of the versioning cost on the optimal overhead (b).

**Impact of Detection and Version Cost.** The detection cost $d$ has almost no effect on the optimal overhead (as shown in the previous scenario). It does however have an impact on the optimal value of $x$. Figure 7(a) shows the optimal $x^*$ obtained for different detection costs. We can see that unless the detection cost is extremely small, the optimal $x^*$ must be a trade-off between the detection cost and the recovery cost in case of error.

As opposed to the detection cost, the versioning cost $v$ has no effect on the optimal $x^*$, but its value can have a significant impact on the overhead. Because all nodes are versioned, the overhead increases linearly with the version cost $v$, as

shown in Fig. 7(b), and we must ensure that this cost remains cheap for ABFR to perform better than CR.

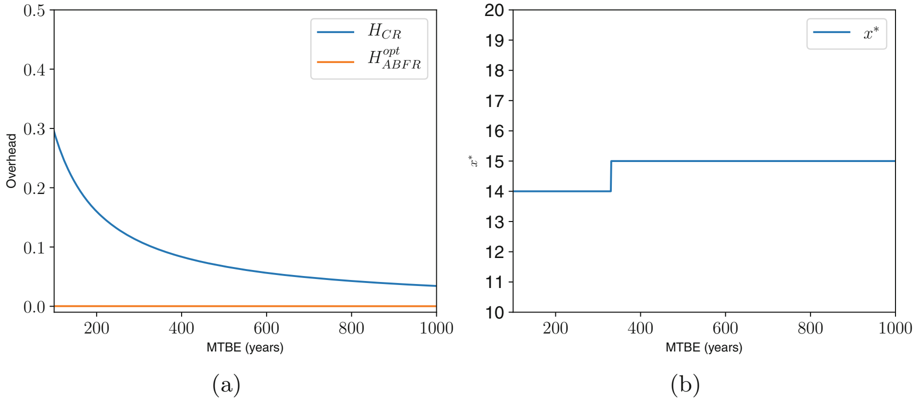



**Fig. 8.** Overhead of CR and ABFR for different MTBE (a) and corresponding optimal detection interval $x^*$ (b).

**Impact of MTBE.** Figure 8(a) shows the overhead obtained with CR and ABFR for different MTBE ranging from 100 years to 1000. Figure 8(b) shows the corresponding optimal detection interval for ABFR. We can see that ABFR scales much better than CR with high error rates ($MTBE < 400$ years). Where CR needs to recompute the entire tree in case of error, ABFR has the ability to detect the error earlier in the computation, and to recompute only a fraction of all the nodes. Indeed, the number of nodes to recompute in case of error depends on the size of the detection interval $D$ and is at most $O(4^x)$ for ABFR, while it is exactly $\Theta(4^n)$ for CR. Note that for the same reason, ABFR remains better than CR even with low error rates ($MTBE > 1000$ years).

**Impact of Error Latency.** Figure 9(a) shows the recovery cost of ABFR normalized with respect to the recovery cost of CR, while Fig. 9(b) shows the cost of diagnosis and recomputation normalized with respect to the recovery cost of ABFR. For the sake of simplicity, we simulated the execution of small trees with $x = n = 10$. Here there are $2^{10} \cdot K = 1024 \cdot K$ iterations in total. The x-axis denotes the number of iterations already done before the error occurred from 1 (first iteration) to $1024 \cdot K$ (last iteration). Because the detection interval $x$ is set to $x = n = 10$, both CR and ABFR detect faults only at the end of the computation.

First, we note that the cost of diagnosis is linear, while the cost of recomputation is not. Indeed, diagnosis is done by recomputing all iterations from the last correct version until we find the error, while recomputation will skip part of

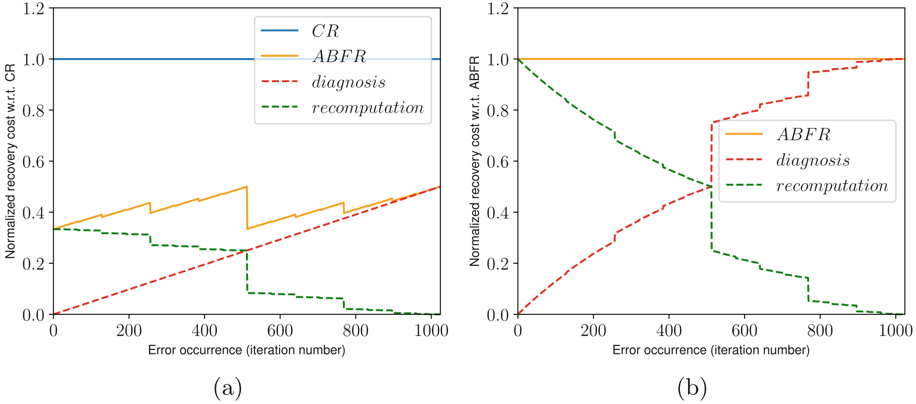



**Fig. 9.** Recovery cost normalized w.r.t. CR (a) and w.r.t. ABFR (b) with detection interval set to $x = n = 10$.

the nodes depending on the location of the error. In particular, the spike that we can observe at iteration 512 corresponds to the biggest propagation step. If the error strikes right before this step, then exactly half of the remaining nodes must be recomputed. On the contrary, if the error strikes right after this step, it is not possible for the error to propagate further, and only one fourth of the remaining nodes needs to be recomputed. Note that, as shown in Sect. 5.2, we never need to recompute more than half of all the nodes, hence the recovery cost of ABFR is at worst 48% better than CR, and at best it is 65% better than CR.

## 9 Related Work

Latent errors, also known as silent errors or silent data corruption, represent a major threat to scientific applications executing on large scale platforms [21–23]. There are several causes of silent errors, such as cosmic radiation, packaging pollution, among others. Silent errors can strike the cache and memory (bit flips) as well as CPU operations; in the latter case they resemble floating-point errors due to improper rounding, but have a dramatically larger impact because any bit of the result, not only low-order mantissa bits, can be corrupted. In contrast to a fail-stop error whose detection is immediate, a latent error is identified only when the corrupted data leads to an unusual application behavior. This detection latency renders periodic checkpointing insufficient: if the error struck before the last checkpoint, and is detected after that checkpoint, then the checkpoint is corrupted and cannot be used for rollback. This is why checkpointing must be coupled with some verification mechanism, in order to detect any latent error before taking a new checkpoint.

Replication remains the most transparent and least intrusive technique and can be used at different levels (duplication, triplication or even more). Combined with checkpointing, replication comes with two flavors: *process replication* [24, 25] and *group replication* [26]. Process replication applies to message-passing

applications with communicating processes. Each process is replicated, and the platform is composed of process pairs, or triplets. Group replication applies to black-box applications, whose parallel execution is replicated several times. The platform is partitioned into two halves (or three thirds). In both scenarios, results are compared before each checkpoint, which is taken only when both results (duplication) or two out of three results (triplication) coincide. If not, one or more silent errors have been detected, and the application rolls back to the last checkpoint. Note that duplication enables to detect but not to correct a latent error, while triplication enables both. Replication is not a new technique. Triple Modular Redundancy, or TMR [27], is the standard fault-tolerance approach for critical systems, such as embedded or aeronautical devices [28]. However, triplication has a high cost, since two-thirds of the processors are executing redundant work, and HPC scientists are not ready to pay such a price.

To address the problem of latent errors in HPC, many application-specific detectors have been proposed. Indeed, application-specific information enables ad-hoc solutions, which dramatically decrease the cost of error detection. Algorithm-based fault tolerance (ABFT) [29–31] is a well-known technique, which uses checksums to detect up to a certain number of errors in linear algebra kernels. Unfortunately, ABFT can only protect datasets in linear algebra kernels, and it must be implemented for each different kernel, which incurs a large amount of work for large HPC applications. Other techniques have also been advocated. Benson, Schmit and Schreiber [32] compare the result of a higher-order scheme with that of a lower-order one to detect errors in the numerical analysis of ODEs and PDEs. Sao and Vuduc [33] investigate self-stabilizing corrections after error detection in the conjugate gradient method. Heroux and Hoemmen [34] propose linear solvers to tolerant soft faults using selective reliability. Elliot et al. [35] design a fault-tolerant GMRES capable of converging despite latent errors. Bronevetsky and de Supinski [36] provide a comparative study of detection costs for iterative methods. Recently, several silent error detectors based on data analytics have been proposed, showing promising results. These detectors use several interpolation techniques such as time series prediction [37] and spatial multivariate interpolation [38–40]. Such techniques offer large detection coverage for a negligible overhead. However, these detectors do not guarantee full coverage; they can detect only a certain percentage of corruptions (i.e., partial verification with an imperfect recall). Nonetheless, the accuracy-to-cost ratios of these detectors are high, which makes them interesting alternatives at large scale. Similar detectors have also been designed to detect silent errors in the temperature data of the Orbital Thermal Imaging Spectrometer (OTIS) [41].

The ABFR approach presented in this paper is similar to ABFT approaches, exploiting application knowledge for error detection, but adding the use of application knowledge to diagnose what state is potentially corrupted, and using that knowledge to limit recomputation, and thereby achieve efficient recovery from latent errors. Recently, we have successfully applied ABFR to stencil computations [4], which are perfectly suited to ABFR due to their regular and neighbor-based communication pattern. The tree-based propagation pattern of N-Body computations is much more challenging for ABFR.

## 10 Conclusion

We have applied ABFR for N-Body tree computations to efficiently recover from latent errors. By exploiting application data flow and intermediate states, ABFR focuses recovery on an accurate estimate of potentially corrupted data, reducing recovery cost significantly. To explore the performance of ABFR, we build an analytical model parameterized by error rate and detection interval for a perfect binary tree. Simulation results show that ABFR reduces 50% of recovery overhead compared to checkpoint-restart approach. While the model is built for binary trees, it can be generalized to higher dimensions of simulations. Future directions include applying ABFR to production N-Body tree codes and demonstrating an application-agnostic ABFR runtime that supports portable and scalable performance.

## References


1. Snir, M., Wisniewski, R.W., Abraham, J.A., Adve, S.V., Bagchi, S., Balaji, P., Belak, J., Bose, P., Cappello, F., Carlson, B., et al.: Addressing failures in exascale computing. Int. J. High Perform. Comput. Appl. **28**(2), 129–173 (2014)
2. Huang, K.H., Abraham, J.: Algorithm-based fault tolerance for matrix operations. IEEE Trans. Comput. **100**(6), 518–528 (1984)
3. Chen, Z.: Online-ABFT: an online algorithm based fault tolerance scheme for soft error detection in iterative methods. In: PPoPP, pp. 167–176 (2013)
4. Fang, A., Cavelan, A., Robert, Y., Chien, A.A.: Resilience for stencil computations with latent errors. In: The 46th International Conference on Parallel Processing (ICPP 2017). IEEE Computer Society Press (2017)
5. Dun, N., et al.: Data decomposition in monte carlo neutron transport simulations using global view arrays. Int. J. High Perform. Comput. Appl. **29**, 348–365 (2015)
6. Fang, A., Chien, A.A.: Applying GVR to molecular dynamics: enabling resilience for scientific computations. Technical report TR-2014-04, University of Chicago (2014)
7. Chien, A., et al.: Versioned distributed arrays for resilience in scientific applications: global view resilience. Procedia Comput. Sci. **51**, 29–38 (2015)
8. Chien, A., et al.: Exploring versioned distributed arrays for resilience in scientific applications: global view resilience. Int. J. High Perform. Comput. Appl. (2016)
9. Platform: NERSC CORI. https://www.nersc.gov/users/computational-systems/cori/
10. Platform: JUQUEEN. http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/JUQUEEN_node.html
11. Dun, N., Pleiter, D., Fang, A., Vandenbergen, N., Chien, A.A.: Multi-versioning performance opportunities in BGAS system for resilience. In: Kunkel, J.M., Balaji, P., Dongarra, J. (eds.) ISC High Performance 2016. LNCS, vol. 9697, pp. 486–504. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41321-1_25
12. Blelloch, G., Narlikar, G.: A practical comparison of $n$-body algorithms. In: Parallel Algorithms. Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society (1997)
13. Eastwood, J., Hockney, R.: Computer Simulation Using Particles. McGrawHill, New York (1981)

14. Van Albada, G., Van Leer, B., Roberts Jr., W.: A comparative study of computational methods in cosmic gas dynamics. Astron. Astrophys. **108**, 76–84 (1982)
15. Appel, A.W.: An efficient program for many-body simulation. SIAM J. Sci. Statist. Comput. **6**(1), 85–103 (1985)
16. Greengard, L., Rokhlin, V.: A fast algorithm for particle simulations. J. Comput. Phys. **73**(2), 325–348 (1987)
17. Barnes, J., Hut, P.: A hierarchical o (n log n) force-calculation algorithm. Nature **324**(6096), 446–449 (1986)
18. Hernquist, L.: Performance characteristics of tree codes. Astrophys. J. Suppl. Ser. **64**, 715–734 (1987)
19. McMillan, S.L., Aarseth, S.J.: An o (n log n) integration scheme for collisional stellar systems. Astrophys. J. **414**, 200–212 (1993)
20. Springel, V., Yoshida, N., White, S.D.: Gadget: a code for collisionless and gasdynamical cosmological simulations. New Astronomy **6**(2), 79–117 (2001)
21. O'Gorman, T.: The effect of cosmic rays on the soft error rate of a DRAM at ground level. IEEE Trans. Electron. Devices **41**(4), 553–557 (1994)
22. Ziegler, J.F., Curtis, H.W., Muhlfeld, H.P., Montrose, C.J., Chin, B.: IBM experiments in soft fails in computer electronics. IBM J. Res. Dev. **40**(1), 3–18 (1996)
23. Moody, A., Bronevetsky, G., Mohror, K., Supinski, B.R.d.: Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: SC. ACM (2010)
24. Ferreira, K., Stearley, J., Laros, J.H.I., Oldfield, R., Pedretti, K., Brightwell, R., Riesen, R., Bridges, P.G., Arnold, D.: Evaluating the viability of process replication reliability for exascale systems. In: SC 2011. ACM (2011)
25. Fiala, D., Mueller, F., Engelmann, C., Riesen, R., Ferreira, K., Brightwell, R.: Detection and correction of silent data corruption for large-scale high-performance computing. In: SC. ACM (2012)
26. Casanova, H., Bougeret, M., Robert, Y., Vivien, F., Zaidouni, D.: Using group replication for resilience on exascale systems. Int. J. High Perform. Comput. Appl. **28**(2), 210–224 (2014)
27. Lyons, R.E., Vanderkulk, W.: The use of triple-modular redundancy to improve computer reliability. IBM J. Res. Dev. **6**(2), 200–209 (1962)
28. Avizienis, A., Laprie, J., Randell, B., Landwehr, C.E.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Sec. Comput. **1**(1), 11–33 (2004)
29. Huang, K.H., Abraham, J.A.: Algorithm-based fault tolerance for matrix operations. IEEE Trans. Comput. **33**(6), 518–528 (1984)
30. Bosilca, G., Delmas, R., Dongarra, J., Langou, J.: Algorithm-based fault tolerance applied to high performance computing. J. Parallel Distrib. Comput. **69**(4), 410–416 (2009)
31. Shantharam, M., Srinivasmurthy, S., Raghavan, P.: Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In: ICS. ACM (2012)
32. Benson, A.R., Schmit, S., Schreiber, R.: Silent error detection in numerical time-stepping schemes. Int. J. High Perform. Comput. Appl. **29**, 403–421 (2014)
33. Sao, P., Vuduc, R.: Self-stabilizing iterative solvers. In: ScalA 2013 (2013)
34. Heroux, M., Hoemmen, M.: Fault-tolerant iterative methods via selective reliability. Research report SAND2011-3915 C, Sandia National Laboratories (2011)
35. Elliott, J., Hoemmen, M., Mueller, F.: Evaluating the impact of SDC on the GMRES iterative solver. In: IPDPS. IEEE (2014)
36. Bronevetsky, G., de Supinski, B.: Soft error vulnerability of iterative linear algebra methods. In: ICS. ACM (2008)

37. Berrocal, E., Bautista-Gomez, L., Di, S., Lan, Z., Cappello, F.: Lightweight silent data corruption detection based on runtime data analysis for HPC applications. In: HPDC. ACM (2015)
38. Bautista Gomez, L., Cappello, F.: Detecting silent data corruption through data dynamic monitoring for scientific applications. In: PPoPP. ACM (2014)
39. Bautista Gomez, L., Cappello, F.: Detecting and correcting data corruption in stencil applications through multivariate interpolation. In: FTS. IEEE (2015)
40. Bautista Gomez, L., Cappello, F.: Exploiting spatial smoothness in HPC applications to detect silent data corruption. In: HPCC. IEEE (2015)
41. Ciocca, E., Koren, I., Koren, Z., Krishna, C.M., Katz, D.S.: Application-level fault tolerance in the orbital thermal imaging spectrometer. In: PRDC. IEEE (2004)

# Multi-fidelity Surrogate Modeling for Application/Architecture Co-design

Yiming Zhang[1], Aravind Neelakantan[2(✉)], Nalini Kumar[2], Chanyoung Park[1], Raphael T. Haftka[1], Nam H. Kim[1], and Herman Lam[2]

[1] Department of Mechanical and Aerospace Engineering, University of Florida, Gainesville, FL 32608, USA
{yimingzhang521,cy.park,haftka,nkim}@ufl.edu

[2] Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32608, USA
{aravindneela,nkumar,hlam}@ufl.edu

**Abstract.** The HPC community has been using abstract, representative applications and architecture models to enable faster co-design cycles. While developers often qualitatively verify the correlation of the application abstractions to the parent application, it is equally important to quantify this correlation to understand how the co-design results translate to the parent application. In this paper, we propose a multi-fidelity surrogate (MFS) approach which combines data samples of low-fidelity (LF) models (representative apps and architecture simulation) with a few samples of a high-fidelity (HF) model (parent app). The application of MFS is demonstrated using a multi-physics simulation application and its proxy-app, skeleton-app, and simulation models. Our results show that RMSE between predictions of MFS and the baseline HF models was 4%, which is significantly better than using either LF or HF data alone, demonstrating that MFS is a promising approach for predicting the parent application performance while staying within a computational budget.

**Keywords:** Performance estimation · Multi-fidelity surrogate
Behavioral emulation

## 1 Introduction

As we approach exascale computing, the next frontier in high-performance computing, it is important that application developers and system designers co-design to develop better performing and more energy efficient application codes and machines [6]. For fast and effective turnaround during the co-design process, application developers create representative applications which are abstract, smaller, and self-contained descriptions of their application code (also called parent app) and only capture the key parameters and features that predominantly influence the outcome of co-design [13]. To further speed up the co-design process

© Springer International Publishing AG 2018
S. Jarvis et al. (Eds.): PMBS 2017, LNCS 10724, pp. 179–196, 2018.
https://doi.org/10.1007/978-3-319-72971-8_9

and to enable architecture design-space exploration (DSE), system architects build simulator models to study the application performance on various underlying architectures. Behavioral Emulation (BE) [19] is one such coarse-grained approach for simulation of extreme-scale systems and applications. While the parent application can be used to drive architecture simulations, abstract application end-point models are often used to represent the parent app to speed up the co-design process.

Representative applications, in the form of mini-apps, proxy-apps, or skeleton apps, have been developed for many scientific HPC codes (parent app) and are a necessity in cases where the actual application cannot be shared with the hardware architects [5,7,16,18]. After development, it is important to validate these representative apps against their parent apps to ensure they are reasonably accurate representations of the application behavior. Typically, this qualitative validation is performed by comparing ratio of computation to communication, weak scaling and strong scaling trends, similarity analysis, etc. Similarly, performance prediction results of architecture simulations are verified against testbed measurements. After validation, both the representative apps and simulator models can be used as platforms to evaluate tradeoffs for improved performance, power, and resilience, different programming models, compilers, etc. and guide the refinement of parent application. *Qualitative* validation is important; however, it is also important to determine *quantitatively* how the improvements in a representative app or architecture translate to the parent app. To the best of our knowledge no solution for quantitative validation of representative applications under a reasonable computational budget has been proposed in the literature.

In general, surrogate models are approximations that are fit to the available data of a phenomenon of interest, herein the parent app. A high-fidelity surrogate model (HFM) can be constructed from more accurate and computationally expensive high-fidelity data (e.g., benchmarking data using parent app); and a low-fidelity model (LFM) can be constructed from computationally cheaper but less accurate low-fidelity data (e.g., skeleton apps, simulation results). In this paper, we propose the use of a multi-fidelity surrogate model (MFS) for identifying the relation between parent and representative apps. The MFS works when LF and HF have similar trends/curvature in the design space. An indication of trend similarity between LF and HF is the scale factor. A scale factor around 1 denotes high correlation between HF and LF whereas a negative or extremely large scale factor denotes an inappropriate LF under certain discrepancy (discussed in Sect. 4).

The concept of MFS has been extensively studied to approximate the high-fidelity models (HFMs) assisted by cheaper low-fidelity models (LFMs) [12]. To balance accuracy and computational cost associated with data collection, the MFS approach aims to develop a surrogate model based mostly on LF samples assisted with only a few HF samples. Typical multi-fidelity models include finite element analysis with different resolution, physical tests versus numerical simulations, etc. [12,17,25].

Representative apps are often used for studying the performance impact of various optimization techniques. But it is important to validate these changes on the parent app. In order to use our proposed approach, changes have to be made to both the representative and parent app. The payoff of the additional effort required for modifying the parent app is the ability to validate performance over a considerably larger design space at a very low cost. In addition to low-cost validation of parent app, our proposed approach can also be used for predicting performance of the parent app with BE simulations of notional architectures as the source of LF data for developing the MFS. The HF data for notional architectures could be obtained from fine-grained simulators over a small subset of design-space. BE's ability to simulate any hardware through an architecture model, whether existing or notional, adds an additional capability of predicting performance of the parent app on the future systems quantitatively at a low cost.

In this paper, we leverage many of the MFS methods developed in other scientific domains and adapt and apply them to reduce the computational cost of validation of representative apps used in the HPC co-design process. After a survey of the related research in Sect. 2, in Sect. 3 we present an overview of the parent application case study (CMT-nek), its representative mini-app (CMT-bone) and skeleton app (CMT-bone-BE), and the Behavioral Emulation approach that we use for performance modeling and simulation. In Sects. 4 and 5, we describe a methodology for developing an MFS for an application from its mini-app (HFM), skeleton app (LFM), and a simulator model (LFM). In Sect. 6, we demonstrate the usefulness of the proposed methodology by applying it to a multi-physics simulation application being developed for exascale systems - CMT-nek [1]. The results demonstrate MFS as a promising approach for predicting parent application performance (HF model) from representative app or architecture simulation (LF model) while staying within a reasonable computational budget.

## 2 Related Research

Mini-apps have become extremely important for exascale DSE and performance optimization. In [9], which presents a validation methodology, the authors state that mini-apps reduce the DSE time by a factor of a thousand, making them extremely useful for exploring the design space of the parent application. Dosanjh, et al. in [10] provide a verification and validation (V&V) methodology for assessing the ability of the mini-app to effectively represent the performance of their parent application. The authors use the difference between mini-app and parent app performance as their validation metric and compare it against a threshold. This approach requires equal number of samples for both the parent app and the mini-app. Since samples for the parent app are typically more expensive to obtain, it can be a limiting factor in extensive validation studies over a large design space. In our proposed approach, constructing an MFS requires much fewer samples of parent app than of the mini-app, thus considerably reducing the computational budget of conducting performance validation.

Mini-apps are used as a tool to evaluate optimization methods to improve the performance of the parent application. But improving the performance of

the mini-app does not guarantee the same for the parent app, making it important to know how representative these mini-apps are of their parent app [13]. For example, in [13] the authors seek to improve the performance of the application on new and future systems using mini-apps. Although the optimizations applied improve the mini-app performance, the impact on actual application performance is not clear. In our work, an MFS for the parent application, built using high-fidelity application performance samples and lower-fidelity mini-app performance samples, can help us draw a relationship between the performance behavior of the two applications.

Several frameworks have been proposed to realize multi-fidelity modeling in various science and engineering domains [12,17,25]. In [17], a Bayesian framework has been applied to predict the data of nuclear radiation based on simulations. A variable fidelity optimization framework has been demonstrated for the design of engine piston [23]. In [25], a deterministic framework has been proposed to predict the strength of composite laminate based on finite element simulations. Large number of application of MFS to mechanical systems can be found in [12], which reports that the MFS reduces computational cost drastically while enabling desirable prediction accuracy. The MFS has also been adopted as a powerful tool for uncertainty propagation [21].

Various MFS frameworks have been proposed for different engineering applications. For example, the Bayesian MFS based on a scale factor has been applied to design buildings [11] and flapping flight [26]. The Bayesian MFS incorporating discrepancy function has been proposed [17,22] as a popular MFS framework for various applications. This Bayesian framework is equivalent to the co-Kriging surrogate [20] with no prior information. Balabanov et al. [4] used a sequential deterministic MFS based on the discrepancy function to combine finite element simulation with different resolutions. Zhang et al. [24,25] proposed a simultaneous deterministic MFS based on the discrepancy function to combine experimental strength and finite element simulation for composite laminate. We can apply this methodology in our HPC community to save computational cost of parent application.

Sampling schemes for multi-fidelity models have been studied correspondingly. HF samples are usually a subset of LF samples. One representative all-at-once sampling strategy is the nested design sampling [15]. First, LF samples are generated using Latin Hypercube Sampling (LHS). Then the HF samples are generated by maximizing the minimum distance between all existing LF samples. Huang et al. [14] proposed a sequential sampling scheme for design optimization using Bayesian MFS. Either LF or HF samples are generated iteratively for design optimization. In our approach, we used Full Factorial Design (FFD) [8] for sampling as it is convenient for parametric study.

In this paper, we leverage many of the MFS methods developed in other scientific domains and adapt and apply them to the HPC domain to reduce the computational cost of validation of representative apps used in the co-design process.

## 3 Application and Architecture Models

In this section, we give an overview of the parent application under study (CMT-nek), its representative mini-app (CMT-bone) and skeleton app (CMT-bone-BE); and a BE simulation approach that we use for performance modeling and simulation. The relationships among their corresponding models are shown in Fig. 1. The parent app, CMT-nek, represents a high-fidelity (HF) model, whereas CMT-bone (a mini-app) and CMT-bone-BE (a skeleton app) are low-fidelity (LF) models, as compared to CMT-nek. BE simulation is a modeling and simulation of CMT-bone-BE. Thus, BE simulation is an even lower fidelity model than CMT-bone-BE and CMT-bone.

In this study, our objective is to first perform validation and uncertainty estimation of the BE simulation results against test samples of CMT-bone-BE (details in Sect. 5). Then a multi-fidelity surrogate model (MFS) is developed using mostly samples from the low-fidelity BE simulation and a few high-fidelity CMT-nek samples. The MFS model is then used to predict CMT-nek results (Sect. 6, case study 1). This experiment is repeated between BE simulation (LF) with CMT-bone (now being a HF, as compared to BE simulation) in case study 2, followed by CMT-nek (HF) and CMT-bone (LF) for case study 3.

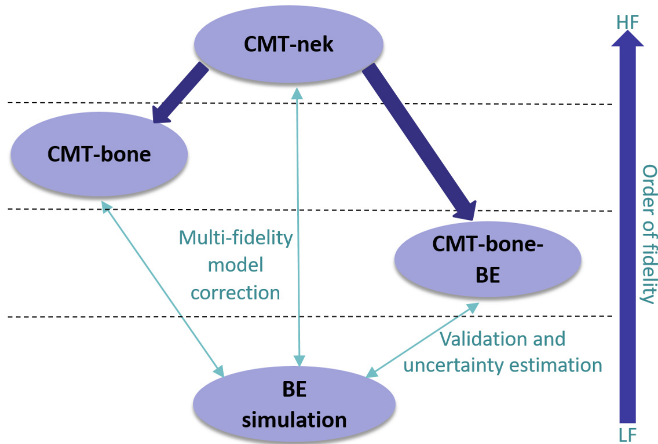



**Fig. 1.** Hierarchy of the CMT models

### 3.1 CMT-nek

CMT-nek [1] is being developed at the PSAAP-II Center for Compressible Multiphase Turbulence (CCMT) at University of Florida to perform simulation of instabilities, turbulence, and mixing in particulate-laden flows under conditions of extreme pressure and temperature [1]. CMT has applications in many environmental, industrial, and national defense and security areas. CMT-nek is being

developed from a production release of petascale code Nek5000 [2], a Gordon Bell prize winning open-source software for simulating unsteady incompressible fluid flow with thermal and passive scalar transport. It is a highly scalable code with strong scaling to over a million MPI ranks on ALCF BG/Q Mira. CMT-nek aims to take advantage of this sustained performance by inheriting the MPI strategies used in Nek5000; and by hooking into the Nek5000 repository, leveraging any changes and optimizations made to Nek5000.

### 3.2 CMT-bone and CMT-bone-BE

CMT-bone is a mini-app that encapsulates the key data structures and compute and communication kernels of CMT-nek. While retaining the workflow of CMT-nek, CMT-bone simplifies the number of variables defined and allocated and also the number of computation and communication operations performed at each time step in the simulation. The authors in [5,18] have validated mini-app CMT-bone with its parent app CMT-nek and found that the key compute kernels are well represented by the proxy application.

CMT-bone-BE is a skeleton app of CMT-nek created to support rapid algorithmic design-space exploration. It models the computation that happens within every simulation timestep to calculate the partial derivative and exchange data between nearby spectral element meshes. CMT-bone-BE ignores the initial problem setup including mesh generation. Mesh generation operations can be abstracted and replaced with a computation model in BE.

### 3.3 Behavioral Emulation (BE) Simulation

Behavioral Emulation (BE) is a coarse-grained modeling and simulation approach that aims to provide timely, flexible, and scalable estimates of application performance on existing and future system architectures. In BE, the complexity of large-scale system simulation is handled by simultaneously dividing the simulation into different levels of system abstraction (e.g., device, node, rack, system) and abstracting the behavior of the components at each of these levels. The coarse-grained component models mimic or emulate the observed execution behavior of the component instead of its cycle-accurate operation. There are two basic types of BE models - application BE objects (AppBEOs) and architecture BE objects (ArchBEOs).

BE simulations are used to predict the execution time of CMT-bone-BE and have computational cost less than that of CMT-bone-BE (and much less than that of CMT-nek and CMT-bone). In our study, BE simulation results are used to produce the LF data points which are used to construct MFS models to predict the execution time of CMT-nek and CMT-bone, using very few data points from these two applications, thus reducing the overall computational budget of the DSE process.

## 4 Multi-fidelity Surrogates

In engineering applications, it is common to have multiple models with different fidelities for solving the same problem such as finite element simulations with different grid resolutions, numerical simulations, and physical experiments. A high-fidelity model (HFM) represents the physical phenomenon more accurately than the low-fidelity model (LFM) but it is often very expensive. An MFS based approach uses both high-fidelity and low-fidelity datasets to approximate the HFM in the design space. An effective MFS is expected to make accurate prediction with limited budget for sampling. Fernández-Godino et al. [12] reviewed recent developments of MFS especially on effectiveness of applying MFS to practical design. Peherstorfer et al. [21] summarized the technical details of MFS for inference and uncertainty propagation.

MFS translates LFM against a few HF samples using an algebraic function. Typical MFS frameworks include two major components: (1) a model to define the relation between LFM and HFM, and (2) the scheme to find parameters of the MFS and associated uncertainty of prediction. The LFM could be translated to HFM through (1) a constant scale factor, or (2) scaling up the LFM and adding to a discrepancy function. After determining the form of algebraic function, the MFS could be developed either using Bayesian inference through Gaussian process, or using a least-square regression minimizing error between fitted model and data.

In this work, we investigate the feasibility of MFS to quantify and mitigate the difference between high-fidelity parent applications (e.g., CMT-nek) and low-fidelity simulations (e.g., BE simulation) in the area of co-design of large-scale system. The least-squares MFS (LS-MFS) [24] was selected for this feasibility study while balancing complexity and predictive capability.

$$\hat{f}_H(\boldsymbol{x}) = \rho \hat{f}_L(\boldsymbol{x}) + \hat{\delta}(\boldsymbol{x}) \tag{1}$$

The LS-MFS is built with two surrogates, $\hat{f}_L(\boldsymbol{x})$, a polynomial response surface (PRS) fitted to low-fidelity data, and $\hat{\delta}(\boldsymbol{x})$, the fitted discrepancy data (Eq. 1). The scale factor $\rho$ and discrepancy function $\hat{\delta}(x)$ are obtained to minimize prediction error at the high-fidelity samples according to Eqs. 2 and 3. $(\boldsymbol{x}_H, \boldsymbol{y}_H)$ denotes the high-fidelity dataset containing $n$ samples.

$$\min_{\rho, \hat{\delta}(x)} : (\hat{\delta}(\boldsymbol{x}_H) - \boldsymbol{d}_H)^T (\hat{\delta}(\boldsymbol{x}_H) - \boldsymbol{d}_H) \tag{2}$$

$$\boldsymbol{d}_H = \rho \hat{f}_L(\boldsymbol{x}_H) - \boldsymbol{y}_H \tag{3}$$

The multi-fidelity surrogate using a single linear regression is obtained from Eqs. (4–7). $\boldsymbol{Y}$ is the vector of high-fidelity samples, $\boldsymbol{X}$ is the augmented design matrix, $\boldsymbol{B}$ is the vector of unknown coefficients and $\boldsymbol{e}$ is the vector for residual errors. $\boldsymbol{X}_i(\boldsymbol{x})$ denotes the $i^{th}$ monomial/basis, and $b_i$ is the coefficient of $\boldsymbol{X}_i(\boldsymbol{x})$. The obtained discrepancy function $\hat{\delta}(\mathbf{x})$ is shown in Eq. 8.

$$\mathbf{Y} = \mathbf{XB} + \mathbf{e} \tag{4}$$

$$\mathbf{Y}_{n\times 1} = \begin{bmatrix} \mathbf{y}_H^{(1)} \\ \vdots \\ \mathbf{y}_H^{(n)} \end{bmatrix}, \mathbf{B}_{p+1} = \begin{bmatrix} \rho \\ b_1 \\ \vdots \\ b_p \end{bmatrix}, \mathbf{e}_{n\times 1} = \begin{bmatrix} \mathbf{e}_H^{(1)} \\ \vdots \\ \mathbf{e}_H^{(n)} \end{bmatrix} \tag{5}$$

$$\mathbf{X}_{n\times(p+1)} = \begin{bmatrix} \hat{f}_L(\mathbf{x}_H^{(1)}) & X_1(\mathbf{x}_H^{(1)}) & \dots & X_p(\mathbf{x}_H^{(1)}) \\ \vdots & \vdots & \ddots & \vdots \\ \hat{f}_L(\mathbf{x}_H^{(n)}) & X_1(\mathbf{x}_H^{(n)}) & \dots & X_p(\mathbf{x}_H^{(n)}) \end{bmatrix} \tag{6}$$

$$\mathbf{B} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y} \tag{7}$$

$$\hat{\delta}(\mathbf{x}) = \sum_{i=1}^{p} b_i X_i(\mathbf{x}) \tag{8}$$

The LS-MFS scales up the LFM and adds a polynomial function $\hat{\delta}(\boldsymbol{x})$ to match a few high-fidelity samples. The scale factor $\rho$ is critical for prediction. Negative or extremely large values of $\rho$ indicates a prediction with large error, which is likely to be associated with undesirable LFMs, inappropriate surrogate forms, or inadequate samples. $\hat{\delta}(\boldsymbol{x})$ is supposed to be a low-order polynomial function while assuming the LFM has a trend similar to HFM. In our study, we adopted a constant $\hat{\delta}(\boldsymbol{x})$ for less than 10 HF samples and a linear polynomial function as $\hat{\delta}(\boldsymbol{x})$ for the rest. We approximated the execution time in logarithmic coordinate to account for the order-of-magnitude variation of execution time. $\hat{f}_L(\boldsymbol{x})$ was developed using a quartic PRS.

# 5 Developing MFS Model

Although various performance metrics can be studied for performance simulation such as energy consumption and communication times between the processors, the metric of interest in this paper is the total execution time for running a typical computational fluid dynamics analysis using CMT-nek (HFM), CMT-bone (HFM), CMT-bone BE (LFM), and BE simulation (LFM). All the benchmarking of the CMT models is performed on the Vulcan HPC platform from Lawrence Livermore National Laboratory (LLNL) [3]. Vulcan is a 24-rack IBM Blue Gene/Q system based on POWER architecture that consists of 24,576 nodes and 400 TB of compute memory. It is important to ensure that the HF and LF data are obtained from the same hardware. The accuracy of the MFS model reflects how representative the LF and HF are of each other.

## 5.1 Design Space

For CMT-nek, the three main application parameters of concern are Element Size (ES), Elements per Processor (EPP) and Number of Processors (NP). Application performance can be affected by changing any one of these parameters. We

chose 125 experimental points based on five-level full factorial design. "Five-level" denotes the 5 points/grids selected along each application parameter with similar space, as shown in Fig. 2. The design of experiment is ES = {5,9,13,17,21}, EPP = {8,32,64,128,256} and NP = {16,256,2048,16384,131072}. The experimental runs require up to 131,072 processors, 34 million elements and 311 billion computational grid points.

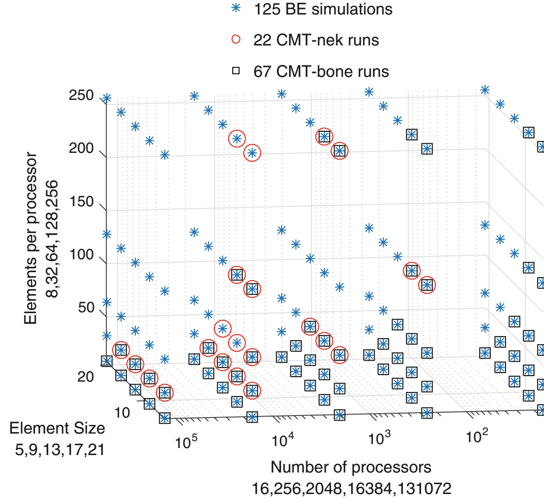



**Fig. 2.** Design of experiments for CMT-nek, CMT-bone, and BE simulations

As the model fidelity increases so does the cost of obtaining a test sample. We obtained data for CMT-bone-BE and BE simulation for the entire design space (125 data points); but for CMT-nek and CMT-bone, data was judiciously obtained from a subset of the design space. For the runs from LFM and HFM, we made 22 runs from CMT-nek (HFM), 67 runs from CMT-bone, and 125 runs for both LFMs (CMT-bone-BE and BE simulation).

### 5.2 Validations of BE Simulation Results

Recall from Fig. 1 that the order of fidelity is as follows: parent app CMT-nek (highest), mini-app CMT-bone, skeleton app CMT-bone-BE, and BE simulation (lowest). In the next section, we will use the BE-simulation results (LF) to predict the performance of CMT-nek (HF parent app). Thus, first it is important to evaluate the accuracy of the BE simulation. To do so, in this section, we will first validate the accuracy of BE simulation against skeleton app CMT-bone-BE. We then evaluate the accuracy of CMT-bone-BE by validating its results against those of mini-app CMT-bone.

**BE simulation vs. CMT-bone-BE.** Validation is the process of comparing the BE simulation results to its respective benchmarking result using CMT-bone-BE. In this study, we have validated the simulation results for the entire design space on Vulcan, one of the largest high-performance computing system available at the Lawrence Livermore National Lab. The validation of the design space covered all the calibration points. But to further evaluate the accuracy of the simulator, true validation was performed by validating points that are not present in the calibration set. Polynomial interpolation was used in the simulator to predict the execution time of the application at these validation points. The obtained simulation results are then validated by running the actual CMT-bone-BE application on Vulcan for those validation points.

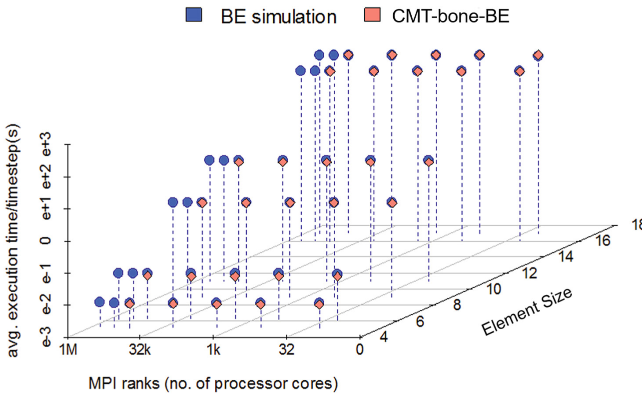
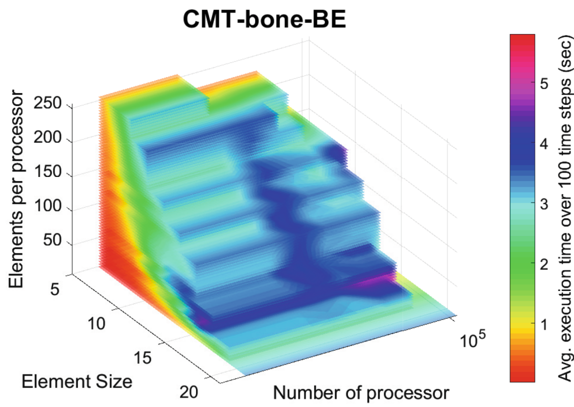


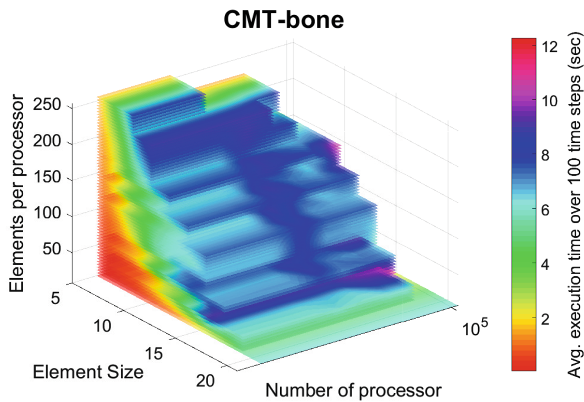

**Fig. 3.** Validation of BE simulation against CMT-bone-BE (Color figure online)

The validation results of BE simulation against CMT-bone-BE are shown in Fig. 3 where the blue points represent the predicted CMT-bone-BE time using BE simulation and the red points are the validation points obtained by running the application (CMT-bone-BE) on Vulcan. We validated the simulations up to 128k NP on Vulcan and predicted the time for 256k NP and 512k NP. The average percentage error between BE simulation and CMT-bone-BE is 4%, thus demonstrating the accuracy of the BE simulator.

**CMT-bone-BE vs. CMT-bone.** The validation of the skeleton app CMT-bone-BE against mini-app CMT-bone (Fig. 4a and b, respectively) is done through comparing their trends under the same experimental setup described above. CMT-bone-BE being the skeleton app, takes less time to execute than that of the mini-app, CMT-bone, and hence the range of their execution time varies. Therefore, to make it easier to compare the trend, the execution time is plotted on a color scale with red being the lowest in the range and blue being highest in the range as shown in Fig. 4. The step-wise increase shows that the predicted execution time for CMT-bone-BE and CMT-bone increases monotonically with ES and EPP and does not




(a)




(b)

**Fig. 4.** Comparing CMT-bone mini-app and CMT-bone-BE skeleton app trends for various parameter values (Color figure online)

change appreciably with NP, and the color scale on the graphs verify the similarity in trend between CMT-bone-BE and CMT-bone.

## 6 Evaluating MFS Predictions — Three Case Studies

Three case studies were used to demonstrate the multi-fidelity surrogate (MFS) approach in which a surrogate model, based mostly on low-fidelity (LF) samples assisted with only a few high-fidelity (HF) samples, is used to predict the performance of a high-fidelity (HF).

- Case 1: Multi-fidelity model based mostly on BE simulation (LF) and few CMT-nek (HF parent app) data points to predict the performance of CMT-nek (HF)
- Case 2: Multi-fidelity model based mostly on BE simulation (LF) and few CMT-bone (relatively HF mini-app) data points to predict the performance of CMT-bone (HF)
- Case 3: Multi-fidelity model based mostly on CMT-bone (relatively LF mini-app) and few CMT-nek (HF) data points to predict the performance of CMT-nek (HF)

The setup was same in all three case studies. A subset of high-fidelity data was selected as the validation/test runs to evaluate predictions while the others were used as training runs to train LS-MFS (least square MFS). The number of samples increased gradually from the remaining runs (which excludes the validation runs) to investigate the effect of sampling plan. For each number of samples, random selection was repeated 20 times to account for the effect of sampling plan. The overall difference was measured using relative root-mean-square error (R-RMSE) between the LS-MFS predictions and the validation runs. The relative maximum difference (R-MD) at the validation runs based on the repeated samples was also provided to understand individual prediction. In this paper, we study LS-MFS using polynomial response surface as it is robust with noise effect. Other frameworks of multi-fidelity surrogates are also available such as co-Kriging. The comparison between different multi-fidelity surrogates is beyond the scope of this paper.

### 6.1 Case Study 1: CMT-nek Predictions from BE Simulations

22 runs of CMT-nek are obtained as shown in Fig. 2. 10 runs (out of 22) were selected randomly and fixed as the validation runs. We first examined LS-MFS to approximate CMT-nek (HF model) runs using a typical set of 12 samples

**Table 1.** Predicting execution times of CMT models based on typical set of 12 samples and evaluating the prediction using R-RMSE (%)

| | Case 1: CMT-nek prediction from BE simulation | Case 2: CMT-bone prediction from BE simulation | Case 3: CMT-nek prediction from CMT-bone |
|---|---|---|---|
| Number of validation runs | 10 | 20 | 10 |
| LS-MFS | 4.49% | 5.40% | 7.34% |
| $\hat{f}_L(\boldsymbol{x})$ | 66.85% | 61.26% | 18.65% |
| Linear fit to $(\boldsymbol{x}_H, \boldsymbol{y}_H)$ | 131.23% | 1901.91% | 131.23% |
| Residual errors of $\hat{f}_L(\boldsymbol{x})$ | 0.77% | 0.77% | 1.05% |
| Residual errors of the linear fit to $(\boldsymbol{x}_H, \boldsymbol{y}_H)$ | 18.40% | 50.71% | 18.40% |

as shown in the Case 1 column of Table 1. The R-RMSE of LS-MFS is 4.49% using BE simulation (LF model) at 10 validation runs. The linear fit using only HF CMT-nek runs was also developed as a comparison with the R-RMSE to be 131.23% at the validation runs. The R-RMSE between original BE simulation and CMT-nek ($\hat{f}_L(\boldsymbol{x})$) at all the 12 points, without translation, is 66.85%. As mentioned before, BE simulation mimics CMT-bone-BE and not CMT-nek. Since CMT-bone-BE is just a skeleton app with very few computational kernels, the percentage difference is high. The LS-MFS was much more accurate than either the $\hat{f}_L(\boldsymbol{x})$ or the linear fit to $(\boldsymbol{x}_H, \boldsymbol{y}_H)$, demonstrating its promise to compensate the difference between high-fidelity and low-fidelity models.

Next, we investigated the effect of the sampling plan on prediction accuracy. The LS-MFS predictions for CMT-nek runs with increasing number of samples were summarized in Fig. 5a. The LS-MFS was unstable using only 2 CMT-nek samples due to over-fitting and became more accurate with increasing CMT-nek samples. The R-RMSE was less than 10% with more than 9 CMT-nek samples and ended with 4.49%. The R-MD was less than 20% with more than 9 CMT-nek samples and ended with 7% as seen in Fig. 5b. The order of $\hat{\delta}(\boldsymbol{x})$ was changed from constant to linear for more than 9 CMT-nek samples which was critical for the accuracy of LS-MFS. We specified the order of $\hat{\delta}(\boldsymbol{x})$ for simplicity in this feasibility study. The performance of LS-MFS could be improved by choosing appropriate $\hat{\delta}(\boldsymbol{x})$. Another observation is the large variation of R-RMSE while repeating HF samples. The design of experiments for HF samples affected LS-MFS noticeably. The evaluation of LS-MFS for CMT-nek is based on up to 12 samples and may suffer large uncertainty due to the scarce runs.

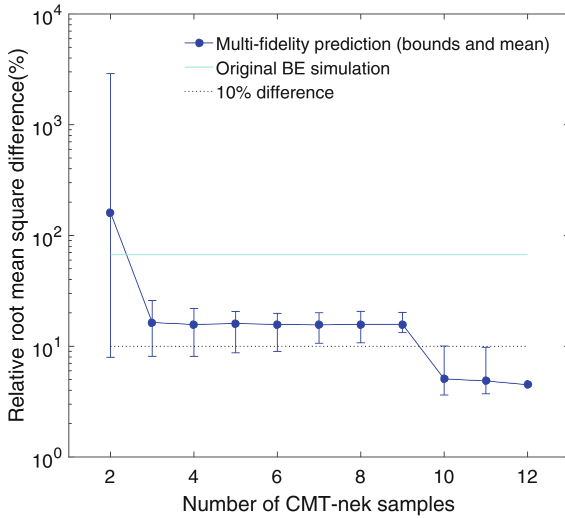
### 6.2 Case Study 2: CMT-bone Predictions from BE Simulations

20 runs (out of 67) were selected randomly and fixed as the validation runs. We performed LS-MFS for CMT-bone (HF model in this case) based on 20 validation runs and up to 47 samples. Again, LS-MFS was most accurate comparing to $\hat{f}_L(\boldsymbol{x})$ and the linear fit to only $(\boldsymbol{x}_H, \boldsymbol{y}_H)$ as shown in Case 2 column of Table 1. The LS-MFS predictions were much closer to HFM than the original BE simulations.

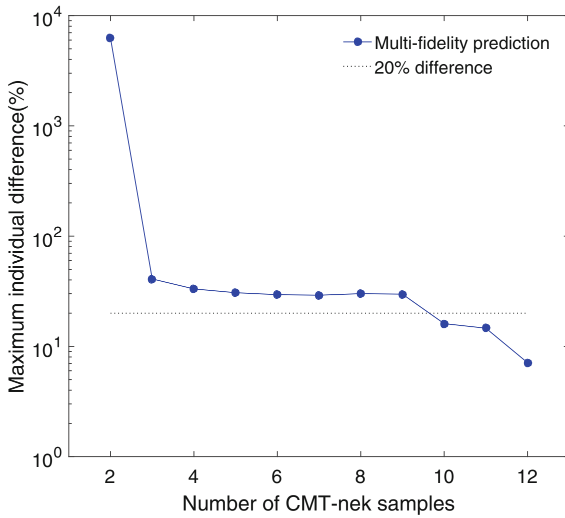
Once again, we investigate the effect of the sampling plan on prediction accuracy. The R-RMSE in Fig. 6a reduced with increasing CMT-bone samples and ended with 5.4%. The R-MD in Fig. 6b oscillated with scarce CMT-bone runs at the beginning and stabilized around 10%. Both R-RMSE and R-MD reduced noticeably with the first few samples and stabilized to less than 10% thus proving to be a promising approach.

### 6.3 Case Study 3: CMT-nek Predictions from CMT-bone

In the final case study, the LS-MFS was developed to predict the high-fidelity parent app (CMT-nek) from its low-fidelity mini-app (CMT-bone). This helps in quantitative validation of the mini-app. The setup was same as in case study 1, where CMT-nek was the HF model. From Case 3 column of Table 1, we see



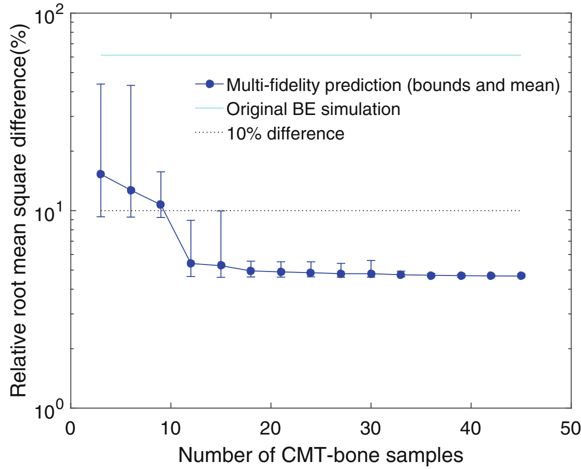

(a) R-RMSE



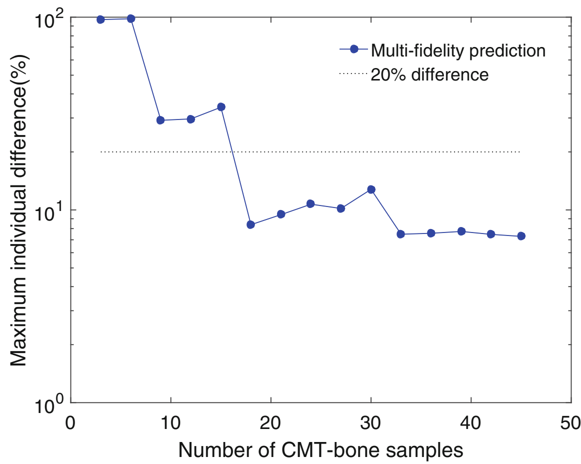

(b) R-MD

**Fig. 5.** Difference between CMT-nek validation runs and multi-fidelity predictions based on BE simulation

that LS-MFS provides the best fit compared to linear fit. The LS-MFS had less-than 10% R-RMSE in Fig. 7a. It is worth noting the significant jump between 9 and 10 samples while changing the order of $\hat{\delta}(\boldsymbol{x})$ in Fig. 7a and b. CMT-bone was close to CMT-nek and a different scheme might be preferred to determine $\hat{\delta}(\boldsymbol{x})$.



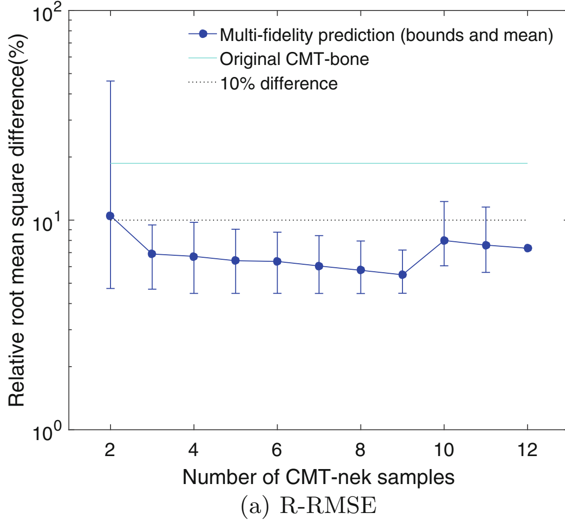

(a) R-RMSE



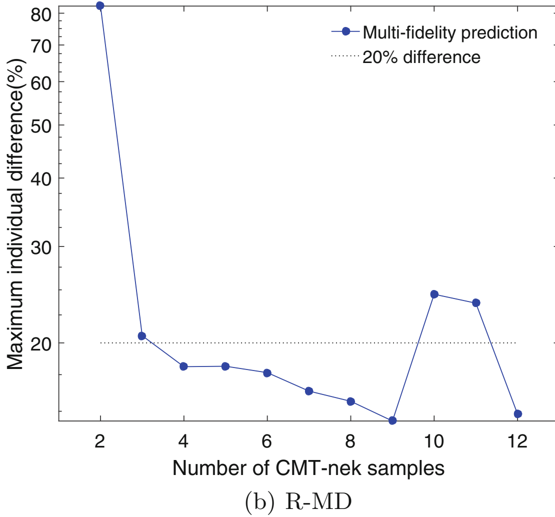

(b) R-MD

**Fig. 6.** Difference between CMT-bone validation runs and multi-fidelity predictions based on BE simulation

A key observation between case study 1 and 3 is that although the CMT-bone samples were much closer to the CMT-nek samples, the MFS predictions of CMT-nek (HF) from BE simulations (LF) were more accurate than the MFS predictions from CMT-bone (LF) as shown in Table 1. Fitting CMT-bone was more challenging considering the scarce samples (67 runs). BE simulations, on the other hand, had all the 125 samples in the design space and thus, lead to better MFS predictions. This is supported by residual errors of $\hat{f}_L(\boldsymbol{x})$ from Table 1.




(a) R-RMSE




(b) R-MD

**Fig. 7.** Difference between CMT-nek validation runs and multi-fidelity predictions based on CMT-bone

In all three cases, the error in prediction ended less than 10%; thus, proving it to be a valuable approach to use for reducing computational budget in the process of co-design. The range of scale factor ($\rho$) for the three cases are summarized in Table 2. The scale factors are around 1 which indicates that the LF and HF have similar trend. The major difference between HF and LF are well-compensated by the constant/linear discrepancy function.

**Table 2.** Range of the scale factors for LS-MFS

| | Case 1 | Case 2 | Case 3 |
|---|---|---|---|
| Minimum $\rho$ | 0.8 | 0.91 | 0.5 |
| Maximum $\rho$ | 0.98 | 1.08 | 1.3 |

## 7 Conclusions

Due to high computational cost, validation samples from HFM are usually obtained at small scale. But with MFS model, we were able to perform quantitative validation at a reduced computational budget. In this paper, we studied the least-square MFS (LS-MFS) using polynomial response surface as it is robust with noise effect. For future work, different multi-fidelity surrogates can be compared. Our ultimate goal is to predict the performance for exascale computation platform which is essentially long-range extrapolation far from the validation samples. In the future, we will investigate the capability of LS-MFS for long-range extrapolation which suffers large uncertainty. We also noticed the LS-MFS predictions were sensitive with the high-fidelity samples. Effective design of experiments for validation runs are expected to improve the accuracy of LS-MFS, which is also a valuable research direction.

**Acknowledgment.** This work is supported by the U.S. Department of Energy, National Nuclear Security Administration, Advanced Simulation and Computing Program, as a Cooperative Agreement under the Predictive Science Academic Alliance Program, under Contract No. DE-NA0002378.

## References

1. Center for Compressible Multiphase Turbulence webpage, 15 February 2015. https://www.eng.ufl.edu/ccmt/
2. NEK5000 webpage. https://nek5000.mcs.anl.gov/
3. Vulcan Supercomputer, LLNL webpage. https://computation.llnl.gov/computers/vulcan
4. Balabanov, V., Grossman, B., Watson, L., Mason, W., Haftka, R.: Multifidelity response surface model for HSCT wing bending material weight. In: 7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, p. 4804 (1998)
5. Banerjee, T., Hackl, J., Shringarpure, M., Islam, T., Balachandar, S., Jackson, T., Ranka, S.: CMT-bone: a proxy application for compressible multiphase turbulent flows. In: 2016 IEEE 23rd International Conference on High Performance Computing (HiPC), pp. 173–182, December 2016
6. Barrett, R.F., Borkar, S., Dosanjh, S.S., Hammond, S.D., Heroux, M.A., Hu, X.S., Luitjens, J., Parker, S.G., Shalf, J., Tang, L.: On the role of co-design in high performance computing (2013)
7. Barrett, R.F., Vaughan, C.T., Heroux, M.A.: Minighost: a miniapp for exploring boundary exchange strategies using stencil computations in scientific parallel computing. Sandia National Laboratories, Technical report SAND 5294832 (2011)

8. Box, G.E.P., Hunter, J.S., Hunter, W.G.: Statistics for Experimenters: Design, Innovation, and Discovery, vol. 2. Wiley-Interscience, New York (2005)
9. Dosanjh, S.S., Barrett, R.F., Doerfler, D., Hammond, S.D., Hemmert, K.S., Heroux, M.A., Lin, P.T., Pedretti, K.T., Rodrigues, A.F., Trucano, T.: Exascale design space exploration and co-design. Future Gener. Comput. Syst. **30**, 46–58 (2014)
10. Dosanjh, S.S., Barrett, R.F., Doerfler, D., Hammond, S.D., Hemmert, K.S., Heroux, M.A., Lin, P.T., Pedretti, K.T., Rodrigues, A.F., Trucano, T., et al.: Assessing the role of mini-applications in predicting key performance characteristics of scientific and engineering applications. J. Parallel Distrib. Comput. **30**, 107–122 (2014)
11. Ellis, M., Mathews, E.: A new simplified thermal design tool for architects. Build. Environ. **36**, 1009–1021 (2011)
12. Fernández-Godino, M.G., Park, C., Kim, N.H., Haftka, R.T.: Review of multi-fidelity models. arXiv preprint arXiv:1609.07196 (2016)
13. Heroux, M.A., Doerfler, D.W., Crozier, P.S., Willenbring, J.M., Edwards, H.C., Williams, A., Rajan, M., Keiter, E.R., Thornquist, H.K., Numrich, R.W.: Improving performance via mini-applications. Sandia National Laboratories, Technical report SAND2009-5574 3 (2009)
14. Huang, D., Allen, T., Notz, W., Miller, R.: Sequential kriging optimization using multiple-fidelity evaluations. Struct. Multi. Optim. **32**, 369–382 (2006)
15. Jin, R., Chen, W., Sudjianto, A.: An efficient algorithm for constructing optimal design of computer experiments. J. Stat. Plan. Infer. **134**, 268–287 (2005)
16. Karlin, I., Keasler, J., Neely, R.: Lulesh 2.0 updates and changes. Technical report LLNL-TR-641973 (2013)
17. Kennedy, M.C., O'Hagan, A.: Bayesian calibration of computer models. J. Roy. Stat. Soc. Ser. B (Stat. Methodol.) **63**(3), 425–464 (2001)
18. Kumar, N., Sringarpure, M., Banerjee, T., Hackl, J., Balachandar, S., Lam, H., George, A., Ranka, S.: CMT-bone: a mini-app for compressible multiphase turbulence simulation software. In: 2015 IEEE International Conference on Cluster Computing, pp. 785–792, September 2015
19. Kumar, N., Pascoe, C., Hajas, C., Lam, H., Stitt, G., George, A.: Behavioral emulation for scalable design-space exploration of algorithms and architectures. In: Taufer, M., Mohr, B., Kunkel, J.M. (eds.) ISC High Performance 2016. LNCS, vol. 9945, pp. 5–17. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46079-6_1
20. Le Gratiet, L.: Multi-fidelity Gaussian process regression for computer experiments. Universit Paris-Diderot-Paris VII (2013)
21. Peherstorfer, B., Willcox, K., Gunzburger, M.: Survey of multifidelity methods in uncertainty propagation, inference, and optimization (2016)
22. Qian, P.Z., Wu, C.J.: Bayesian hierarchical modeling for integrating low-accuracy and high-accuracy experiments. Technometrics **50**, 192–204 (2008)
23. Xiong, Y., Chen, W., Tsui, K.L.: A new variable-fidelity optimization framework based on model fusion and objective-oriented sequential sampling. J. Mech. Des. **130**(11), 111401 (2008)
24. Zhang, Y., Kim, N.H., Park, C., Haftka, R.T.: Multi-fidelity surrogate based on single linear regression. ArXiv e-prints arXiv:1705.02956 (2017)
25. Zhang, Y., Meeker, J., Schutte, J., Kim, N., Haftka, R.: On approaches to combine experimental strength and simulation with application to open-hole-tension configuration. In: Proceedings of the American Society for Composites: Thirty-First Technical Conference (2016)
26. Zheng, L., Hendrick, T.L., Mittal, R.: A multi-fidelity modelling approach for evaluation and optimization of wing stroke aerodynamics in flapping flight. J. Fluid Mech. **721**, 118–154 (2013)

# A Slurm Simulator: Implementation and Parametric Analysis

Nikolay A. Simakov(✉), Martins D. Innus, Matthew D. Jones, Robert L. DeLeon, Joseph P. White, Steven M. Gallo, Abani K. Patra, and Thomas R. Furlani

Center for Computational Research, State University of New York, University at Buffalo, Buffalo, NY, USA
nikolays@buffalo.edu

**Abstract.** Slurm is an open-source resource manager for HPC that provides high configurability for inhomogeneous resources and job scheduling. Various Slurm parametric settings can significantly influence HPC resource utilization and job wait time, however in many cases it is hard to judge how these options will affect the overall HPC resource performance. The Slurm simulator can be a very helpful tool to aid parameter selection for a particular HPC resource. Here, we report our implementation of a Slurm simulator and the impact of parameter choice on HPC resource performance. The simulator is based on a real Slurm instance with modifications to allow simulation of historical jobs and to improve the simulation speed. The simulator speed heavily depends on job composition, HPC resource size and Slurm configuration. For an 8000 cores heterogeneous cluster, we achieve about 100 times acceleration, e.g. 20 days can be simulated in 5 h. Several parameters affecting job placement were studied. Disabling node sharing on our 8000 core cluster showed a 45% increase in the time needed to complete the same workload. For a large system (>6000 nodes) comprised of two distinct sub-clusters, two separate Slurm controllers and adding node sharing can cut waiting times nearly in half.

**Keywords:** HPC · SLURM · Batch jobs scheduler · Simulator

## 1 Introduction

Different fields of science and engineering exhibit different demands on computational resources. Responding to that, modern HPC clusters have significantly heterogeneous architecture, where in addition to traditional computational nodes a number of specialized nodes can be present including high memory nodes, GPU or MIC accelerated nodes and fast/large local file storage nodes. In addition, many HPC centers have various generations of these nodes operational at the same time. Managing such a facility efficiently can be challenging. Previously, centers would have a separate

**Electronic supplementary material** The online version of this chapter (https://doi.org/10.1007/978-3-319-72971-8_10) contains supplementary material, which is available to authorized users.

© Springer International Publishing AG 2018
S. Jarvis et al. (Eds.): PMBS 2017, LNCS 10724, pp. 197–217, 2018.
https://doi.org/10.1007/978-3-319-72971-8_10

scheduler for each resource type, limiting users to use a specific resource even if multiple resources could serve the user request. This often led to resource imbalances where some resources had large queues while other were almost idle. To overcome this problem, Slurm workload manager [1, 14, 17] can manage all these resources under a single controller. Slurm is an open-source HPC resource manager used in a large number of HPC centers, including those which include systems on the Top500 list. There are a number of ways by which Slurm can support heterogeneous resource organization. For example, GPU nodes can be organized as a separate partition or as a part of a general partition with higher priorities for jobs requesting GPUs. There are many other adjustable parameters which influence the scheduler and in many cases their overall effect on the system performance can far from obvious. Performing a Slurm parametric optimization on a live-production system can have undesirable consequences for the end-users. Therefore, the ability to execute Slurm in a "simulated" mode, where the actual system is modelled and actual historical jobs are used as a probe can be very helpful to optimize job throughput at an HPC center.

Besides mentioned aid in organizing heterogeneous components of a cluster, the Slurm simulator can be useful in number of other parameters optimization affecting the Slurm and system performance. These can include: identification of adequate priority boost for priority users or users under deadline, optimization of parameters affecting the work of main and backfill scheduler. The simulator can also be used for testing and development of scheduling related components of Slurm.

Here we report on our Slurm simulator which is a further development of the Slurm simulator done by Lucero [9] and by Trofinoff and Benini [16]. The Slurm simulator is based on the actual Slurm source code and thus can be used to study most of the Slurm parameters. Special attention was given to simulation speed and the capability to simulate historical job load on medium and large clusters. We performed validation and analysis of the effect of several scheduler parameters on the HPC cluster performance.

## 2 Related Work

There are a number of job scheduler simulators for traditional HPC and Grid resources. Among them there are Bricks [15], SimGrid [8], Simbatch [3], GridSim [4] and Alea [7]. While providing a general framework for studying scheduling strategies they are only of limited interest for HPC centers looking to make specific changes to optimize their scheduling policies. Maui and Moab Scheduler has a built in scheduler simulator, however it also provides limited help for Slurm users as well [6, 10, 11].

One way to simulate the full workloads of a real scheduler is by scaling the actual job wall time and submission time. However, Slurm has many time-dependent qualities like discrete execution of schedulers (main and backfill) and priorities calculations. Therefore, conclusions drawn from such a scaled simulation may not be strictly applicable to the non-scaled system.

The Slurm Simulator originally developed by Alejandro Lucero [9] and improved by Trofinoff and Benini [16] is based on the actual Slurm source code and has the potential to be helpful for Slurm parametric optimization. Both schedules are capable of simulation of small clusters possibly with realistic and long workloads. However, our

experience with them shows a low simulation speed for small clusters and an inability to handle mid-sized clusters with a realistic multiweek load. In addition, they are also based on an older version of Slurm.

## 3 Implementation Details

In this section, we describe modifications made to Slurm to allow its usage as its own simulator. A brief description of Slurm is given to help aid in understanding some of the concepts appearing later in the paper. For a detailed description refer to Slurm documentation [14] and the original articles [1, 17].

Slurm has a fault tolerant, multi-daemon and multi-thread design. The main daemon is the Slurm controller daemon (slurmctrld). It manages resources and allocates work on them. It also receives and services request calls from many Slurm utilities (for example squeue, sinfo and sbatch). Slurmd is a communication daemon running on every node managed by Slurm. It starts and finishes the user's jobs, per controller request, on that node and performs other node operations like execution of prolog and epilog. Slurmdbd is used for users accounts storage and is important for access rights and historical usage for fair-share calculation. Slurm control daemon (slurmctrld) has a multi-thread design where the most crucial parts are executed as separate threads with a number of health monitoring threads. For the simulation, the most important threads are ones which periodically execute the main scheduler, backfill scheduler, database synchronization and priority decay calculation. For every allocated new job Slurm spins-off a separate thread to initiate the user's job on the designated resources. A separate thread is also spun-off for each retiring job. This multi-thread design is used to achieve a high fault tolerance.

Although this multi-daemon, multi-thread design is good for the intended Slurm utilization, it has a drawback for simulation since it significantly affects performance due to the need for synchronization. For the simulator, there is no need for such a high tolerance but there is a need to perform a simulation in a reasonable amount of time. Previous Slurm simulators [9, 16] continued to use this feature of Slurm and maintain the overall Slurm workflow. The previous simulators were able to simulate small clusters with reasonable speed but had a hard time simulating real mid-size clusters. Although they occasionally were able to simulate a 256 nodes homogeneous cluster with a simulation speed of 10 simulated days per day, in many cases the simulator stalled at a certain point of the simulation without finishing it. The problem is the large number of threads which were spun-off to serve new job allocations and deallocation of resources from retired jobs. In the simulation the number of threads has a higher probability to accumulate than it does in real time due to time acceleration in the simulation. The unserved threads therefore tend to pile up and effectively hang the main process. In Slurm, it is possible to limit the thread counts, however, the yet-unspun threads will be accumulated in a special agent queue for later execution, which leads to similar problems and delays in job actual starting time and resource deallocation. To overcome this issue and obtain a higher simulation speed, we minimized the number of daemons and threads in our simulator.

Similar to previous implementations, our simulator was also developed with actual Slurm source code (a forked version of the original Slurm). The simulator was implemented as separate conditionally compiled source files with a relatively small number of modifications to the main Slurm code, allowing compilation of the code in normal and simulated modes. This layout in conjunction with Git tracking and merging capabilities would simplify simulator porting to newer Slurm versions. Unlike previous implementations, we didn't attempt to keep the entire Slurm functionality but used only the bare minimum necessary for a simulation and did not introduce an external simulation controller.

In this implementation the Slurm controller daemon, slurmctld, in addition to resource management and job scheduling, also controls the simulation. The controller daemon was serialized: all threads that are not crucial for simulation remain unstarted while the functions of crucial ones are called from the simulator main event loop in a serial way. In the simulation mode, instead of entering the main thread loop, slurmctrld enters the simulator main event loop where it remains until the end of the simulation. The simulator main event loop replaces the functionality of all crucial threads of normal Slurm and controls the execution of batch job submission, scheduling (both main and backfill), jobs priority decay calculation, synchronization with slurmdbd, jobs allocation and deallocation. The calls to respective Slurm functions are done within the loop based on expiration of their scheduled execution time, i.e. if the scheduled time is equal or less than the current simulated time. For periodically executed functions like the main and the backfill scheduler, the scheduled execution time is calculated as the previous execution time plus the sleeping time as defined in the respective Slurm agent thread. Scheduled execution time for batch job submission is an input parameter and job deallocation time is calculated as job start time plus job duration time which is also an input parameter. The simulator does not start a separate job launching and termination thread; instead it simulates the positive slurmd response which eliminates the need for the actual slurmd daemon. Thus, the simulator uses only two daemons: slurmctld and slurmdbd.

Even though slurmctrld in simulated mode is sequential, still the calls to mutex locks cost up to 40% of the total execution time, overwriting of the locking function with a dummy placeholder allows one to save that time. By default, Slurm is compiled in debug mode with assert enabled, since it is not crucial in simulation mode compilation without debugging and without asserts gains additional speed. The overall improvement in comparison with the real Slurm for the backfill scheduler is of factor of 10.

Because many Slurm functions rely on calls to time and gettimeofday functions, these functions were overwritten in the simulator to return current simulation time, which is similar to the previous simulator implementation. In addition, simulated time is allowed to tick along with the real clock and the simulator would increment the simulated time by one second at the end of the event loop if no important event happened (submission of a new job or resources deallocation).

The significant increase in backfill scheduler speed can affect the job placement. In real Slurm a single backfill cycle can take several minutes while in the simulation it

takes seconds, that can lead to jobs starting much earlier in the simulation. To compensate for that we scale the execution time taken by the backfill scheduler by the speed-up factor calculated as the average ratio of execution time of real and simulated Slurm over the number of jobs it is attempting to schedule.

The overall simulation process is as follows: (1) compile Slurm in simulation mode, (2) prepare the Slurm configuration files, (3) initiate and populate the Slurm accounting system, (4) prepare the job trace file containing the description of jobs submitted to the simulator, (5) prepare the simulation configuration files, (6) run the simulation (execute slurmdbd and slurmctld) and (7) analyze the results. Our Slurm Simulator code is available at https://github.com/nsimakov/slurm_simulator. A number of utilities were developed to help and to improve usability during these steps. These utilities as well as a documentation are available at https://github.com/nsimakov/slurm_sim_tools.

## 4 Studied HPC Cluster Systems

### 4.1 Micro Cluster

For the validation of the Slurm simulator, a small theoretical cluster, named Micro-cluster, was modelled using regular Slurm. Slurm has a front-end mode where a single slurmd communication daemon is used for all nodes. This mode is often used by Slurm developers for validation and developmental purposes. In this mode, most of the Slurm infrastructure is in-place and it functions in the same way as the real system; the users batch scripts are submitted through sbatch command and squeue and other utilities function in the same way. Because in this mode there is only one real compute node the users batch jobs simply consist of a sleep command with a requested duration as an argument. The utilization of this model allows us to execute real Slurm under the same workload multiple times in order to evaluate the variability intrinsic to real Slurm. The utilization of a single historical workload from the actual cluster does not offer such an option.

The characteristics of the Micro-cluster's nodes are shown in Table 1. This configuration includes two different types of compute nodes, large memory nodes and GPU nodes. This selection allows us to validate job placement based on the resource requests within the batch job. The 500 trace jobs were generated requesting either non-specific nodes or specific nodes. Jobs were distributed between five users grouped in two accounts (three users in one account and two in other account). Specific requests can be either for a CPU type or for a large size memory or for a GPU as a generic resource (GRES for GPU). The job sizes were randomly selected and varying from serial to 1–8 node parallel jobs. The wall time request was randomly selected between 5 and 30 min. Actual execution time was randomly selected between 0 and the requested wall time (10% of the jobs were set to use the entire requested time and 10% to use none, to model failed jobs). The distribution of core hours assigned to users and accounts in the generated job trace is shown in Fig. 1. The network topology was each

of the two types of compute node were connected to their own switch, the large-memory node and the GPU node were connected to the same switch and all of these switches connected to the top switch.

**Table 1.** Specification of modeled Micro cluster.

| Node type | Number of nodes | Cores per node | CPU type | RAM, GB |
|---|---|---|---|---|
| Compute | 4 | 12 | CPU-N | 48 |
| Compute | 4 | 12 | CPU-M | 48 |
| High memory | 1 | 12 | CPU-G | 512 |
| GPU compute | 1 | 12 | CPU-G | 48 |

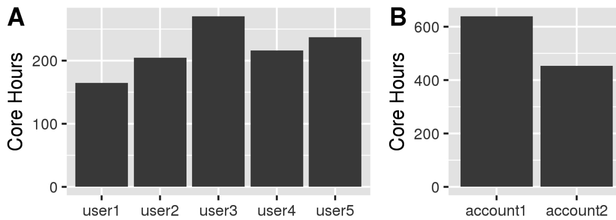



**Fig. 1.** (A) Core hours consumed by users on Micro cluster. (B) Core hours charged to account (users 1,2,3 belong to account 1 and users 4 and 5 belong to account 2).

### 4.2 Rush – HPC Production Cluster

Rush cluster is a cluster used in our center for academic users. Its' specification is summarized in Table 2. The workload was simulated using historical jobs running on the actual cluster between October 4, 2016 and October 28, 2016 comprising 23.8 days. The start and end days corresponds to two system scheduled services days when all the nodes were drained down. All historical jobs were included in the simulation including cancelled jobs, because although not running they affect the job placement on the cluster. The requested jobs' resources were extracted from the Slurm accounting and actual users' batch jobs. Slurm accounting does not store complete information about resources requested by jobs, particularly it doesn't store requested CPU types or job dependencies. This information was extracted from the users' scripts, however some portion of this information can be specified as arguments to the sbatch utility and therefore was not captured. The historical job set consists of 65,000 jobs from 161 users utilizing 83 accounts consuming a total of 3,300,000 core hours. The distribution of jobs over core counts and wall time is shown in Fig. 2, these properties were obtained from the OpenXDMoD tool installed in our center [12]. The complete Slurm configuration parameters can be found in the supplementary material.

**Table 2.** Specification of real rush cluster.

| Node type | Number of nodes | Cores per node | CPU type | RAM |
|---|---|---|---|---|
| Compute | 32 | 16 | Intel E5-2660 | 128 GB |
| Compute | 372 | 12 | Intel E5645 | 48 GB |
| Compute | 128 | 8 | Intel L5630 | 24 GB |
| Compute | 128 | 8 | Intel L5520 | 24 GB |
| High memory | 8 | 32 | Intel E7-4830 | 256 GB |
| High memory | 8 | 32 | AMD 6132HE | 256 GB |
| High memory | 2 | 32 | Intel E7-4830 | 512 GB |
| GPU compute | 26 | 12 | Intel X5650 | 48 GB |

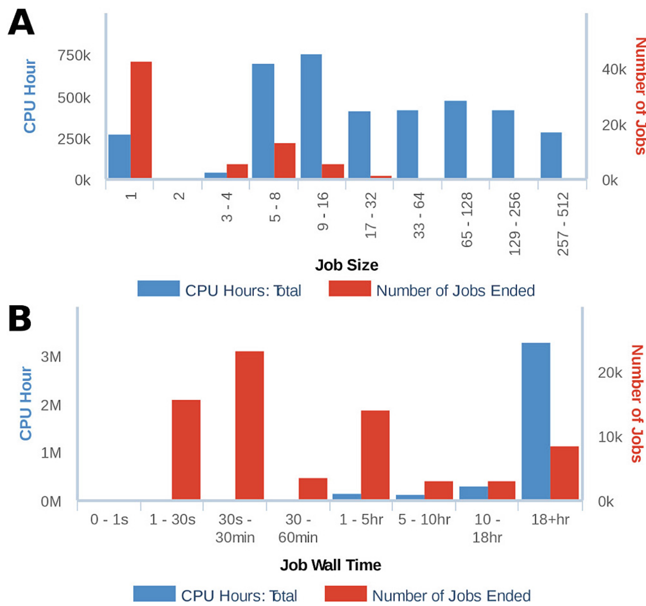



**Fig. 2.** Rush Cluster workload characterization for the period between October 4, 2016 and October 28, 2016. (A) Distribution of core hours and number of jobs over the number of allocated cores. (B) Distribution of core hours and number of jobs over the jobs' duration.

## 4.3 Stampede2 – Supercomputer

Stampede2 is a supercomputer at The University of Texas at Austin's Texas Advanced Computing Center (TACC). It consists of 4,200 Intel Xeon Phi Knights Landing (KNL) nodes and 1,736 Intel Xeon Skylake-X (SKX) nodes.

At the time of this article writing Stampede2 was still in the deployment stage and thus there was little historical workload available. Furthermore, the available workload is mainly from the initial deployment stage and it would not be representative of the workload during production stage. For the simulation, the workload was generated

from the historical workload of Stampede1 supercomputer which is 6,400 node supercomputer at TACC.

The workload was generated as follows. First, a job bank was created from stampede1 historical jobs running after 2015-05-16 and submitted before 2015-08-08 (12 weeks period). All single node jobs were converted to cores-requested jobs using average CPU utilization with rounding to closest biggest core count of 1,2,4,6,8 and 12. Jobs with CPU utilization higher than 12 was considered as requested a whole node. The CPU utilization data was obtained from XDMoD [12] using TACC-Stats data [5]. Next jobs were randomly selected from job bank (without replacement). Number of selected jobs was proportional to node counts of Stampede 1 and 2. Next portion of jobs was set to be executed on KNL nodes (number of jobs proportional to portion of KNL nodes).

Two configuration for Slurm controllers were tested: one controller for all nodes and separate controllers for KNL and SKX nodes. Three options for node sharing on SKX nodes was simulated: no sharing, sharing by cores (each shared job has dedicated cores) and sharing by sockets (each shared job has all cores from same physical CPU).

# 5 Results and Discussion

## 5.1 Validation

The Slurm simulator is essentially the normal Slurm resource manager with several modifications and simplifications. These include: serialization, faster performance of individual components, neglect of jobs epilog time and node failures. Therefore, for the proper use of the simulator it is important to know how these simplifications affected the simulator performance.

**Validation on Micro-cluster.** The first validation of the Slurm simulator was done against the small, 10 nodes, model cluster. The multiple simulated runs were compared to multiple runs of the regular Slurm under the same workload. The workload consisted of 500 jobs and takes 12.9 h to complete. In all cases, Slurm preformed a proper job placement based on user requests: jobs requesting large amount of memory, GPUs or specific CPU type executed on nodes with suitable characteristics.

The job start time was used to characterize the differences between the Slurm runs. Other characteristics like waiting times and system utilization are essentially derivatives of the individual job start times. It is interesting to compare multiple runs of Slurm in both simulated and normal mode. The start time difference between a single simulated and a real Slurm run is shown in Fig. 3(A). For that particular runs, the mean start time difference is 0.8 min with a standard deviation of 57.0 min and number of outliers exceeding 4 h. For a 12.9 h workload, such variability is comparable to the differences between two real Slurm runs with similar initial conditions (Fig. 3(B)). The mean difference for that real Slurm runs is 1.4 min with a standard deviation of 50.3 min. In Slurm, many routings are executed regularly with a sleep cycle between the executions. Among these routings are main and backfill schedulers and priority

decay calculations. Because of the system jitter and varying workload, the execution time of these functions vary leading to differences in relative function start-up times. This can create an opportunity hole for some jobs, which can be placed on a resource earlier and eventually lead to different job placement for other jobs leading to relatively large differences in scheduling from run to run.

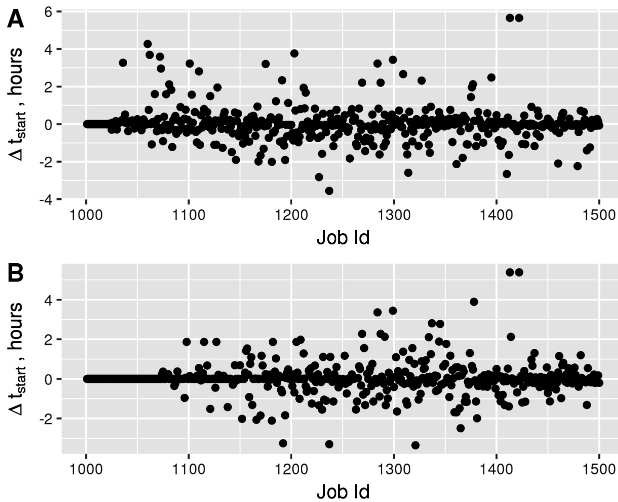



**Fig. 3.** Job start time difference between (A) simulated and real Slurm runs and (B) two real Slurm runs.

The multiple Slurm runs were obtained by varying the time between the Slurm controller boot time and the first job submission time, the submission time of other jobs relative to the first job remains the same. This serves as initial seed and varies the initial calls to the main and the backfill schedulers relative to the first job in the workload. To study the variability, seven real Slurm runs and 120 simulated runs were performed. The job start times were compared to the average job start times over all real Slurm time.

The start time differences for all real and simulated jobs is shown in Fig. 4(A) along with its distribution in Fig. 4(B). The difference between mean values for real and simulated Slurm distribution is 3.2 min (on average the simulated Slurm jobs start later) and the standard deviation is 36.9 and 42.5 min for real and simulated Slurm runs respectively. There are a number of jobs in both simulated and real Slurm exceed 3 h. For each run the mean and standard deviation of all job start time differences were calculated. The Student t-test and the Kolmogorov-Smirnov test were used to compare the simulated and the real runs. The tests showed that although the means cannot be distinguished the standard deviations are different (p-values < 0.01). Some of this 15% difference in standard deviation between real and simulated runs can be attributed to a smaller number of independent runs there (7). There is an interesting time dependence exhibited during the runs. In the beginning of the run the deviation is small and starts to grow as the accumulation in job placement varies. However instead of growing

indefinitely it reaches certain level and stays there. This is probably caused by a steady stream of new jobs and the retirement of old jobs resulting in historical job placement having a decaying memory when it affects the allocation of new jobs. As for the starting times the system utilization (Fig. 5) and the job priorities (Fig. 6) change over time in a similar fashion between the normal and the simulated runs. For the Slurm simulator it took 17 s to complete this 12.9 h workload, that is the simulation speed for the Micro-cluster was 112 simulated days per hour.

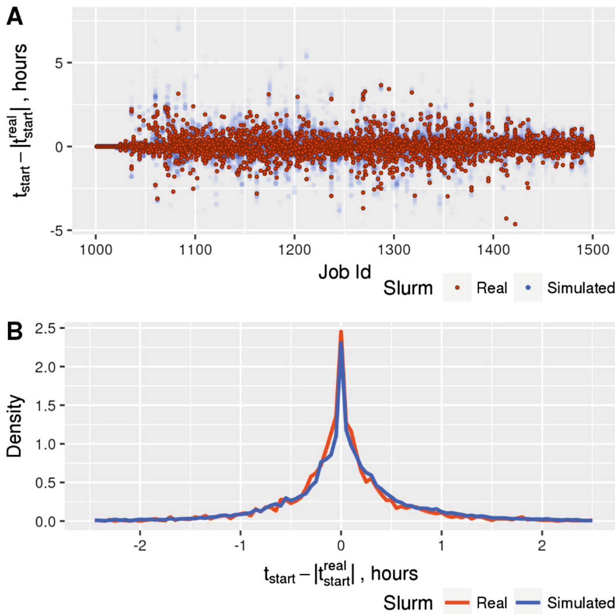



**Fig. 4.** Comparison of job starting times between multiple real and simulated Slurm Runs. (A) Difference between job start times and mean job start times over all real Slurm runs for real and simulated Slurm runs. (B) Distribution of start time differences from A for real and simulated Slurm runs.

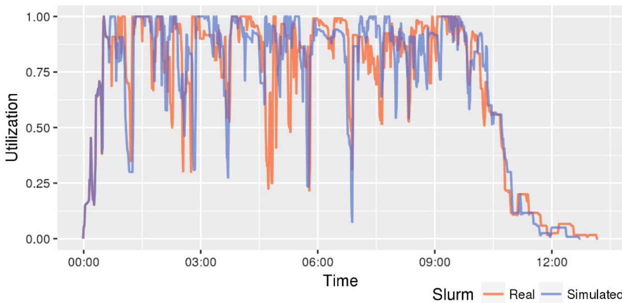



**Fig. 5.** Comparison of Micro-cluster utilization between single real and simulated Slurm runs, aggregation was done over 1 min period.

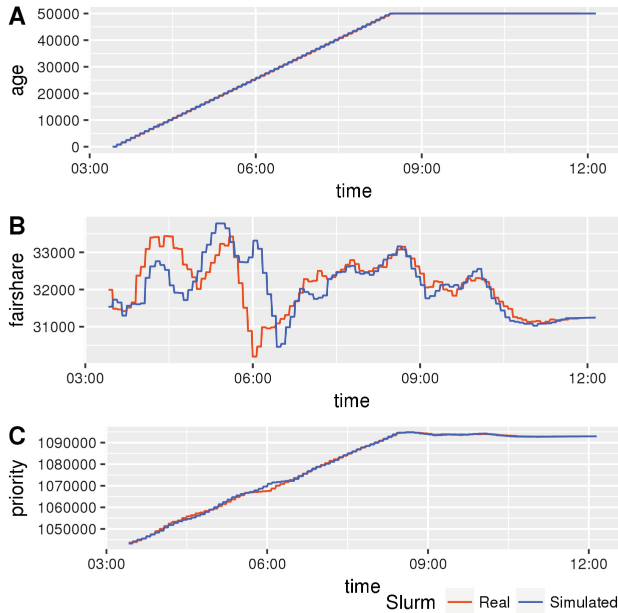



**Fig. 6.** Comparison of job priority (C) and its' components, age factor (A) and fair-share factor (B), change over time between single real and simulated Slurm runs for one of the long pending jobs. The fair-share priority factor has a step-like form because these values are recalculated every 5 min.

Often instead of absolute prediction there is an interest in predicting a change upon modification of some parameter. In order to study this, fair-share priority factor weight was increased by 20% and the simulator and the normal Slurm was rerun using the new value of this parameter. The fair-share priority factor alters the job's priority based upon the user for the resource allocation order in the main scheduler and in the backfill scheduler. The fair-share factor takes into consideration the individual user's previous resource usage as well as the resource usage of all other users from same account. The fair-share factor allows users with lower previous usage to receive an allocation to resource faster than users with higher usage. The resulting change in waiting times is shown in Fig. 7. The effect of 20% increase in fair-share weight is small and does not exceed 5% of jobs wait time. This is due to similar resource utilization by all users (Fig. 1(A)). The first three users belong to account one and the last two users belong to account two. The resource utilization by account one users is 30% higher than that by users from account two users (Fig. 1(B)). The simulation predicts that on average users from account one would have longer waiting times while users from account two shorter. However, the variation in predicted values is high and a significant portion of the distributions lays on both sides of zero. The wait time differences from two sets of real Slurm runs stays within predicted values (Fig. 7). Because of relatively small difference in resource utilization by all users and associated accounts the resulting difference is small as well. To produce a more pronounce difference a second

experiment was performed where 70% of user 4 was reassigned to user 3 (Fig. 7(B) and (C)). The simulation predicts that user 4 will have 18% smaller waiting time and more than 75% of distribution for user 4 from account two lays below zero. Indeed, in real Slurm run users 4 and 5 have significantly lower wait time (Fig. 7(A)), however unlike the mean values from simulation the user 1 and 2 are higher. Nevertheless, the values from real Slurm are within simulator predicted values (Fig. 8).

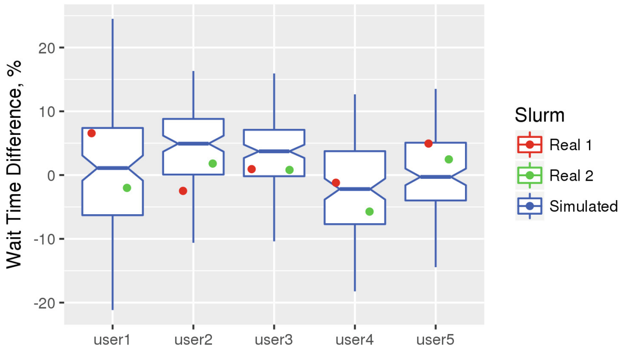



**Fig. 7.** Effect of increasing fair-share priority factor weight by 20% on waiting time for each user. Change in waiting time for simulated runs shown as boxplot where the lower and higher sides corresponds to first and third quartiles, the horizontal bar in the box is median and the whiskers extends to the furthest value not exceeded 1.5 of inter-quartile range. The wait time differences from two sets of real Slurm runs are shown as red and green points. (Color figure online)

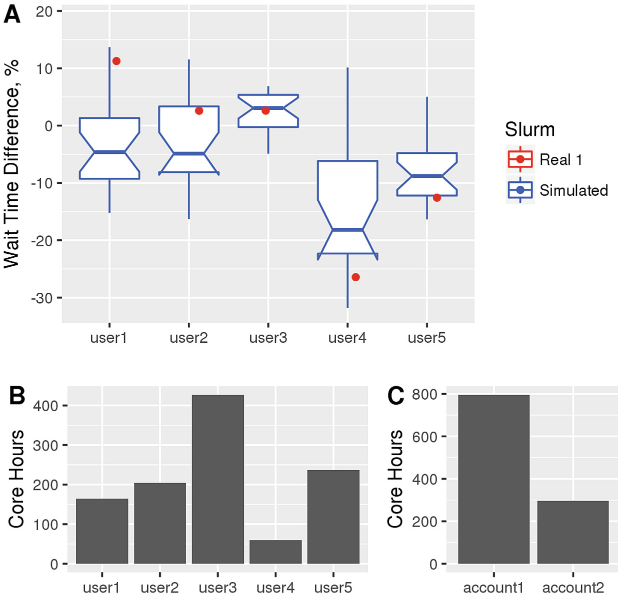



**Fig. 8.** (A) Same as Fig. 7 but with modified workload shown in (B) and (C).

The model Micro cluster allowed us to study the simulator in comparison to multiple real Slurm runs. The simulator predicted mean start times statistically undistinguishable from actual Slurm start times with a slightly larger standard deviation. The simulator properly allocates resources and updates jobs priority values. The simulator predicted ranges for waiting time change upon modification of fair-share priority factors similar to those of the real Slurm runs.

### 5.2 Validation on the Rush Cluster

In order to evaluate how the simulator compares with real Slurm executed on a real system we performed a simulation of using an actual historical workload from our cluster. As opposed to the modeled Micro-cluster here we only have one historical result. In the simulation, we do not take into consideration the cluster usage prior to the beginning of the simulation or node failures. Both have a significant effect on the job scheduling as the first one affects the job priorities calculation and the second makes the effective simulated cluster bigger. Node failures is a significant component of aging clusters and can be added to the simulator in the future.

In normal Slurm running on medium to large clusters the backfill scheduler takes a significant amount of time requiring more than several minutes per single loop. Due to simplification in the simulator the backfill scheduler runs more than 10 times faster. An artificially faster scheduler can affect the job placement and produce significant deviation from the real Slurm performance. Although the backfill scheduler execution time for real and simulated Slurm have similar patterns (Fig. 9(A)) the actual Slurm has much more noise than the simulated one (Fig. 9(B)). Such variability is because real Slurm needs to spin-off a number of threads to serve various users requests, jobs allocation and deallocation. These threads can slow the backfill scheduler due to thread locking and the decreased amount of CPU time available to the scheduler thread. Slurm in simulated mode does not have such interruptions. Ideally a good simulator would incorporate such effects. In this article, we simply scale the backfill scheduler loop by the speed-up factor. In the real Slurm the dependency of the backfill scheduler run time on the number of jobs attempted to schedule is non-linear. However, we were not able to produce a better model than simple scaling from the available data. Collecting a wider range of Slurm performance metrics might help to improve the backfill scheduler model.

The difference of jobs start time between historical and simulated Slurm runs is shown in Fig. 10. The average difference in jobs start time between historical and simulated Slurm run was −2.4 h (on average simulated jobs start earlier) with a standard deviation of 12.0 h. There is a large number of outliers with time difference more than 4 days. The outliers' deviation from zero decreases over time, this is due to neglect of the initial resource usage. Because in fair-share the previous utilization contribution decays over time, the influence of initial historical usage diminished resulting in smaller deviation further from the starting point. It is interesting to compare this standard deviation with one between two simulated runs. The difference of job start

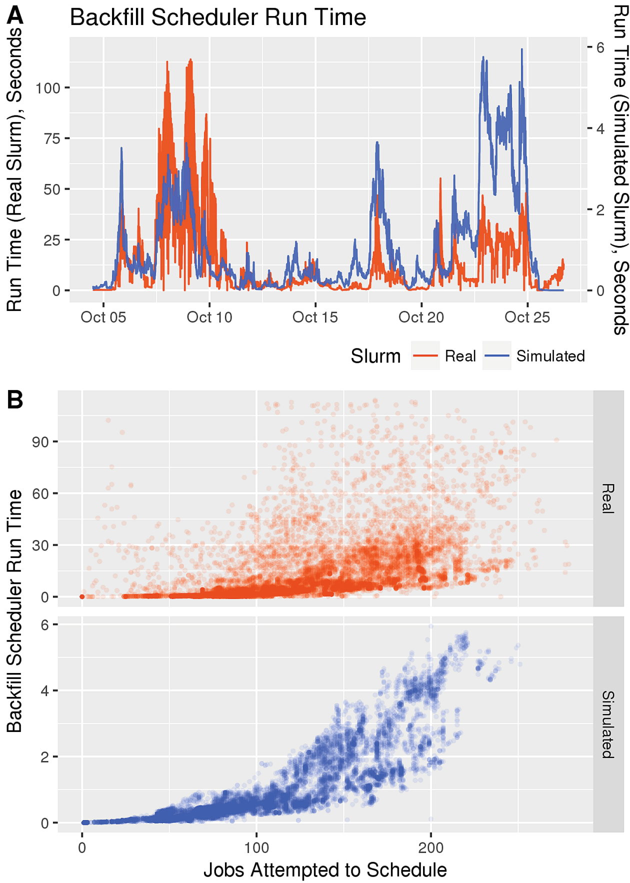



**Fig. 9.** (A) Backfill scheduler execution time historical variation in real Slurm along used workload. (B) Dependency of backfill scheduler execution time on the number of jobs attempted to schedule for single historical and simulated runs

times between two simulated runs is shown in Fig. 11. It has a mean of 1.3 min and a standard deviation of 2.5 h. Figure 12 shows the whole resource utilization. Both historical and simulated results show a similar pattern. Interestingly in the beginning the simulated run has a higher utilization. Probably the omission of initial historical usage by users allows the simulator to allocate resources more efficiently. Throughout the middle part of the timeline there are a number places where the historical run has a higher utilization than the simulated runs. Most likely some of the job resources requests were specifically tailored by users to fit the gaps in the cluster at that time and in the simulator these gaps were different preventing the job placement.

Given the differences in initial conditions and lack of node failures modeling the simulator shows a reasonable approximation of historical workflows. Combined with results from the Micro-cluster the simulator can be used for studies of the effects of various Slurm parameters on the system performance.

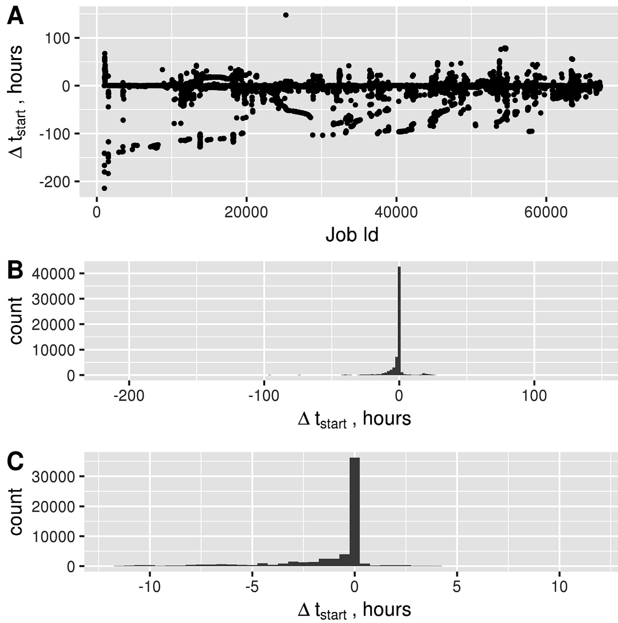



**Fig. 10.** (A) Difference in jobs start time between real and simulated Slurm runs. (B) Distribution of that start time difference. (C) Same as B, zoomed to −12 to 12 h region.

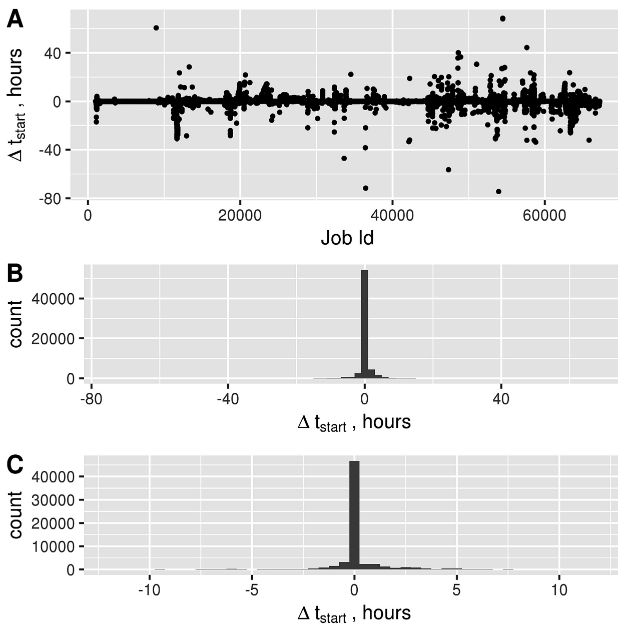



**Fig. 11.** (A) Difference in jobs start time between two simulated Slurm runs. (B) Distribution of that start time difference. (C) Same as B, zoomed to −12 to 12 h region.

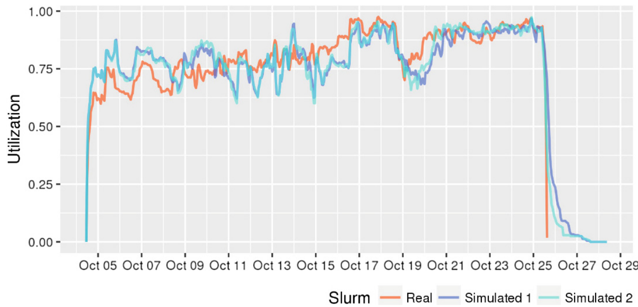



**Fig. 12.** Comparison of resource utilization between historical Slurm run and two simulated runs.

## 5.3 Study of Various Parameters

**Node Sharing.** Many smaller HPC centers, like ours, have a large portion of serial jobs and jobs which do not fully utilize an entire node (Fig. 1(A)). Allowing multiples of such jobs to be executed on the same node increases the overall system throughput. This is often referred to as node sharing mode as opposed to node exclusive mode where the whole node is allocated to a single job. It was shown that many applications running on a fraction of a node's cores have less than 5% decrease in their performance when running in none sharing mode [13]. In certain cases of large parallel jobs, the jobs can actually complement each over in sub-system usage leading to faster execution time [2]. In order to quantitatively determine the increase in the overall system gained by node sharing the simulator was run in node sharing and exclusive modes.

The exclusive mode takes 10.8 more days (45% more time) to complete the same workload (Fig. 13). The average increase in waiting time is 5.1 days with a standard deviation of 6.6 days. The 45% increase in time to complete the same load can be translated into the need to have a 45% larger cluster to serve the same workload. This is a major savings even after considering a potential 3% slow down by some jobs. Long wait times are a significant difficulty and users could potentially adjust their computational work load to decrease the wait time potentially endangering the quality of their work. In other words, if we would switch back to exclusive mode, the users would adjust by decreasing their usage and in this way the benefit of shared mode can be rephrased as it allows users to do more than 40% more computational job. Therefore, there is a good reason to have shared mode enabled and the users preferred to use exclusive mode still can ask for the whole node.

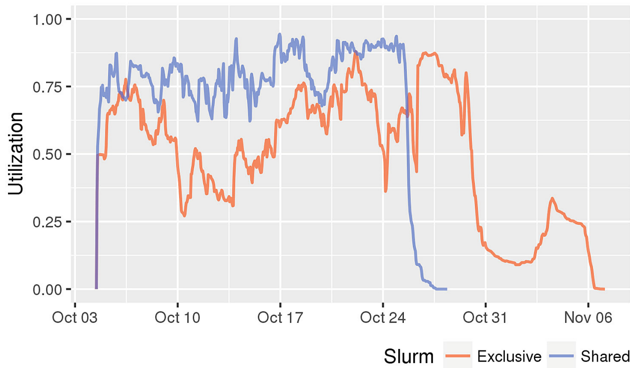



**Fig. 13.** Comparison of resource utilization for exclusive and shared node modes.

**Maximal Number of User's Jobs Considered by Backfill Scheduler for Scheduling.** The backfill scheduler allows resource allocation to jobs which do not affect the start time of higher priority jobs. In many centers a large portion of the jobs are scheduled by the backfill scheduler. This scheduler scans through pending jobs and attempts to allocate resources for them without affecting the highest priority jobs. Such a single scan can take more than several minutes and due to multiple operations requiring controller locking can significantly affect the controller responsiveness. Extremely long run times can also lead to a decreased number of jobs being scheduled. Therefore, there should be a good balance between execution time of single scan of the backfill scheduler and the quality of its scheduling. There are a number of parameters which affect the scheduler performance. Here we will consider the bf_max_job_user parameter which defines the maximal number of user's jobs considered by the backfill scheduler for scheduling. In the reference simulation bf_max_job_user was set to 20 user's job, we will show how the performance of the cluster would be affected by decreasing this parameter to 10 user's jobs.

The simulation showed only small increase in time needed to complete the workload, namely 40 min or 0.1% from the referenced time. The mean wait time is 8 min slower and the standard deviation of the wait time differences is 3 h. Surprisingly, there is no strong dependency of the job wait time on the total number of jobs submitted by the user during the simulated period (Fig. 14). With this small decrease in resource utilization and increase user's wait time, there is a 25% decrease in the number of jobs to consider for scheduling which leads to a 30% decrease in backfill scheduler run time. Therefore, a reasonable decrease of the maximal number of user's jobs considered by a backfill scheduler for scheduling have small average effect on job placement and offers significant improvement in backfill scheduler run time. This can be a good choice for systems where backfill scheduler run time is an issue.

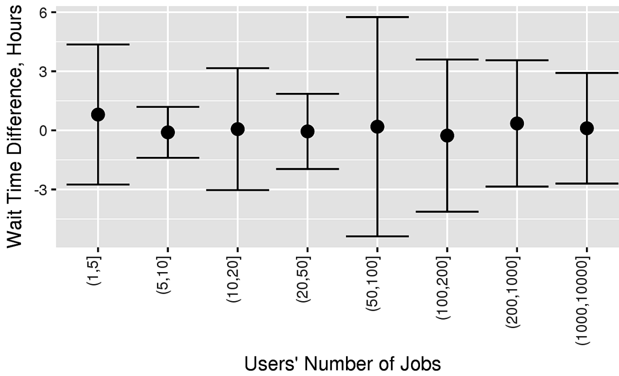



**Fig. 14.** Wait time difference between bf_max_job_user parameter set 10 and 20 for users grouped by their total number of jobs. bf_max_job_user parameter define maximal number of user's job considered by backfill scheduler. The mean values shown as points and error bar correspond to standard deviation.

## 5.4 Stampede2: Node Sharing and Multiple Slurm Controllers

Stampede2 is a new supercomputer at The University of Texas at Austin's Texas Advanced Computing Center (TACC). We chose to model it to show the capability of the simulator to handle large systems with a realistic workload and to illustrate the simulator potential for optimization of the Slurm scheduler for such systems.

Stampede2 is composed of two distinct parts: the first consists of Intel Xeon Phi Knights Landing (KNL) nodes (4,200 nodes) and the second consists of Intel Xeon Skylake-X (SKX) nodes (1,736 nodes). Because of the architectural differences between these nodes, users most likely would specifically request a particular node type for their jobs. This allows us to consider using separate controllers for each part to reduce the load on the controller and thus potentially improve performance. The drawback of separate controllers is the extra labor associated with maintenance of an additional Slurm controller. Therefore, it would be useful to estimate the benefits of separate controllers first prior to the actual implementation.

Because of the large number of cores per node on SKX nodes (48 cores per node) the benefits of node sharing can be substantial. However, Stampede2 is a large system and enabling node sharing will increase the number of consumable resources (cores and memory are needed to be tracked now instead of only nodes) and would lead to a significantly higher computational load on the backfill scheduler. Such an increase could potentially render the entire system inefficient or inoperable. For this reason, the Slurm documentation does not recommend node sharing for large systems. Therefore, it is of interest to compare the simulated performance of Slurm without node-sharing with different node-sharing schemes enabled to estimate the feasibility of node sharing on the SLX nodes. In this article two configurations for node sharing were modeled: sharing by sockets, where all the cores from the same physical CPU are assignable to a single job and sharing by cores, where each core can be allocated individually.

**Table 3.** Simulated waiting times on Stampede2. Mean wait hours weighted by node hours increases the contribution of large jobs to the mean.

| Controller | Node sharing on SKX nodes | Wait hours, mean | Wait hours, mean weighted by node hours |
|---|---|---|---|
| *Jobs on SKX Nodes* | | | |
| Single | No sharing | 10.9 (0%) | 17.0 (0%) |
| | Sharing by sockets | 8.2 (−25%) | 15.5 (−9%) |
| | Sharing by cores | 8.2 (−24%) | 15.5 (−9%) |
| Separate | No sharing | 7.1 (−35%) | 15.0 (−12%) |
| | Sharing by sockets | 5.3 (−51%) | 13.8 (−19%) |
| | Sharing by cores | 5.5 (−49%) | 13.9 (−18%) |
| *Jobs on KNL Nodes* | | | |
| Single | No sharing | 8.6 (0%) | 9.2 (0%) |
| | Sharing by sockets | 7.2 (−16%) | 9.2 (−1%) |
| | Sharing by cores | 7.3 (−15%) | 9.1 (−1%) |
| Separate | No sharing | 8.2 (−4%) | 9.4 (2%) |

The mean wait time for different configurations is summarized in Table 3. For SLX jobs, dedicating separate controller resulted in a 35% improvement in waiting time. In this case the smaller number of jobs per controller and smaller number of resources to track (3.4 times less nodes) for each controller lead to much better performance by the backfill scheduler. Indeed, the backfill scheduler with only a single controller reached the run time limit in 70% of all runs. A separate dedicated controller reached the limit on only 32% of the runs. Interestingly, that there is almost no benefit for KNL jobs, this is probably due to a much smaller decrease in nodes for the KNL controller than for SKX node controller (1.4 times vs 3.4 times).

Another 22%–25% improvement in wait time can be achieved by allowing node sharing. Surprisingly sharing by sockets shows a marginally shorter waiting time than node sharing by cores. This may be due to the smaller number of tractable resources or it may be an artifact of the workload composition. The percentage of backfill scheduler runs which hit the run time limits are very similar in both cases (30% for sharing by sockets and 32% for sharing by cores).

These simulations show that it is possible to enable node sharing on SLX nodes of Stampede 2 to improve wait time. The best performance is achieved with node sharing and with dedicated controllers for the SLX and KNL nodes. The difference between allocations by cores and sockets is very small and it is problematic whether it is more beneficial to do node sharing by cores or by sockets. For this system, we only ran a single simulation for each configuration due to time constraints. Similar to the micro-cluster, it is expected that there should be a variation in wait time and ideally multiple simulations must be done to generate reasonable statistics. Therefore, these findings should be regarded as preliminary.

### 5.5 Simulation Speed

The simulator speed heavily depends on the cluster size, workload and Slurm configuration. For a small 120 core cluster the simulation speed was 112 simulated days per hour while for a medium 8000 core cluster it was in the range of 0.8 to 17.3 simulated days per hour depending on the Slurm configuration. In the reference configuration, it was 5.4 simulated days per hour. In exclusive node job allocation mode it was 0.8 simulated days per hour. With a smaller maximal number of user's job considered by the backfill scheduler, it was 17.3 simulated days per hour. For the tested large system, it was around 1 simulated day per hour. In most cases the simulation speed correlates with the backfill scheduler run time and can serve as an indicator of whether the backfill scheduler run time needs to be optimized.

## 6 Conclusions

A new Slurm simulator was developed capable of simulation of large clusters with a simulation speed of multiple days per hour. Its validity was established by a comparison with actual Slurm runs which showed similar mean values for job start times between the simulated and actual data with a slightly larger standard deviation for the simulation results. We have exercised this simulator in studying a number of Slurm parameters that affect system utilization and throughput such as fair share policy, maximum number of user jobs considered for backfill, and node sharing policy. As expected fair share policy alters job priorities and start times but in a non-trivial fashion. Decreasing the maximal number of user's job considered by the backfill scheduler from 20 to 10 was found to have a minimal effect on average scheduling and serves to decrease the backfill scheduler run time by 30%. The simulation study of node sharing on our cluster showed a 45% increase in the time needed to complete the workload in exclusive mode compared to shared mode. An initial analysis of Stampede2 supercomputer scheduling shows that it can benefit from separate Slurm controllers and node sharing.

**Acknowledgments.** This work was supported by the National Science Foundation under awards OCI 1025159, 1203560, and is currently supported by award ACI 1445806 for the XD metrics service for high performance computing systems.

## References

1. Balle, S.M., Palermo, D.J.: Enhancing an open source resource manager with multi-core/multi-threaded support. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2007. LNCS, vol. 4942, pp. 37–50. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78699-3_3
2. Breslow, A.D., Porter, L., Tiwari, A., Laurenzano, M., Carrington, L., Tullsen, D.M., Snavely, A.E.: The case for colocation of high performance computing workloads. Concurrency Comput. Pract. Experience **28**(2), 232–251 (2016)

3. Caniou, Y., Gay, J.-S.: Simbatch: an API for simulating and predicting the performance of parallel resources managed by batch systems. In: César, E., Alexander, M., Streit, A., Träff, J.L., Cérin, C., Knüpfer, A., Kranzlmüller, D., Jha, S. (eds.) Euro-Par 2008. LNCS, vol. 5415, pp. 223–234. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00955-6_27
4. Casanova, H., Giersch, A., Legrand, A., Quinson, M., Suter, F.: Versatile, scalable, and accurate simulation of distributed applications and platforms. J. Parallel Distrib. Comput. **74** (10), 2899–2917 (2014)
5. Evans, T., Barth, W.L., Browne, J.C., DeLeon, R.L., Furlani, T.R., Gallo, S.M., Jones, M.D., Patra, A.K.: Comprehensive resource use monitoring for HPC systems with TACC stats. In: 2014 First International Workshop on HPC User Support Tools, pp. 13–21, November 2014
6. Jackson, D.B., Jackson, H.L., Snell, Q.O.: Simulation based HPC workload analysis. In: Proceedings 15th International Parallel and Distributed Processing Symposium, IPDPS 2001, 8 p. (2001)
7. Klusácek, D., Rudová, H.: Alea 2: job scheduling simulator. In: Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques, p. 61 (2010)
8. Legrand, A., Marchal, L., Casanova, H.: Scheduling distributed applications: the SimGrid simulation framework. In: Proceedings of the 3rd International Symposium on Cluster Computing and the Grid, Washington, DC, USA, pp. 138–145 (2003)
9. Lucero, A.: Slurm Simulator. In: Slurm User Group Meeting (2011)
10. Maui Scheduler. http://www.adaptivecomputing.com/products/open-source/maui/. Accessed 03 Apr 2017
11. Moab HPC Suite. http://www.adaptivecomputing.com/products/hpc-products/moab-hpc-basic-edition/. Accessed 03 Apr 2017
12. Palmer, J.T., et al.: Open XDMoD: a tool for the comprehensive management of high-performance computing resources. Comput. Sci. Eng. **17**(4), 52–62 (2015)
13. Simakov, N.A., Sperhac, J., Yearke, T., Rathsam, R., Palmer, J.T., DeLeon, R.L., White, J.P., Furlani, T.R., Innus, M., Gallo, S.M., Jones, M.D., Patra, A., Plessinger, B.D.: A quantitative analysis of node sharing on HPC clusters using XDMoD application kernels. In: Proceedings of the XSEDE16 on Diversity, Big Data, and Science at Scale - XSEDE16, New York, NY, USA, pp. 1–8 (2016)
14. Slurm Workload Manager. https://slurm.schedmd.com/. Accessed 03 Apr 2017
15. Takefusa, A., Matsuoka, S., Aida, K., Nakada, H., Nagashima, U.: In: Proceedings of the 8th IEEE International Symposium on High-Performance Distributed Computing, August 3–6, 1999. IEEE Computer Society (1999)
16. Trofinoff, S., Benini, M.: Using and Modifying the BSC Slurm Workload Simulator. In: Slurm User Group Meeting (2015)
17. Yoo, A.B., Jette, M.A., Grondona, M.: SLURM: simple Linux utility for resource management. In: Feitelson, D., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2003. LNCS, vol. 2862, pp. 44–60. Springer, Heidelberg (2003). https://doi.org/10.1007/10968987_3

# Short Papers

# Path-Synchronous Performance Monitoring in HPC Interconnection Networks with Source-Code Attribution

Adarsh Yoga[1,2](✉) and Milind Chabbi[3]

[1] Hewlett Packard Labs, Palo Alto, CA, USA
[2] Rutgers University, Piscataway, NJ, USA
adarsh.yoga@cs.rutgers.edu
[3] Scalable Machines Research, Cupertino, CA, USA
milind@scalablemachines.co

**Abstract.** Performance anomalies involving interconnection networks have largely remained a "black box" for developers relying on traditional CPU profilers. Network-side profilers collect aggregate statistics and lack source-code attribution. We have incorporated an effective protocol extension in the Gen-Z communication protocol for tagging network packets in an interconnection network; additionally, we have backed the protocol extension with hardware and software enhancements that allow tracking the flow of a network transaction through every hop in the interconnection network and associate it back to the application source code. The result is a first-of-its-kind hardware-assisted telemetry of disparate, autonomous interconnection networking components with application source code association that offers better developer insights. Our scheme works on a sampling basis to ensure low runtime overhead and generates modest volumes of data. Simulation of our methods in the open-source Structural Simulation Toolkit (SST/Macro) shows its effectiveness—deep insights into the underlying network details to the developer at minimal overheads.

## 1 Introduction

Interconnection networks used in today's supercomputers play a vital role in the overall performance, efficiency, and scalability of scientific simulation and modeling. HPC applications achieve a paltry 5–15% of a machine's peak performance [1–3] on modern microprocessor-based supercomputers. A significant fraction of the loss comes from inter-node data movement.

When applications fail to make effective use of the compute resources at scale, application developers resort to profilers to understand bottlenecks. There is sufficient state-of-the-art and commercial tools for CPU profiling [4–10] that capture metrics such as CPU cycles, cache misses, branch mis-predictions, etc. and associate the measurements back to the application source code or application data objects [11,12].

M. Chabbi—Work done while at Hewlett Packard Labs.

© Springer International Publishing AG 2018
S. Jarvis et al. (Eds.): PMBS 2017, LNCS 10724, pp. 221–235, 2018.
https://doi.org/10.1007/978-3-319-72971-8_11

Domain scientists can only reason about performance when the measurements are attributed back to the application source code. Unfortunately, network performance problems are a "black box" from an application developer's viewpoint. CPU-side profilers typically quantify the amount of delay waiting for a network communication but offer little insight into *why* an instance of network transaction was slow. Even the most sophisticated network performance analysis techniques [13–16] only reason about communication endpoints but do not capture measurements from under-the-hood workings from the autonomous interconnection hardware, which includes network interface cards (NICs), bridges, and switches.

Figure 2 in Appendix A shows the execution profile of NWChem [17]—a US Department of Energy flagship computational chemistry code—running with 1024 MPI ranks on the Dragonfly [18,19] interconnection network on the NERSC Edison [20] supercomputer. The figure shows a hotpath in the CPU profile taken using HPCToolkit [5], a state-of-the-art CPU PMU-based profiler. The figure shows a deep call stack with various layers of host-side code leading to the vendor-provided networking API `dmapp_lock_acquire` to acquire a lock on a remote node. The execution spends a significant (26%) part of execution waiting in this networking API, but the profiles cannot obtain any insights on the cause of this wait. This leaves an application developer with many unanswered questions:

1. Is there load imbalance in the code? Our conversation with the NWChem application developers eliminated this case of any load imbalance and contention for a single lock—the workload is dynamically balanced.
2. Is the network lock implementation suboptimal? Our conversation with Cray Inc. eliminated this possibility—the network lock is local spinning MCS [21] lock.
3. Is the communication network performing poorly?
4. Is there an interference from another job that affected this execution?
5. Is the observed, seemingly network problem, indeed a network bandwidth problem or delays in the local NICs to inject messages?
6. If locking is frequent, is the lock-release message getting delayed in the interconnection network? If so, can we use a separate high-priority virtual channel for such network communication that appear on the critical path?

Clearly, traditional CPU profilers cannot offer answers to these questions since they cannot measure what happens in the interconnection network hardware components. Once a network-related transaction leaves the CPU, even in a simplistic network, the following events happen. The message gets enqueued as a command to the NIC. The NIC notices the command at some later point, which introduces an arbitrary delay. Now, the NIC may initiate a DMA transfer from the local DRAM if the command is a send/put. It packetizes a put/send command into multiple MTU-sized packets and injects them one by one. The NIC may then wait for the acknowledgement of every packet (which is the case in Gen-Z [22] protocol). Different packets may take different paths in the network based on the network routing heuristics. At each router hop, a packet may be subject to different policies and arbitration delays before being forwarded to

an output port. At the destination NIC, the packets may arrive out of order (which happen in Gen-Z [22]). The destination NIC may delay injecting packet-level acknowledgments. The destination CPU may get notified some time later after the entire message is reassembled and may introduce further delays before a message-level acknowledgment is generated. Finally, an acknowledgment message may be subject to the same set of uncertainties on its return journey.

With myriad autonomous, unsynchronized components, it is virtually impossible to track how a message gets affected in its roundtrip from one host to another. Prior network performance analysis efforts have conducted an event-driven simulation of the network with characteristic workloads for designing superior networks without paying attention to delivering developer insights. Production hardware has offered simple counters in network routers to collect aggregate runtime data, which offer coarse-grained statistics for system administration to spot anomalous or overloaded hardware components; these techniques are tedious, vendor specific, and often not accessible to CPU profilers.

No prior art has addressed the challenge of tracking an individual message from its source location through every hop in every hardware component in an interconnection network and associated the observed performance metrics to the source and target host codes. This level of detailed measurement in conjunction with full CPU-side context-sensitive profiling is the basis of delivering rich, end-to-end application insights. Such detailed profiling and tracing can alone answer questions that we raised previously in the NWChem example. Evidently, tracking every message and every packet in the network with this level of detailed statistics is a recipe for performance data deluge and will bring the network to a grinding halt in merely collecting the measurement data. Statistical sampling comes to our rescue in collecting detailed data with sparse sampling.

Our strategy is to "mark" network transaction to be monitored on a sampling basis at the origin (CPU) and record statistics of such marked messages at every hop along its journey in an interconnection network. By retaining both CPU-side profiles and network profiles for a sparse set of samples, we are able to observe what happens to network transactions and elevate the measurements to application source code in a manner that sheds lights on the causes of network-related problems to the application developer. The result is that the application developer, with full understanding of the problems, may,

1. Choose to refactor the source code to better utilize the network, or
2. Provision more network resources to reduce network-related bottlenecks that are caused by her application, or
3. Conclusively infer that the problem was not caused by the application but due to an interference with another job, the solution is in better network provisioning or job scheduling, or
4. Pinpoint that the problem is not in the network provisioning but in the networking algorithms, an anomalous router, or local network interface (NIC) software or hardware.

## 2 Related Work

There is a rich literature on profiling and tracing CPU executions. Profiling provides aggregated metrics whereas tracing captures the time-varying behavior of executions. Profiling and tracing come in various flavors and granularities. It is common to instrumentation source code or binary, manually or via a tool, at function, loop, or basic-block granularities. Hardware event-based sampling is an orthogonal method where CPU PMU counter overflow triggers an interrupt that a profiler captures and attributes to application binary and in-turn to the source code [4,5,23]. None of these techniques measure data from interconnection network hardware.

MPI profilers [24–26] capture communication metrics at endpoints: they measure time spent in networking-related tasks by wrapping or intercepting each MPI library functions. Advanced methods [13] are able to replay execution traces to pinpoint the root causes of some performance bugs. However, none of these methods obtain measurements from networking hardware. As a result, although one might observe anomalous communication delays, there exists little evidence to isolate problems to a host-side NIC, a router, the destination NIC, or destination CPU.

Networking hardware design is often performed via low-level event-driven simulators [27–29]. These simulators are driven by predefined communication patterns to assess the strength of hardware designs or algorithms. A low-level simulator can simulate only a small (often milliseconds to a second) amount of real execution. High-level simulators [30–32] capture runtime communication traces on real execution and replay the communication traces to drive coarse-grained network simulators. Both high-level and low-level network simulators treat the CPU execution as a black box and focus only on the networking aspect and hence are incapable of offering insights to application an developer at the source code level.

There is rich literature in network profilers for Ethernet [33–35]. We are unaware of any tool that can a) attribute network profiles to application source code, or b) perform path-synchronous sampling to capture a specific network transaction (e.g., traversal of a specific packet) throughout its journey. Network-side monitoring schemes such as sFlow [35] and netflow [34] capture the source and destination of a packet when flowing through a component. They, however, lack the full path information of a sampled transaction and hence the hop-by-hop details of any specific packet is unavailable. sFlow can aggregate the data from many components over long periods of time and filter the data by the traffic originating from the same source going to the destination to reconstruct an "average" behavior and a "typical" path; but such schemes cannot attribute the observed behavior to the application source code because over time there can be many source code locations contributing to the same "flow". Samples from different components lack temporal correlation. This lack of temporal correlation means one can observe only aggregate behavior of traffic and not be able to pinpoint a specific anomaly to its causes. Aggregate metrics handicap the ability to pinpoint the cause of a transient anomalous behavior.

## 3 Methodology

A key requirement for attributing network behavior to application source code is to identify what happens to a transaction initiated by a line of source code (could be an assembly instruction) throughout its journey through the network. Such hop-by-hop tracking retains temporal correlation among performance metrics generated by unsynchronized components. We track hop-by-hop metrics of a small subset of packets as they are forwarded across a network. This is done on sampling basis because observing every transaction is infeasible both from space and time overhead viewpoint. In other words, one in N packet originating from a source is chosen to be tracked throughout its journey. The choice of N can be arbitrary or more intelligent. Each endpoint may choose the same or different value of N. Endpoints need not coordinate when they track a packet. Sampling ensures that any event that is statistically significant will be observed with the frequency proportional to its occurrence. We propose the following extensions:

**Protocol Extension:** every packet of the protocol carries a special Performance Monitoring (PM) tag. The PM tag may be present at a designated offset in the packet header to make it quick to inspect by the hardware. We call a packet whose PM tag is enabled as a "marked" packet. We have already incorporated a PM tag in the Gen-Z protocol [22] to enable performance tools.

**Hardware Extensions:**
1. The NIC exposes a special tag "track me" (TM) to the software. The software may assert the TM bit in a command it issues to the local NIC indicating the NIC to track the command.
2. The NIC propagates the TM bit from a CPU-issued command into a (one or more) packet(s) by setting the PM bit in the packets that it injects into the network on behalf of the command.
3. Every switch inspects the PM tag of each packet it routes. If the PM tag is enabled in an incoming packet, the switch logs a performance data record into its local buffer (typically an SRAM). The PM tag is propagated through the switch from an incoming packet to the corresponding outgoing packet.

The fact that a marked packet's information is logged at each hop allows us to achieve the path-synchronous sampling. A key piece of information logged at each hop is the unique identity of the next hop of the packet. The next hop information allows us to, in a post mortem pass, reconstruct the full path along the journey of a marked packet. In systems with request-response protocol (e.g., Gen-Z), the PM tag is retained from request to response so that its journey is tracked in both directions. To accomplish this, the endpoint hardware (e.g., NIC) may be modified to propagate the PM tag from request to response. We assume that every network packet at least contains its source identifier (SID), destination identifier (DIS), a tag (need not be unique), and the PM tag. The log in each component contains at least the following information:

1. The arrival time of the packet or command (component local time).
2. The departure time of the packet or command (component local time).

3. The identity of the next hop (out going port) of the packet/command.
4. (Optional) In addition to the first three necessary data, a component may include any additional data: for example, anomalous condition at the time of routing the designated marked packet (e.g., ran out of credit when transmitting this packet), position of the packet in a router's input queue on arrival, conflict during router arbitration, etc.

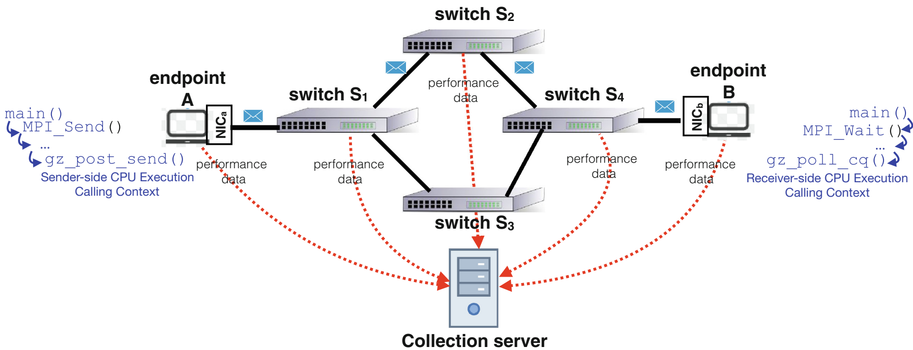



**Fig. 1.** An interconnection network with four switches, two endpoints and their respective NICs. A message sent from A to B traverses the path A→$NIC_a$→$S_1$→$S_2$→$S_4$→$NIC_b$→B. The profiler captures CPU-side contexts, marks the message to be tracked in the network and logs data to a collection server. NICs and switches that the marked packet traverses also log their data to the collection server.

Figure 1 depicts the workflow when the endpoint A wants to send a message to endpoint B and the packet follows the route A→$NIC_a$→$S_1$→$S_2$→$S_4$→$NIC_b$→B in an anecdotal network:

1. The software (profiler running on the source CPU, endpoint A) on a sampling basis chooses a transaction to be monitored. The choice can be random sampling or more intelligent, if desired.
2. The software captures its CPU calling context ($CTXT_1$) and creates a locally unique command id ($CID_1$) representing the network command.
3. The software (at time $T_1$) issues the network command to $NIC_a$ passing the unique id ($CID_1$) setting the TM flag.
4. Software logs the tuple $\langle CTXT_1, CID_1, T_1, A, B\rangle$.
5. $NIC_a$ at a later point (time $T_2$) inspects the command, generates some M network packets for the command, and by observing the TM flag, it enables the PM tag in one of (randomly chosen or otherwise) the M network packets.
6. $NIC_a$ injects the PM-marked packet at time $T_3$ to the switch $S_1$. Let the id of the marked packet be PKID. Let the last packet corresponding to $CID_1$ leave at time $T_4$. $NIC_a$ logs the local information tuple $\langle CID_1, A, B, PKID, S_1, T_2, T_3, T_4\rangle$.

7. The switch $S_1$ notices the marked packet with PKID at time $T_5$ and forwards it to switch $S_2$ at time $T_6$ and the logs the information tuple $\langle$A, B, PKID, $S_2$, $T_5$, $T_6\rangle$.
8. The switch $S_2$ notices the marked packet with PKID at time $T_7$ and forwards it to switch $S_4$ at time $T_8$ and the logs the information tuple $\langle$A, B, PKID, $S_4$, $T_7$, $T_8\rangle$.
9. The switch $S_4$ notices the marked packet with PKID at time $T_9$ and forwards it to $NIC_b$ at $T_{10}$ and the logs the information tuple $\langle$A, B, PKID, $NIC_b$, $T_9$, $T_{10}\rangle$.
10. $NIC_b$ at time $T_{11}$ assembles all packets and create an entry for the endpoint B and produces the log entry $\langle$A, B, PKID, $CID_2$, B, $T_{10}$, $T_{11}$, TM = 1$\rangle$.
11. The CPU at endpoint B at time $T_{12}$ in calling context $CTXT_2$ receives the full message and on noticing the TM flag, logs the tuple $\langle CTXT_2$, $CID_2$, $T_{12}$, A, B$\rangle$.

For brevity, we are not discussing the case of response or acknowledgment or dropped packets. In unreliable networks when a marked packet is dropped, no further logs will be available—a clear indication of a dropped packet. We do not discuss what additional information a component may log. There can be component-specific fields, which, for example, can include link-level credits.

*Collection server:* Hardware has a limited local buffer to log performance data. Hence, we use a management software running on each hardware component to periodically drain the logs collected to a centralized server. The SRAM buffer on the hardware acts as a circular buffer. All modern HPC networking components have additional management hardware with Ethernet connections of ∼1 GBPS. The management software on each component is capable of NFS mounting a remote distributed server and dump logs from local memory to a unique file on the remote server.

*Post-mortem analysis:* The collection server contains logs collected from all components through which every marked packet traverses. A post-mortem analysis of the logs in the collection server allows a software tool to reconstruct the complete path traversed by each marked packet initiated at a source and associate the data with the application source code in its calling context. In the previous example, starting from the CPU-side log of the endpoint A, we can go through the following steps to reconstruct the path:

1. Endpoint A's log entry $\langle CTXT_1$, $CID_1$, $T_1$, A, B$\rangle$ tells that at source-code context $CTXT_1$, a command $CID_1$ was issued to target B.
2. Sifting through $NIC_a$'s logs for $CID_1$ shows the following entry: $\langle CID_1$, A, B, PKID, $S_1$, $T_2$, $T_3$, $T_4\rangle$. $T_2 - T_1$ is the in-node delay. The command took a total of $T_4 - T_2$ time to get injected. The marked packet has the tag PKID and was injected at time $T_3$ and was sent to switch $S_1$.
3. Sifting through switch $S_1$'s logs for $\langle$A, B, PKID, $T_3 \pm \Delta\rangle$ shows a record $\langle$A, B, PKID, $S_2$, $T_5$, $T_6\rangle$. The packet's delay at hop $S_1$ is $T_6 - T_5$. It was forwarded to $S_2$.

4. Sifting through switch $S_2$'s logs for $\langle$A, B, PKID, $T_6 \pm \Delta\rangle$ shows a record $\langle$A, B, PKID, $S_4$, $T_7$, $T_8\rangle$. The packet's delay at hop $S_2$ is $T_8 - T_7$. It was forwarded to $S_4$.
5. Sifting through switch $S_4$'s logs for $\langle$A, B, PKID, $T_8 \pm \Delta\rangle$ shows a record $\langle$A, B, PKID, B, $T_9$, $T_{10}\rangle$.$\rangle$. The packet's delay at hop $S_4$ is $T_{10} - T_9$. It was forwarded to the destination $NIC_b$.
6. Sifting through $NIC_b$'s logs for $\langle$A, B, PKID, $T_{10} \pm \Delta\rangle$ shows the following entry: $\langle$A, B, PKID, $CID_2$, B, $T_{10}$, $T_{11}$, TM = 1$\rangle$. $T_{11} - T_{10}$ is the delay at $NIC_b$. It was delivered to the destination B.
7. Sifting through endpoint B's logs for $\langle$A, B, $CID_2$, $T_{11} \pm \Delta\rangle$ shows the following entry: $\langle CTXT_2$, $CID_2$, $T_{12}$, A, B$\rangle$. $CTXT_2$ is the receiving application calling context. The packet's journey ends here.

Full calling context with source code attribution at both endpoints along with hop-by-hop metrics for the traversal: $A(CTXT_1) \rightarrow NIC_a \rightarrow S_1 \rightarrow S_2 \rightarrow S_4 \rightarrow NIC_b \rightarrow B(CTXT_2)$ including the in-NIC delays is easily reconstructed. Since each component logs data into its local buffer, there is no need for concurrency control. There is no need for perfectly synchronized clocks across the system; but, we expect the components to be close enough in time via standard protocols such as NTP.

*Alternative uses:* Our approach samples a randomly chosen transaction in a window of N transactions. Alternatively, we may also sample the exact $N^{th}$ transaction. In fact, a precise, predetermined, transaction may be sampled, if desired. Instead of the software at the source of a transaction enabling the PM tag, any component may choose to enable the PM tag and capture the partial path. Although we suggested unsynchronized sampling from endpoints, we do not preclude sampling in a synchronized manner, which is useful for debugging purposes. Our approach associates metrics to the source-code location that initiated a transaction. We do not preclude associating metrics to some other place in the source code, e.g., a network wait event associated with a non-blocking transaction.

## 4 Implementation

We implemented our network performance monitoring prototype using the SST/-Macro event-driven network simulator framework [32] and open sourced it [36]. SST/Macro models hardware components such as CPU, memory, NIC, switch, crossbar. The networking components of SST/Macro are mature with various, configurable network topologies, bandwidths, latencies, and algorithms of packet-based routing and arbitration, ideally suited for our evaluation. SST/Macro is easy to extend with additional hardware and software components, which was necessary for our extensions. SST/Macro is driven by "skeleton" C++ code that mimics an HPC C++ code written using MPI and needs trivial or no modifications to work with SST/Macro.

We enhanced SST/Macro in the following ways. We introduced a new flag (PM bit) in SST/Macro packet format. We extended SST/Macro NIC and switch hardware components with the additional capability to log “marked” packets to a bounded SRAM buffer. We chose a bounded buffer of 2 KB in each router and NIC. We introduced a new hardware subcomponent “drainer” in NICs and routers, which reads the performance data accummulated in a local bounded SRAM buffer and transfers it to an on-component management software. The management software NFS mounts a file on the remote data collection server and drains the incoming performance logs to the server.

Additionally, we also implemented extensions to the NIC-software interface to express the ability to track a message. We extended the NIC with the ability to mark one out of N packets with the PM bit if the command issued from the CPU carried the TM flag and append its log in a local SRAM.

We drive the profiling with a software profiler in SST that uses random sampling to determine if a message needs to be monitored. If so, it sets a special TM tag when it issues a command to its NIC. Also, the CPU profiler collects the calling context and logs the CPU metrics about the message in a per-CPU log file.

The postmortem analysis inspects the log files to reconstruct the path taken by each marked packet by each endpoint and associates performance metrics to each hop on the path as described in the previous section. The output of our postmortem analysis is a set of files containing the path information of all the marked messages. The path information also contains the performance metrics attributed to each hop along the path.

To visualize how the application behaves, we generate a heatmap and a set of stacked bar graphs from the performance metrics using a graphing software called Plotly [37]. Figure 3a in Appendix B shows the heatmap for an example NCAST program. The heatmap shows the total time taken by each marked packet to travel from the source CPU to the destination CPU. The points on the x-axis correspond to the time at which messages were initiated. The points on the y-axis correspond to the processes that sent the messages. A point on the heatmap that is darker than other points signifies the message took relatively longer to travel from the source to the destination. In addition to the heatmap, we also generate a set of stacked bar graphs, one for each process that initiated a message in the program. Figure 3b shows a bar graph for process 97 in the ncast program. Each bar represents the cumulative time spent by the message in each network component along its path and each stack in a bar represents the time spent at each network component. A large stack in a bar shows that the message was stuck in the component for a long time. To summarize, we can use the heatmap to identify what messages were delayed, and then use the stacked bar graph corresponding to the process that initiated that message to identify network component that caused the delay.

## 5 Evaluation

We evaluated our prototype implementation to answer the following questions: (1) how effective is our prototype in finding performance bottlenecks in the network due to the application? (2) does our prototype monitor network traffic with low overhead? All our experiments were run on a four socket, 15-core Intel Xeon E7-4890 machine clocked at 2.8 GHz and containing 1 TB DRAM. Our setup simulated the NERSC Edison [20] system with the Dragonfly [18,19] topology containing 5586 compute nodes.

*Effectiveness:* To evaluate the effectiveness of our prototype, we executed it with an MPI skeleton program and ran with 4096 MPI ranks. In the skeleton program—NCAST—a single MPI process (rank 42) is bombarded with multiple large (4 MB) messages from all the other MPI processes in the network. As a large number of messages are sent to a single node, the NIC at the destination CPU becomes a bottleneck. Also, since all packets would flow through a single network switch before reaching the destination, the switch at the last hop becomes congested. Our goal is to use our prototype implementation to precisely identify the network component that is the bottleneck in the NCAST program.

Figure 3 in Appendix B shows the graphs generated by our prototype after executing the NCAST program. The heatmap in Fig. 3a in Appendix B reveals a surprising and non-obvious performance problem—the messages are all serialized; the MPI ranks are sending messages one after another, resulting in the diagonal in the heatmap. Samples from all CPUs except for CPU 42 are sparse and CPU 42 samples show that it is continuously sending messages to other processes, which is reflected in the thick horizontal line in the heatmap. The reason for serialization is the large message size sent from all other nodes. For large messages, each MPI process sends a short notification message to the target (rank 42); and the target one-by-one fetches the large message from the sources. The concurrency gets completely destroyed—a subtle anomaly invisible in the CPU-only profiles but distinctly visible in full network telemetry.

On the diagonal, we can see that the points above CPU 80 appear darker which means that those messages take relatively longer than the earlier messages. This shows that the messages that are being sent later are getting delayed at either the destination or at a network switch. Figure 3b in Appendix B shows the stacked bar graph of CPU 97. The large stack in the bar graph represents the time spent at switch 21. Switch 21 directly connects to node 42 which is receiving messages from all other nodes. Hence the stack is large since all the packets are queued at switch 21. We observe a similar pattern in the bar graphs corresponding to all other CPUs after CPU 80. This shows that all the messages are queued up at switch 21, which has become chocked.

*Efficiency:* We evaluate the efficiency of our network performance monitoring scheme by measuring the simulation and wall clock time on three MPI skeletons: NCAST, broadcast, and a mutiapp. We execute each application five times and report the geometric mean of the overhead. The skeletons were designed such

that their simulation time was at least one second. We tracked one in every hundred NIC command at each endpoint. Our measurements showed that the hardware extensions added a negligible 0.16% mean overhead to the simulation time. The wall clock time for simulation marginally increased (4.8%) over the original execution without our extensions to SST/Macro. The average size of the log files generated for the three applications is 61 MB.

## 6 Conclusions

Application developers better understand performance when measurements are attributed back to the source code. However, it is hard to attribute performance measurement data from myriad autonomous, asynchronously operating hardware components in an HPC system back to application source-code. Traditional profilers have either focused only on CPU-side hardware measurements for source-code attribution or focused on network-side hardware measurements without source-code attribution.

We developed a protocol extension to track the flow of packets and collect hardware performance data in the emerging memory-semantic-based communication protocol—Gen-Z. We enhanced the router and NIC hardware and management software with additional components for logging performance data. We enhanced traditional CPU profilers to unify CPU profiles with telemetry from networking hardware. Our sampling-based scheme implemented in the SST/-Macro simulator shows promise of our technique in offering a unified system-wide performance insights for application developers.

Our future work involves extensively evaluating our methods on serious workloads, working with hardware development teams to incorporate our proposed extensions, and working with software profiling tools to best utilize the network telemetry.

**Acknowledgments.** This work was supported (in part) by the US Department of Energy (DOE) under Cooperative Agreement DE-SC0012199, the Blackcomb 2 Project.

# A NWChem Profiles from HPCToolkit

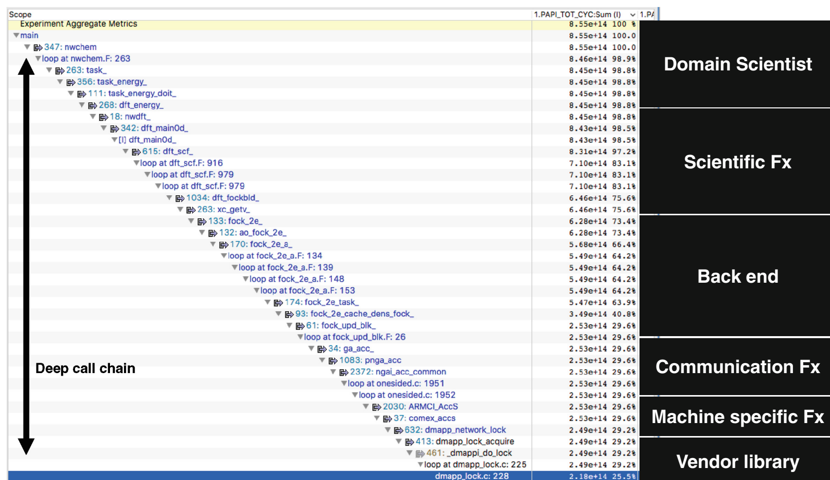
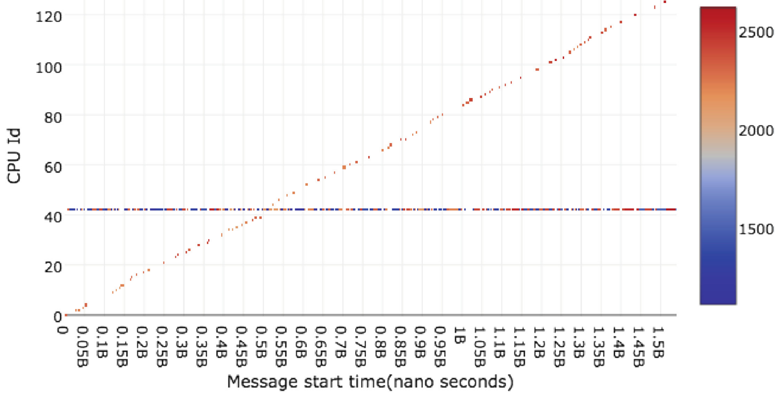
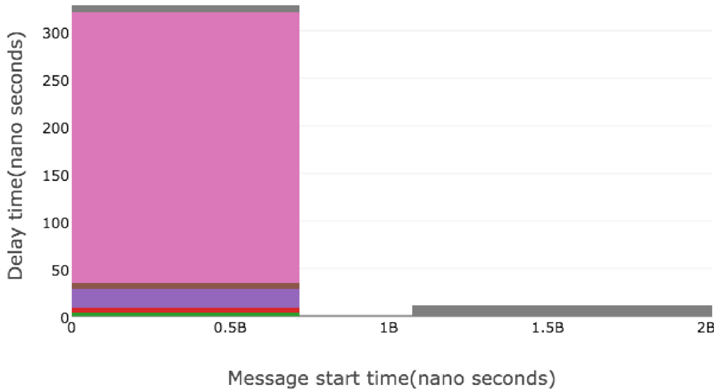



**Fig. 2.** CPU execution hotspot in NWChem running on NERSC Edison with 1024 MPI ranks captured via HPCToolkit [5] profiler. 25% of execution on all MPI processes waste time waiting to acquire remote locks embedded deep inside many layers of host code. The cause of the lock waiting despite good load balance is unknown since CPU profiles do not capture networking hardware component internals.

## B Profiles of NCAST Program




**(a)** Heatmap showing the time taken by each marked packet in the NCAST program. Sample points that are darker in color correspond to messages that were delayed the most.




**(b)** The stacked bar graph of process 97 in the NCAST program. The colored stacks in each bar represent the delay at each hop of the packet.

**Fig. 3.** Figure shows the visualization graphs generated for the NCAST program running 4096 MPI ranks.

## References


1. Oliker, L., Canning, A., Carter, J., Shalf, J., Ethier, S.: Scientific application performance on leading scalar and vector supercomputing platforms. Int. J. High Perform. Comput. Appl. **22**(1), 5–20 (2006)
2. Dongarra, J., Heroux, M.A.: Toward a new metric for ranking high performance computing systems. Sandia report, SAND2013-4744 312, p. 150 (2013)
3. Egawa, R., Komatsu, K., Momose, S., Isobe, Y., Musa, A., Takizawa, H., Kobayashi, H.: Potential of a modern vector supercomputer for practical applications: performance evaluation of SX-ACE. J. Supercomput., March 2017
4. Intel Inc.: Intel VTune. https://software.intel.com/en-us/intel-vtune-amplifier-xe
5. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: HPCToolkit: tools for performance analysis of optimized parallel programs. Concurr. Comput. Pract. Exp. **22**(6), 685–701 (2010)
6. Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The Scalasca performance toolset architecture. Concurr. Comput. Pract. Exp. **22**(6), 702–719 (2010)
7. Shende, S.S., Malony, A.D.: The Tau parallel performance system. Int. J. High Perform. Comput. Appl. **20**(2), 287–311 (2006)
8. Oracle Inc.: Oracle Solaris Studio. http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html
9. Intel Inc.: Intel Trace Analyzer and Collector, October 2017. https://software.intel.com/en-us/intel-trace-analyzer
10. Allinea Inc.: Allinea MAP - C/C++ profiler and Fortran profiler for high performance Linux code, October 2017. https://www.allinea.com/products/map
11. Liu, X., Mellor-Crummey, J.: A data-centric profiler for parallel programs. In: Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis, vol. 28 (2013)
12. Rane, A., Browne, J.: Enhancing performance optimization of multicore chips and multichip nodes with data structure metrics. In: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques, Minneapolis, MN, USA. IEEE Computer Society (2012)
13. Böhme, D., Geimer, M., Arnold, L., Voigtlaender, F., Wolf, F.: Identifying the root causes of wait states in large-scale parallel applications. ACM Trans. Parallel Comput. **3**(2), 11:1–11:24 (2016)
14. Isaacs, K.E., Gamblin, T., Bhatele, A., Schulz, M., Hamann, B., Bremer, P.T.: Ordering traces logically to identify lateness in message passing programs. IEEE Trans. Parallel Distrib. Syst. **27**(3), 829–840 (2016)
15. Weber, M., Brendel, R., Hilbrich, T., Mohror, K., Schulz, M., Brunst, H.: Structural clustering: a new approach to support performance analysis at scale. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 484–493, May 2016
16. Isaacs, K.E., Giménez, A., Jusufi, I., Gamblin, T., Bhatele, A., Schulz, M., Hamann, B., Bremer, P.T.: State of the art of performance visualization. In: Borgo, R., Maciejewski, R., Viola, I. (eds.) EuroVis - STARs. The Eurographics Association (2014)
17. Valiev, M., Bylaska, E., Govind, N., Kowalski, K., Straatsma, T., Dam, H.V., Wang, D., Nieplocha, J., Apra, E., Windus, T., de Jong, W.: NWChem: a comprehensive and scalable open-source solution for large scale molecular simulations. Comput. Phys. Commun. **181**(9), 1477–1489 (2010)

18. Kim, J., Dally, W.J., Scott, S., Abts, D.: Technology-driven, highly-scalable dragonfly topology. In: Proceedings of the 35th Annual International Symposium on Computer Architecture, ISCA 2008, Washington, DC, USA, pp. 77–88. IEEE Computer Society (2008)
19. Alverson, B., Kaplan, L., Roweth, D.: Cray XC Series Network. http://www.cray.com/sites/default/files/resources/CrayXCNetwork.pdf
20. National Energy Research Scientific Computing Center: Edison. http://www.nersc.gov/users/computational-systems/edison/
21. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst. **9**(1), 21–65 (1991)
22. Gen-Z Consortium: Gen-Z: Draft Core Specification, July 2017. http://genzconsortium.org/specifications/draft-core-specification-july-2017/
23. Linux wiki: Linux perf tool. https://perf.wiki.kernel.org/index.php/Main_Page
24. Zaki, O., Lusk, E., Gropp, W., Swider, D.: Toward scalable performance visualization with jumpshot. High Perf. Comput. Appl. **13**(2), 277–288 (1999)
25. Karrels, E., Lusk, E.: Performance analysis of MPI programs. In: Dongarra, J., Tourancheau, B. (eds.) Proceedings of the Workshop on Environments and Tools For Parallel Scientific Computing, pp. 195–200. SIAM Publications (1994)
26. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: The vampir performance analysis tool-set. Tools High Perf. Comput. 139–155 (2008)
27. McDonald, N.: SuperSim: a flexible event-driven cycle-accurate network simulator. https://github.com/HewlettPackard/supersim
28. Carothers, C.: ROSS: Rensselaer's Optimistic Simulation System. https://github.com/carothersc/ROSS/wiki
29. Carothers, C.D., Bauer, D., Pearce, S.: ROSS: a high-performance, low memory, modular time warp system. In: Proceedings of the Fourteenth Workshop on Parallel and Distributed Simulation, PADS 2000, Washington, DC, USA, pp. 53–60. IEEE Computer Society (2000)
30. Liu, N., Carothers, C., Cope, J., Carns, P., Ross, R.: Model and simulation of exascale communication networks. J. Simul. **6**(4), 227–236 (2012)
31. Jain, N., Bhatele, A., White, S., Gamblin, T., Kale, L.V.: Evaluating hpc networks via simulation of parallel workloads. In: SC16: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 154–165, November 2016
32. Rodrigues, A.F., Hemmert, K.S., Barrett, B.W., Kersey, C., Oldfield, R., Weston, M., Risen, R., Cook, J., Rosenfeld, P., CooperBalls, E., Jacob, B.: The structural simulation toolkit. SIGMETRICS Perform. Eval. Rev. **38**(4), 37–42 (2011)
33. So-In, C.: A survey of network traffic monitoring and analysis tools. https://www.cse.wustl.edu/~jain/cse567-06/ftp/net_traffic_monitors3.pdf
34. Cisco Inc.: Cisco IOS NetFlow. https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html
35. sFlow organization: sFlow. http://www.sflow.org/
36. Hewlett Packard Labs: Network Performance Monitoring (NWPM) Tool. https://github.com/HewlettPackard/genz_tools_network_monitoring
37. Plotly Technologies Inc.: Collaborative data science (2015). https://plot.ly

# Performance and Energy Usage of Workloads on KNL and Haswell Architectures

Tyler Allen[1(✉)], Christopher S. Daley[2], Douglas Doerfler[2], Brian Austin[2], and Nicholas J. Wright[2]

[1] Clemson University, Clemson, SC, USA
tnallen@clemson.edu
[2] Lawrence Berkeley National Laboratory, Berkeley, CA, USA
{csdaley,dwdoerf,baustin,njwright}@lbl.gov

**Abstract.** Manycore architectures are an energy-efficient step towards exascale computing within a constrained power budget. The Intel Knights Landing (KNL) manycore chip is a specific example of this and has seen early adoption by a number of HPC facilities. It is therefore important to understand the performance and energy usage characteristics of KNL. In this paper, we evaluate the performance and energy efficiency of KNL in contrast to the Xeon (Haswell) architecture for applications representative of the workload of users at NERSC. We consider the optimal MPI/OpenMP configuration of each application and use the results to characterize KNL in contrast to Haswell. As well as traditional DDR memory, KNL contains MCDRAM and we also evaluate its efficacy. Our results show that, averaged over our benchmarks, KNL is 1.84× more energy efficient than Haswell and has 1.27× greater performance.

**Keywords:** Benchmarking · Power consumption · Energy
Hyperthreads · Manycore architecture · Intel Knights Landing
Haswell

## 1 Introduction

Manycore architectures promise significant gains in application performance and energy efficiency over past High Performance Computing (HPC) architectural designs. The first mainstream manycore architecture, Intel Knights Landing (KNL), already boasts early adoption in several clusters hosted by major HPC facilities, including Cori at the National Energy Research Scientific Computing (NERSC) Center [16,30], Trinity at Los Alamos National Laboratory (LANL) [15], and Theta at Argonne National Laboratory (ANL) [39]. These first pre-exascale manycore systems are intended to pave the way towards exascale-at-twenty-megawatt computing for the DoE [19]. In this paper, we use modern HPC workloads to evaluate how well the KNL satisfies the trajectory requirements for exascale.

© Springer International Publishing AG 2018
S. Jarvis et al. (Eds.): PMBS 2017, LNCS 10724, pp. 236–249, 2018.
https://doi.org/10.1007/978-3-319-72971-8_12

We use applications representative of the NERSC workload to characterize and quantify the performance and efficiency benefits of KNL. This application-suite is a broad composite of prominent applications used on NERSC systems [10] with some traditional micro-benchmarks for fine-grain evaluation. We evaluate this workload on the current NERSC flagship supercomputer, Cori. We use the most popular KNL configuration: KNL with MCDRAM memory configured to operate as a cache, where MCDRAM is Intel's on-package high bandwidth memory. We contrast this by evaluating the same workload on Cori Haswell nodes at the same node count. To make this a fair comparison, we use optimal Message Passing Interface (MPI) and Open Multi-Processing (OpenMP) configurations for both KNL and Haswell, as well as compiling with the appropriate vector Instruction Set Architecture (ISA) (AVX512, AVX2 respectively). We also contrast against KNL runs that use only Double Data Rate (DDR) memory. Henceforth, we will refer to the KNL cache mode configuration as KNL-Cache and the KNL flat mode configurations as KNL-DDR and KNL-MCDRAM depending on whether data is explicitly allocated in DDR or MCDRAM.

Our key findings are:

– Optimized micro-benchmark configurations run 1.5 to 4.0 times faster on KNL nodes than Haswell nodes.
– The STREAM memory bandwidth is approximately 140 GiB/s less when using KNL-Cache compared to KNL-MCDRAM. In addition, cache conflicts can reduce KNL-Cache memory bandwidth by up to an additional 100 GiB/s.
– The average performance of the full benchmark suite is only better on KNL compared to Haswell when using KNL-Cache. KNL-Cache improves the performance of every single benchmark compared to KNL-DDR. This indicates the value of MCDRAM to overall performance.
– The average energy efficiency of the full benchmark suite is better on KNL than Haswell in both KNL-Cache and KNL-DDR modes. This indicates that there can still be an overall win in terms of work done within a given energy budget by using an architecture with small cores and no MCDRAM.
– The performance and energy gains of hyperthreads are dependent on the individual application, indicating the importance of understanding application characteristics.

## 2 Related Work

Barnes et al. provide an initial evaluation of strictly performance of some NERSC applications on KNL in comparison to Haswell [12]. A similar evaluation was performed on Trinity by Agelastos et al. [7]. Parker et al. perform a KNL performance study in their evaluation of Theta [31]. The Theta evaluation analyzes power for two micro-benchmarks, whereas we provide a more in depth analysis of the power efficiency improvement of KNL compared to Haswell. Several authors have analyzed the efficacy of hyperthreads, but this work predates KNL and do not give insights into the performance/power benefits on KNL [14,18,37,41].

Lawson et al. evaluated dynamic voltage and frequency scaling (DVFS) techniques on KNL for power limiting and provide a power model, whereas we evaluate efficiency gains and optimal usage without limiting power [28]. Peng et al. [33] evaluate the performance of micro-benchmarks and mini-apps on a KNL system with a focus on the performance impact of various KNL memory modes, and Ramos and Hoefler create a performance model for KNL memory modes [34]. Our evaluation uses a fixed memory mode (KNL-Cache), and contrasts it with the performance of KNL-DDR to show the benefits of MCDRAM. Work has been done on providing single metrics for evaluating trade-offs between power and performance [35]. However, in this paper we optimize applications based on performance and then report the energy consumption of this configuration.

## 3 NERSC's Cori Supercomputer

The Cori supercomputer is located at the U.S. Department of Energy's Office of Science National Energy Research Scientific Computing Center [16,30]. Cori is based on the Cray XC40 architecture [17] and was deployed in two phases. Phase 1 consists of 2,388 nodes based on dual-socket, 16-core Intel E5-2698 v3 Xeon® processors clocked at 2.3 GHz and a total 128 GB of DDR4 2133 memory [4]. Phase 2 is 9,688 nodes in size and utilizes a single Intel Xeon Phi™ 7250 Knights Landing processor with 68 cores at 1.4 GHz and 96 GB of DDR4 2400 memory [3]. The two node types share a Cray Aries dragonfly high-speed network and a common storage subsystem. The KNL processor can be configured to support a variety of non-uniform memory access (NUMA) and memory modes that are meant to allow the processor to be configured to the particular needs of a given application [38]. In particular, the on-chip mesh can be configured in 3 different clustering modes: all-to-all, quadrant (quad), and sub-NUMA (with the option of 2 or 4 NUMA regions). In addition, the KNL has a 16 GB on-chip high-speed memory (MCDRAM) that can be configured as a directly addressable memory region in its own NUMA region (flat) or it can be used as a cache for DDR. The majority of KNL nodes on Cori are configured to quad/cache mode as it provides an easy on-ramp for users coming from traditional Xeon® nodes. However, Cori does support the other available modes dynamically via a subset of the nodes allocatable in a dedicated *reboot queue* of the scheduler for those users that want to set a mode better suited to the needs of their application.

## 4 Method and Instrumentation

The key areas of evaluation and characterization for the NERSC workload on Cori are performance per watt and the efficacy of unique features of the KNL architecture including MCDRAM and four hardware-threads-per-core. We used the Integrated Performance Monitoring (IPM) profiling tool in our experiments

to map the performance over the parameter space of the selected NERSC workload applications on KNL. We also performed these experiments on KNL without utilizing MCDRAM and on Haswell nodes. In this section we describe our experimental design and methodology and discuss IPM and the enhancements we made to IPM for our experiments.

### 4.1 Integrated Performance Monitoring Tool

We introduce the power monitoring enhancement to the IPM library as part of this work[1] [20–22]. IPM is a NERSC profiling library for high-performance applications, first introduced by Fürlinger et al. [21]. IPM aggregates several low level interfaces to provide a large quantity of performance data. In order to adapt IPM to the new manycore architecture HPC paradigm and energy-constrained computing, we added energy measurement to the IPM feature set.

IPM now supports measurement of energy consumption over the course of application execution. The newly added IPM `PMON` module is included in IPM by using the −`enable-pmon` configure option. The PMON module follows the standard IPM module interface and requires only the additional `IPM_PMON`= 1 environment flag at runtime in order to activate energy collection. Energy measurements through IPM are currently only supported on Cray systems through the Cray power monitoring and management stack [29]. IPM measures energy over the full duration of the application, or programmer-specified sections. By default, the energy counter is initialized when the application calls `MPI_Init` and accumulated until the application calls `MPI_Finalize`. We are able to measure energy at three sources: the full node energy prior to distribution, CPU power, and DDR memory energy [36]. As a consequence of the architecture, the KNL MCDRAM energy consumption is included in the CPU power measurement and cannot be measured separately [36]. Using this energy measurement, we are also able to compute the average memory, CPU, and total power consumption of an application. We derive an average power value (W) by dividing this accumulated energy (J) by the application wall clock time (s). IPM energy information can be found in both the IPM standard output summary information and the standard IPM XML output files. Users should note that in the XML output, every rank provides an energy reading for the entire node. Therefore, user post-processing should appropriately handle the case where multiple ranks from the same node producing duplicate values.

### 4.2 Experiment Methodology

We designed our experiments to show the effect of the following parameter variations:

- Varying MPI ranks-per-node and OpenMP threads-per-rank with a fixed amount of total concurrency

[1] IPM is open-source and available on github: https://github.com/nerscadmin/IPM.

- Varying the total concurrency to evaluate the effects of using one, two, three, and four threads-per-physical-core
- Using KNL nodes in various modes, including KNL-Cache, KNL-MCDRAM and KNL-DDR, and Haswell nodes

We performed experiments using every combination of the parameter variations. The lightweight attribute of IPM makes it possible to collect all of the data required from each of these experiments in a single run per configuration. All applications are built with `icc` version 17.0.2.174 using the `-xMIC-AVX512` optimization flag to enable the 512-bit vector optimizations for KNL. (Haswell builds used the `-xAVX2` flag instead.)

With the rise of manycore architectures, MPI/OpenMP hybrid parallelism is seeing increased popularity. We designed an experiment to explore the performance relationship between MPI/OpenMP for our applications. We first fixed the amount of total concurrency such that there is only one MPI rank or OpenMP thread per core. One thread or process-per-core is exactly 68 threads-per-node on Cori. At times we needed to use MPI counts and/or OpenMP thread counts in powers of 2 because of the domain decomposition requirements of the applications. Experimentally, most applications did not receive a significant performance increase when using 68 cores instead of 64. To evaluate with two-or-more threads per core, we simply used appropriate values for ranks-per-node and threads-per-rank. We used the `OMP_PLACES=threads` and `OMP_PROC_BIND=spread` environment variable settings for our experiments to ensure OpenMP threads are not grouped on a single core. We also used the *Slurm* option `--cpu_bind=cores` to ensure MPI ranks are spread across different cores.

We conducted our experiments using KNL and Haswell nodes. We used the flat KNL modes to evaluate the benefit of MCDRAM for performance and power consumption. The `numactl` tool is used to explicitly allocate memory in MCDRAM or DDR. We also used Haswell nodes to compare the performance, but also the energy efficiency, of KNL nodes to Haswell nodes. This required us to modify our experiment somewhat, as Haswell nodes have fewer cores and threads-per-core than KNL. We only use one and two threads-per-core for Haswell experiments, and limit our MPI concurrencies accordingly. Haswell nodes also do not support the `AVX512` instruction set, and so `AVX2` optimizations are used instead.

### 4.3 Applications and Micro-benchmarks

Table 1 lists the applications used for this study. For application descriptions please refer to the respective references. The table lists the details of decomposition in addition to a brief description of the level of tuning performed for the KNL processor. Minimal refers to no source codes changes, but compiler optimizations may have been performed. Significant refers to code restructuring, thread and/or vectorization optimizations performed specifically for the KNL architecture.

**Table 1.** Application benchmark details

| Application | Science area | Level of tuning | Nodes | Rnks-Thds/Rnk | | GiB/ |
|---|---|---|---|---|---|---|
| | | | | HSW | KNL | node |
| STREAM [6] | Memory bandwidth | Minimal | 1 | 32t | 68t | 6.7 |
| RandN [11] | Random memory access | Minimal | 1 | 64t | 256t | 6.5 |
| DGEMM [1] | Dense linear algebra | Intel MKL | 1 | 32t | 136t | 2.3 |
| GTC-P [2,40] | Fusion | Moderate OpenMP | 8 | 32r-1t | 32r-8t | 0.17 |
| MILC [13] | Quantum chromodynamics | QPhiX dslash solver | 8 | 32r-1t | 32r-2t | 8.3 |
| Nyx-AMReX [9] | Cosmology | Minimal | 2 | 16r-4t | 16r-16t | 58 |
| Castro-AMReX [8] | Astrophysics | Minimal | 4 | 32r-1t | 32r-2t | 6.5 |
| Quantum Espresso [23] | Quantum chemistry | Significant | 4 | 4r-8t | 4r-16t | 21 |
| BD-CATS [32] | Data analytics for cosmology | Minimal | 16 | 16r-4t | 16r-16t | 5.4 |

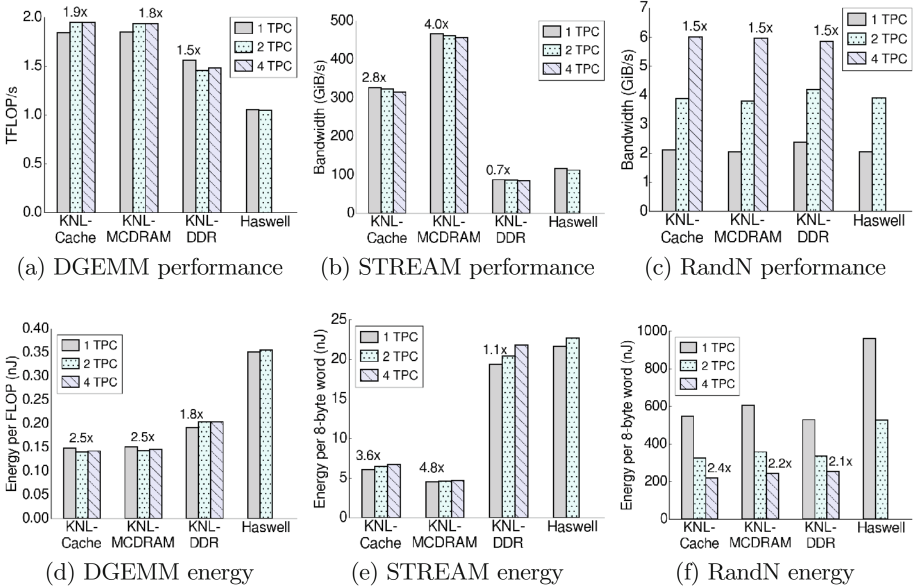
## 5 Results

### 5.1 Micro-benchmarks

In this section we evaluate the performance and energy efficiency of the DGEMM, STREAM and RandN micro-benchmarks on the KNL and Haswell architectures. The micro-benchmarks stress peak floating point, sequential memory access, and random memory access performance, respectively. The KNL results are obtained in the three modes discussed earlier: KNL-Cache, KNL-MCDRAM and KNL-DDR. We choose the problem size so that the memory footprint per compute node is less than the memory capacity of MCDRAM. This allows us to evaluate the benefit of MCDRAM under ideal circumstances.

Figure 1 shows the absolute performance and energy efficiency of the micro-benchmarks. The top row of the figure shows the performance of the benchmarks in the appropriate units: floating point rate for DGEMM in units of TFLOP/s and memory bandwidth for STREAM and RandN in units of GiB/s. The bottom row of the figure shows the energy cost of performing a single operation in the benchmark: 1 double precision FLOP in DGEMM and transferring a single 8-byte word to/from memory in STREAM and RandN. The energy metric is calculated by dividing the average power usage in Watts (J/s) by the micro-benchmark performance metric printed to standard output. In the case of DGEMM, we divide [J/s] by [FLOP/s] to obtain [J/FLOP].

The results in this figure show that the optimal micro-benchmark configurations run 1.5 to 4.0 times faster on the KNL architecture compared to the Haswell architecture. Our best KNL performance results are a peak floating point rate of 2 TFLOP/s, a sequential memory bandwidth of 466 GiB/s and a random memory access bandwidth of 6 GiB/s. The STREAM micro-benchmark performs better in KNL-MCDRAM mode than in KNL-Cache mode (466 GiB/s vs 327 GiB/s), indicating that KNL-Cache mode introduces some overhead for memory-bandwidth bound applications. This is because streaming stores incur an additional memory read in KNL-Cache mode to determine whether a line is already present in MCDRAM [27, p. 565]. We have found that switching off streaming stores with the compiler option `-qopt-streaming-stores=never`




(a) DGEMM performance (b) STREAM performance (c) RandN performance

(d) DGEMM energy (e) STREAM energy (f) RandN energy

**Fig. 1.** Performance and energy efficiency of DGEMM, STREAM and RandN micro-benchmarks on KNL and Haswell architectures with various counts of threads per core (TPC). The optimal KNL configuration in each mode is marked with the relative improvement over the optimal Haswell configuration.

reduces STREAM performance in KNL-MCDRAM mode to 347 GiB/s, explaining most of the performance difference. All micro-benchmarks perform worse in KNL-DDR mode than the other modes, with the exception of RandN in 1 and 2 threads per core (TPC) configurations (Fig. 1c). These lower concurrency configurations are impacted by the higher memory latency of MCDRAM [33]. It is only at the highest TPC concurrency where MCDRAM can support the large number of memory requests needed to hide the memory latency disadvantage. The DGEMM and STREAM micro-benchmarks benefit more from MCDRAM than RandN (Figs. 1a and b). DGEMM and STREAM performance remains approximately the same when adding multiple threads per core because the floating point performance and memory bandwidth are already saturated.

The figure also shows that the optimal KNL configurations are 2.4 to 4.8 times more energy efficient than the Haswell architecture. Energy efficiency improves more than performance because of the difference in power usage of the nodes. For example, the optimally performing DGEMM configuration consumes 270 W on KNL and 360 W on Haswell[2]. The best KNL energy results are 0.15 nJ/FLOP in DGEMM, 4.5 nJ/word in STREAM and 200 nJ/word in

[2] The DGEMM power consumption is approximately 2 to 8 W higher on KNL over a range of concurrencies than the synthetic Firestarter benchmark designed to create near-peak power consumption [24].

RandN. These are significantly larger than the exascale target of 20 pJ/FLOP (i.e. 0.02 nJ/FLOP) to achieve an exaFLOP within a 20 MW power budget. Finally, the order of magnitude difference between STREAM and RandN indicates the high energy cost of random memory access workloads in traditional CPU architectures.

The evaluated Xeon Phi™ and Xeon® processors use a 14 nm and 22 nm technology size, respectively. This is a slightly unfair comparison because a smaller feature size is more energy efficient. Product sheets show that a 14 nm 16-core Xeon® Broadwell processor has a Thermal Design Power (TDP) of 115 W [5] compared to the Haswell Xeon® in Cori which has a TDP of 135 W [4]. Therefore, a rough estimate of the energy efficiency improvement over a 14 nm Xeon® can be obtained by multiplying the energy efficiency improvement in the figures by [115/135]. The optimal STREAM configuration on KNL would therefore be 4.1x more energy efficient than a 14 nm Xeon®.

## 5.2 STREAM Variability

The performance of the STREAM micro-benchmark varies considerably in KNL-Cache mode because of cache conflicts in the direct-mapped MCDRAM cache. This effect cannot be controlled and depends on the specific physical memory pages allocated to a job at runtime. It is a known issue that Intel has partially mitigated by creating a kernel module named Zonesort [25,26]. The kernel module reorders memory pages to reduce cache-conflicts and is run on Cori before every Slurm job step.

Figure 2 is a cumulative density plot showing STREAM performance over 48 trials. The results show that over 50% of trials achieve a bandwidth of 324–327 GiB/s and that there is a long tail of degradation towards 225 GiB/s. We monitored a performance counter measuring DDR traffic named `OFFCORE_RESPONSE_0:ANY_REQUEST:DDR` in each trial and found that high values correlate with poor

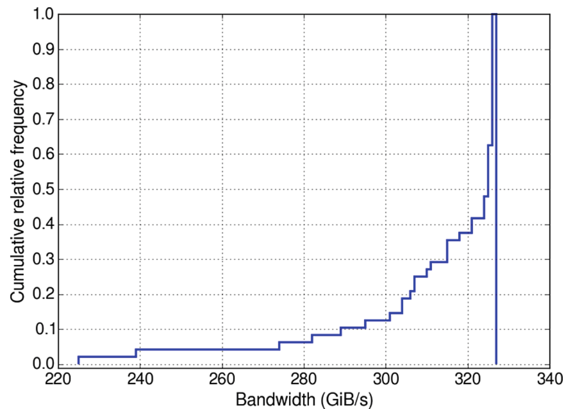


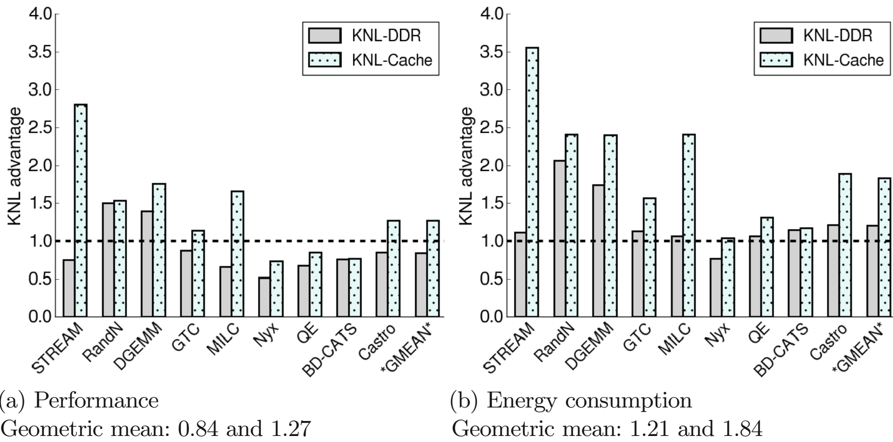

**Fig. 2.** STREAM bandwidth in KNL-Cache mode over 48 trials

STREAM memory bandwidth and high DDR memory power. This indicates that the direct-mapped cache on KNL can cause performance and energy inefficiencies and that Zonesort does not eliminate all cache-conflicts. It is significant because this effect can reduce STREAM memory bandwidth by a factor of 2 compared to the optimally performing KNL-MCDRAM configuration.

### 5.3 Performance and Energy Consumption Across Applications

In this section we compare application performance and energy consumption across architectural configurations for the application benchmark suite. We identify the fastest MPI/OpenMP configuration for each benchmark on KNL and then use this MPI count and a variable number of OpenMP threads for every experiment. Our tests are designed to show application sensitivity to memory bandwidth and hyperthreading on KNL. The later experiment studying hyperthreads is performed in KNL-cache mode.

Figure 3 summarizes the application performance and energy consumption on KNL and Haswell. The results are normalized so that values greater than 1.0 indicate that the application has a higher figure of merit on KNL than Haswell. The results show that 6 out of 9 applications perform better on the KNL node architecture. The KNL performance is best when using KNL-Cache mode in every experiment, and the KNL-DDR mode is worse than Haswell for all scientific applications, indicating the importance of MCDRAM to application performance. The greatest MCDRAM gains occur in STREAM and MILC which are applications bound by memory bandwidth. In some cases, the opposite is true: when applications like RandN and BD-CATS are dominated by random memory access, MCDRAM provides negligible gains in performance. Perhaps the most significant result is that all applications consume less energy on KNL compared



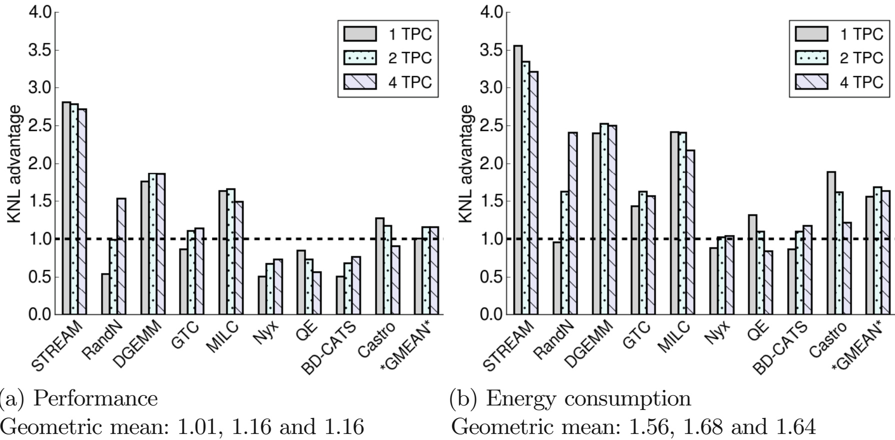

(a) Performance
Geometric mean: 0.84 and 1.27

(b) Energy consumption
Geometric mean: 1.21 and 1.84

**Fig. 3.** Figures of merit improvement of KNL relative to Haswell. The best KNL configuration is compared against the best Haswell configuration.

to Haswell. If we follow the approach from earlier then we estimate that the 1.84x energy improvement over Haswell would be a 1.56x energy improvement over a 14 nm Broadwell processor.

Figure 4 shows that application performance is more variable when changing the number of threads per core. Several applications, e.g. STREAM, Quantum Espresso and Castro, perform worse when using hyperthreads because of either resource saturation or increased overhead of using more threads. Other applications with random memory access, e.g. RandN and BD-CATS, have significant gains when using all 4 threads per core. On average, hyperthreads improve performance by approximately 16% over the optimal Haswell configuration. We find that 2 and 4 hyperthreads per core deliver similar average performance, however, we find that the 4 hyperthreads per core configuration consumes more energy than the 2 hyperthreads per core configuration. Therefore, based on energy consumption, 4 threads per core configurations are only helpful for a very specialized workload, e.g. a graph analytics workload dominated by random memory access.




(a) Performance
Geometric mean: 1.01, 1.16 and 1.16

(b) Energy consumption
Geometric mean: 1.56, 1.68 and 1.64

**Fig. 4.** Figures of merit improvement of KNL relative to Haswell. The KNL configuration at each thread count is compared against the best Haswell configuration. The KNL results are obtained in KNL-Cache mode.

## 6 Conclusions

We have shown that KNL is a solid step towards exascale efficiency, but that there is still significant progress left to be made. On the NERSC workload, we have shown that KNL improves performance for 6 out of 9 applications vs. Haswell, but manages to reduce energy consumption for every application. Also, for applications with memory locality, the MCDRAM present on KNL can provide enormous performance benefits in comparison to DDR4, and simultaneously reduces the energy-per-operation for every application. MCDRAM is a critical

feature of this architectural shift. Future architectures will need to make even greater strides towards efficiency.

## 7 Future Work

The work in this paper relied on our experience with the applications to explain the observed performance results. We plan a more thorough approach that will automatically characterize applications using hardware performance counters. We have already started to create this performance analysis framework by adding PAPI multiplexing support to IPM and developing scripts to create derived performance metrics based on this data to quantify the performance requirements of applications. This will allow us to understand at a deeper level the overall sensitivity of the larger NERSC workload to features on modern CPUs, such as MCDRAM and hyperthreads.

**Acknowledgment.** This research used resources of the National Energy Research Scientific Computing Center, 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.

## References

1. DGEMM. http://www.nersc.gov/research-and-development/apex/apex-benchmarks/dgemm/
2. GTC-P. http://www.nersc.gov/research-and-development/apex/apex-benchmarks/gtc-p/
3. Intel Xeon Phi Processor 7250 16GB, 1.40 GHz, 68 core. https://ark.intel.com/products/94035/Intel-Xeon-Phi-Processor-7250-16GB-1_40-GHz-68-core
4. Intel Xeon Processor E5–2698 v3 40M Cache, 2.30 GHz. https://ark.intel.com/products/81060/Intel-Xeon-Processor-E5-2698-v3-40M-Cache-2_30-GHz
5. Intel Xeon Processor E7–4850 v4 40M Cache, 2.10 GHz. https://ark.intel.com/products/93806/Intel-Xeon-Processor-E7-4850-v4-40M-Cache-2_10-GHz
6. STREAM: Sustainable Memory Bandwidth in High Performance Computers. https://www.cs.virginia.edu/stream/FTP/Code/
7. Agelastos, A.M., Rajan, M., Wichmann, N., Baker, R., Domino, S., Draeger, E.W., Anderson, S., Balma, J., Behling, S., Berry, M., Carrier, P., Davis, M., McMahon, K., Sandness, D., Thomas, K., Warren, S., Zhu, T.: Performance on Trinity phase 2 (a Cray XC40 utilizing Intel Xeon Phi processors) with acceptance applications and benchmarks. In: Cray User Group CUG, May 2017. https://cug.org/proceedings/cug2017_proceedings/includes/files/pap138s2-file1.pdf
8. Almgren, A.S., Beckner, V.E., Bell, J.B., Day, M.S., Howell, L.H., Joggerst, C.C., Lijewski, M.J., Nonaka, A., Singer, M., Zingale, M.: CASTRO: A new compressible astrophysical solver. I. hydrodynamics and self-gravity. Astrophys. J. **715**, 1221–1238 (2010)
9. Almgren, A.S., Bell, J.B., Lijewski, M.J., Lukić, Z., Andel, E.V.: Nyx: A massively parallel AMR code for computational cosmology. Astrophys. J. **765**(1), 39 (2013). http://stacks.iop.org/0004-637X/765/i=1/a=39

10. APEX Benchmark Distribution and Run Rules. http://www.nersc.gov/research-and-development/apex/apex-benchmarks/
11. Austin, B., Wright, N.J.: Measurement and interpretation of microbenchmark and application energy use on the Cray XC30. In: Proceedings of the 2nd International Workshop on Energy Efficient Supercomputing, pp. 51–59. IEEE Press (2014)
12. Barnes, T., Cook, B., Deslippe, J., Doerfler, D., Friesen, B., He, Y., Kurth, T., Koskela, T., Lobet, M., Malas, T., Oliker, L., Ovsyannikov, A., Sarje, A., Vay, J.L., Vincenti, H., Williams, S., Carrier, P., Wichmann, N., Wagner, M., Kent, P., Kerr, C., Dennis, J.: Evaluating and optimizing the NERSC workload on knights landing. In: 2016 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), pp. 43–53, November 2016
13. Bauer, B., Gottlieb, S., Hoefler, T.: Performance modeling and comparative analysis of the MILC Lattice QCD application su3_rmd. In: Proceedings CCGRID2012: IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (2012)
14. Coghlan, S., Kumaran, K., Loy, R.M., Messina, P., Morozov, V., Osborn, J.C., Parker, S., Riley, K.M., Romero, N.A., Williams, T.J.: Argonne applications for the IBM Blue Gene/Q, Mira. IBM J. Res. Dev. **57**(1/2), 12:1–12:11 (2013)
15. LANL Trinity Supercomputer. http://www.lanl.gov/projects/trinity/
16. NERSC Cori Supercomputer. https://www.nersc.gov/systems/cori/
17. Cray XC Series Supercomputers. http://www.cray.com/products/computing/xc-series
18. Evangelinos, C., Walkup, R.E., Sachdeva, V., Jordan, K.E., Gahvari, H., Chung, I.H., Perrone, M.P., Lu, L., Liu, L.K., Magerlein, K.: Determination of performance characteristics of scientific applications on IBM Blue Gene/Q. IBM J. Res. Dev. 57(1), 99–110 (2013). https://doi.org/10.1147/JRD.2012.2229901
19. The Opportunities and Challenges of Exascale Computing. https://science.energy.gov/~/media/ascr/ascac/pdf/reports/Exascale_subcommittee_report.pdf
20. Fuerlinger, K., Wright, N.J., Skinner, D.: Effective performance measurement at petascale using IPM. In: 2010 IEEE 16th International Conference on Parallel and Distributed Systems, pp. 373–380, December 2010
21. Fürlinger, K., Wright, N.J., Skinner, D.: Performance analysis and workload characterization with IPM. In: Müller, M., Resch, M., Schulz, A., Nagel, W. (eds.) Tools for High Performance Computing 2009, pp. 31–38. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11261-4_3
22. Fürlinger, K., Wright, N.J., Skinner, D., Klausecker, C., Kranzlmüller, D.: Effective holistic performance measurement at petascale using IPM. In: Bischof, C., Hegering, H.G., Nagel, W., Wittum, G. (eds.) Competence in High Performance Computing 2010, pp. 15–26. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-24025-6_2
23. Giannozzi, P., Baroni, S., Bonini, N., Calandra, M., Car, R., Cavazzoni, C., Ceresoli, D., Chiarotti, G.L., Cococcioni, M., Dabo, I., Dal Corso, A., de Gironcoli, S., Fabris, S., Fratesi, G., Gebauer, R., Gerstmann, U., Gougoussis, C., Kokalj, A., Lazzeri, M., Martin-Samos, L., Marzari, N., Mauri, F., Mazzarello, R., Paolini, S., Pasquarello, A., Paulatto, L., Sbraccia, C., Scandolo, S., Sclauzero, G., Seitsonen, A.P., Smogunov, A., Umari, P., Wentzcovitch, R.M.: QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials. J. Phys. Condens. Matter **21**(39), 395502 (19pp) (2009). http://www.quantum-espresso.org

24. Hackenberg, D., Oldenburg, R., Molka, D., Schöne, R.: Introducing FIRESTARTER: a processor stress test utility. In: 2013 International Green Computing Conference Proceedings, pp. 1–9, June 2013
25. He, Y., Cook, B., Deslippe, J., Friesen, B., Gerber, R., Hartman-Baker, R., Koniges, A., Kurth, T., Leak, S., Yang, W.S., Zhao, Z.: Preparing NERSC users for Cori, a Cray XC40 system with Intel many integrated cores. In: Cray User Group CUG, May 2017. https://cug.org/proceedings/cug2017_proceedings/includes/files/pap161s2-file1.pdf
26. Hill, P., Snyder, C., Sygulla, J.: KNL system software. In: Cray User Group CUG, May 2017. https://cug.org/proceedings/cug2017_proceedings/includes/files/pap169s2-file1.pdf
27. Jeffers, J., Reinders, J., Sodani, A.: Intel Xeon Phi Processor High Performance Programming: Knights, Landing edn. Morgan Kaufmann, Boston (2016)
28. Lawson, G., Sundriyal, V., Sosonkina, M., Shen, Y.: Runtime power limiting of parallel applications on Intel Xeon Phi Processors. In: 2016 4th International Workshop on Energy Efficient Supercomputing (E2SC), pp. 39–45, November 2016
29. Martin, S.J., Kappel, M.: Cray XC30 power monitoring and management. In: Cray User Group 2014 Proceedings (2014)
30. National Energy Research Scientific Computing Center. https://www.nersc.gov
31. Parker, S., Morozov, V., Chunduri, S., Harms, K., Knight, C., Kumaran, K.: Early evaluation of the Cray XC40 Xeon Phi System 'Theta' at Argonne. In: Cray User Group CUG, May 2017. https://cug.org/proceedings/cug2017_proceedings/includes/files/pap113s2-file1.pdf
32. Patwary, M.M.A., Dubey, P., Byna, S., Satish, N.R., Sundaram, N., Lukić, Z., Roytershteyn, V., Anderson, M.J., Yao, Y., Prabhat: BD-CATS: big data clustering at trillion particle scale. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC 2015, pp. 1–12. ACM Press, New York (2015). http://dl.acm.org/citation.cfm?doid=2807591.2807616
33. Peng, I.B., Gioiosa, R., Kestor, G., Laure, E., Markidis, S.: Exploring the Performance Benefit of Hybrid Memory System on HPC Environments. CoRR abs/1704.08273 (2017). http://arxiv.org/abs/1704.08273
34. Ramos, S., Hoefler, T.: Capability models for manycore memory systems: a case-study with Xeon Phi KNL. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 297–306, May 2017
35. Roberts, S.I., Wright, S.A., Fahmy, S.A., Jarvis, S.A.: Metrics for energy-aware software optimisation. In: Kunkel, J.M., Yokota, R., Balaji, P., Keyes, D. (eds.) ISC 2017. LNCS, vol. 10266, pp. 413–430. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58667-0_22
36. Rush, D., Martin, S.J., Kappel, M., Sandstedt, M., Williams, J.: Cray XC40 power monitoring and control for knights landing. In: Cray User Group CUG, May 2017. https://cug.org/proceedings/cug2016_proceedings/includes/files/pap112s2-file1.pdf
37. Saini, S., Jin, H., Hood, R., Barker, D., Mehrotra, P., Biswas, R.: The impact of hyper-threading on processor resource utilization in production applications. In: Proceedings of the 2011 18th International Conference on High Performance Computing, pp. 1–10, HIPC 2011, IEEE Computer Society, Washington, DC, USA (2011). https://doi.org/10.1109/HiPC.2011.6152743

38. Sodani, A.: Knights landing (KNL): 2nd generation Intel Xeon Phi Processor. In: Hot Chips 27, Flint Center, Cupertino, CA, August 23–25 2015. http://www.hotchips.org/wp-content/uploads/hc_archives/hc27/HC27.25-Tuesday-Epub/HC27.25.70-Processors-Epub/HC27.25.710-Knights-Landing-Sodani-Intel.pdf
39. ANL Theta Supercomputer. https://www.alcf.anl.gov/theta
40. Wang, B., Ethier, S., Tang, W.M., Ibrahim, K.Z., Madduri, K., Williams, S., Oliker, L.: Modern Gyrokinetic Particle-In-Cell Simulation of Fusion Plasmas on Top Supercomputers. CoRR abs/1510.05546 (2015). http://arxiv.org/abs/1510.05546
41. Zhao, Z., Wright, N.J., Antypas, K.: Effects of hyper-threading on the NERSC workload on Edison. In: Cray User Group CUG, May 2013. https://www.nersc.gov/assets/CUG13HTpaper.pdf

# A Survey of Application Memory Usage on a National Supercomputer: An Analysis of Memory Requirements on ARCHER

Andy Turner[1(✉)] and Simon McIntosh-Smith[2]

[1] EPCC, University of Edinburgh, Edinburgh EH9 3JZ, UK
a.turner@epcc.ed.ac.uk
[2] Department of Computer Science, University of Bristol, Bristol BS8 1UB, UK
S.McIntosh-Smith@bristol.ac.uk

**Abstract.** In this short paper we set out to provide a set of modern data on the actual memory per core and memory per node requirements of the most heavily used applications on a contemporary, national-scale supercomputer. This report is based on data from all jobs run on the UK national supercomputing service, ARCHER, a 118,000 core Cray XC30, in the 1 year period from 1st July 2016 to 30th June 2017 inclusive. Our analysis shows that 80% of all usage on ARCHER has a maximum memory use of 1 GiB/core or less (24 GiB/node or less) and that there is a trend to larger memory use as job size increases. Analysis of memory use by software application type reveals differences in memory use between periodic electronic structure, atomistic N-body, grid-based climate modelling, and grid-based CFD applications. We present an analysis of these differences, and suggest further analysis and work in this area. Finally, we discuss the implications of these results for the design of future HPC systems, in particular the applicability of high bandwidth memory type technologies.

**Keywords:** HPC · Memory · Profiling

## 1 Introduction

Memory hierarchies in supercomputer systems are becoming increasingly complex and diverse. A recent trend has been to add a new kind of high-performance memory but with limited capacity, to high-end HPC-optimised processors. Recent examples include the MCDRAM of Intel's Knights Landing Xeon Phi, and the HBM of NVIDIA's Pascal P100 GPUs. These memories tend to provide 500–600 GBytes/s of STREAM bandwidth, but to only about 16 GiB of capacity per compute node.

To establish whether these fast but limited capacity memories are applicable to mainstream HPC services, we need to revisit and update our data on the typical memory requirements of modern codes. This is an area where conventional

© Springer International Publishing AG 2018
S. Jarvis et al. (Eds.): PMBS 2017, LNCS 10724, pp. 250–260, 2018.
https://doi.org/10.1007/978-3-319-72971-8_13

wisdom abounds, yet it is likely to be out of date. The underpinnings of this conventional wisdom were recently reviewed by Zivanovic *et al.* [1]. One of the key findings from this previous study is that the amount of memory provisioned on large HPC systems is a consequence of a desired high performance for HPL, where larger memory is required to achieve good scores, rather than the actual memory requirements of real HPC applications.

There are many factors which affect the memory capacity requirements of any scientific code, and these factors are likely to have been changing rapidly in recent years. For example, the ratio of network performance to node-level performance tends to influence how much work each node needs to perform, and as the node-level performance tends to grow faster than the network-level performance, the trend is for each node to be given more work, typically implying larger memory requirements. Because of these changes, we cannot rely on conventional wisdom, nor even older results, when estimating future memory capacity requirements. Instead, we need up-to-date, good quality data with which to reason and then to inform our predictions.

In this study we have used ARCHER, the UK's national supercomputer, as an example of a reasonably high-end supercomputer. ARCHER reached #19 in the Top500 upon its launch in 2013. It is a 4,920 node Cray XC30, and consists of over 118,000 Intel Ivy Bridge cores, with two 2.7 GHz, 12-core E5-2697 v2 CPUs per node[1]. 4,544 of the 4,920 nodes have 64 GiB per node (2.66 GiB per core), while the remaining 376 'high memory' nodes have 128 GiB each (5.32 GiB per core).

We set out to analyse all of the codes running on ARCHER for their current memory usage, in the hope that this will inform whether future processors exploiting smaller but faster HBM-like memory technologies would be relevant to ARCHER-class national services. Zivanovic *et al.* [1] also studied the memory footprints of real HPC applications on a system of similar scale to ARCHER. Their approach differs from ours in that they used profiling tools to instrument a particular subset of applications using a standard benchmark set (PRACE UEABS [2]). In contrast, we are sampling the memory usage of *every* job run on ARCHER in the analysis period. Thus our data should complement that from Zivanovic's study.

## 2 Data Collection and Analysis

We use Cray Resource Usage Reporting (RUR) [3] to collect various statistics from all jobs running on ARCHER. This includes the maximum process memory used across all parallel processes in a single job. It is this data that provides the basis of the analysis in this paper. Unfortunately, RUR does not include details on the number of processes per node, executable name, user ID and project ID which allow the memory use to be analysed in terms of application used and research area (for example). Tying the RUR data to these additional properties of jobs on ARCHER requires importing multiple data feeds into our

[1] https://www.archer.ac.uk/about-archer/hardware/.

service management and reporting database framework, SAFE [4]. All of the data reported in this paper rely on multiple data feeds linked together through SAFE. Applications are identified using a library of regexp against executable name that has been built up over the ARCHER service with help from the user community. With this approach we are currently able to identify around 75% of all usage on ARCHER.

Memory usage numbers below are presented as maximum memory use in GiB/node. As there are 24 cores per node on ARCHER, a maximum memory use of 24 GiB/node corresponds (if memory use is homogeneous) to 1 GiB/core. Note that, as described above, the actual value measured on the system is maximum memory use across all parallel processes running in a single job. The measured value has then been converted to GiB/node by multiplying by the number of processes used per node in the job. This is a reasonable initial model as the majority of parallel applications on ARCHER employ a symmetric parallel model, where the amount of memory used per process is similar across all processes. However, if an application has asymmetric memory use across different parallel processes, this will show up as an overestimation of the maximum memory use per node. Indeed, we discuss an example of exactly this effect in the section on grid-based climate modelling applications below.

We have analysed memory usage data from Cray RUR for all applications run on ARCHER in the 1 year period from $1^{st}$ July 2016 to $30^{th}$ June 2017 inclusive.

## 3 Application Memory Usage

First we look at overall memory usage for all jobs on ARCHER in the period, and then go on to look at the data for the top 10 applications used on the service (these 10 applications cover over 50% of the usage). We have broken the applications down into four broad types to facilitate this initial analysis:

- Periodic electronic structure: VASP, CASTEP, CP2K
- N-body models: GROMACS, LAMMPS, NAMD
- Grid-based climate modelling: Met Office UM, MITgcm
- Grid-based computational fluid dynamics: SBLI, OpenFOAM

Due to space restrictions we are not able to include memory usage figures for all applications listed above. Instead we plot the data that best represents the trends for that application class, or that we use to illustrate a particular point. An expanded version of this paper that includes plots for all the applications listed above (along with the numerical data that was used to produce the plots) can be found online [5].

### 3.1 Overall Memory Use

Table 1 shows a breakdown by memory use for all jobs on ARCHER in the 12 month analysis period. Additional columns show the usage for *Small* jobs (32 nodes or less) and *Large* jobs (more than 32 nodes). Just under 80% of all usage in the period uses a maximum of 24 GiB/node (1 GiB/core). Memory usage for larger jobs is generally higher, with large jobs showing only 70% of usage at a maximum of 24 GiB/node, and over 25% of usage in the range [24,96) GiB/node. These results generally echo the results from Zivanovic *et al.* [1] with the exception that we do not observe large memory requirements for smaller jobs, as seen in their application benchmarks. This could be due to the benchmarks chosen in the previous study not being representative of the usage pattern of those applications on ARCHER (see, for example, the results for CP2K below which is also one of the applications in the PRACE UEABS).

**Table 1.** % usage breakdown by maximum memory use per node for all jobs run on ARCHER during the analysis period. (Small: 32 nodes or less; Large: more than 32 nodes.)

| Max. memory use (GiB/node) | Usage | | |
|---|---|---|---|
| | All | Small | Large |
| (0,12) | 61.0% | 69.5% | 53.0% |
| [12,24) | 18.6% | 19.4% | 16.9% |
| [24,48) | 11.5% | 7.7% | 14.8% |
| [48,96) | 6.9% | 3.0% | 11.2% |
| [96,128) | 2.0% | 0.4% | 4.2% |

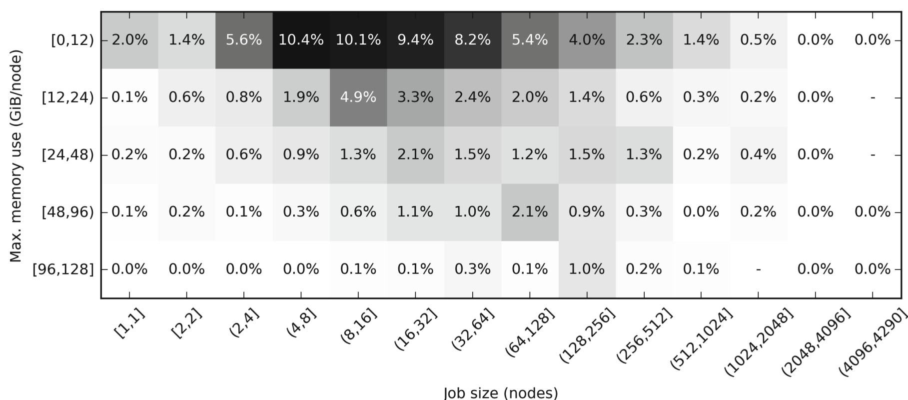



**Fig. 1.** Usage heatmap of maximum memory versus job size for all jobs in the period.

Figure 1 shows a heatmap of the usage broken down by job size versus overall memory use in GiB/node (extrapolated from the maximum process memory usage). The trend for higher maximum memory use as job size increases can be seen as a diagonal feature running from top left to bottom right.

### 3.2 Periodic Electronic Structure (PES) Applications

The top three of the top ten most heavily used applications on ARCHER are PES modelling applications: VASP, CASTEP, and CP2K. Although the implementation of the theory differs across the three applications, the algorithms used are similar, involving dense linear algebra and spectral methods (generally small Fourier transforms). Table 2 shows the breakdown of usage by maximum memory use for these three applications combined.

**Table 2.** % usage breakdown by maximum memory use per node for VASP, CASTEP and CP2K jobs run on ARCHER during the analysis period. (Small: 32 nodes or less; Large: more than 32 nodes.)

| Max. memory use (GiB/node) | Usage | | |
|---|---|---|---|
| | All | Small | Large |
| (0,12) | 65.4% | 68.6% | 55.4% |
| [12,24) | 21.4% | 20.0% | 25.7% |
| [24,48) | 9.4% | 8.5% | 12.1% |
| [48,96) | 3.7% | 2.7% | 6.7% |
| [96,128) | 0.1% | 0.1% | 0.1% |

Comparing to the overall distribution (Table 1), we can see that this distribution is very similar, with a large majority of usage at 24 GiB/node (1 GiB/core) or less. This is unsurprising, as PES applications make up such a large part of the use of ARCHER (almost 30% from just these three applications, and over 40% if all similar applications are included). Only 13% of usage needs more than 24 GiB/node, and this only increases to 19% for larger jobs. The heatmap of usage broken down by maximum memory use and job size for CP2K is shown in Fig. 2. Heatmaps for VASP and CASTEP show the same trends as that for CP2K. When compared to the overall heatmap (Fig. 1) CP2K does not mirror the trend that larger job sizes lead to increased memory use per node. For PES applications, the larger jobs have similar memory use per node as smaller jobs.

It is interesting to compare our results for CP2K (Fig. 2) with those reported in Zivanovic *et al.* [1]. In particular, they report that the small CP2K benchmark (Test Case A: bulk water) has a memory requirement of approx. 6 GiB/core running on a single node (16 cores), whereas on ARCHER, small CP2K jobs generally have maximum memory requirements of less than 0.5 GiB/core. This would suggest that, generally, the size of problem people are using these low core-count jobs to study on ARCHER is substantially smaller than the small CP2K benchmark in the PRACE UEABS.

### 3.3 *N*-body atomistic simulation applications

The *N*-body atomistic modelling applications, GROMACS, LAMMPS, and NAMD, are important applications in the top ten on ARCHER. Two of these, GROMACS and NAMD, are almost exclusively used for biomolecular simulations, while LAMMPS is used more broadly across a number of research areas. All three applications use very similar algorithms, with pairwise evaluation of short-range forces and energies, and Fourier transforms for long range electrostatic forces. The parallelisation strategies differ across the applications. Table 3 shows the breakdown of usage by maximum memory use for these three applications combined.

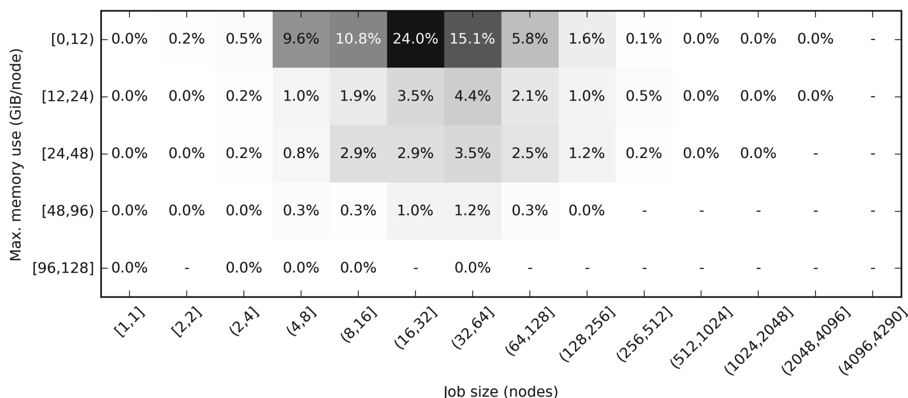



**Fig. 2.** Usage heatmap of maximum memory versus job size for CP2K jobs in the period.

**Table 3.** % usage breakdown by maximum memory use per node for GROMACS, LAMMPS and NAMD jobs run on ARCHER during the analysis period. (Small: 32 nodes or less; Large: more than 32 nodes.)

| Max. memory use (GiB/node) | Usage | | |
|---|---|---|---|
| | All | Small | Large |
| (0,12) | 91.6% | 96.6% | 80.7% |
| [12,24) | 2.7% | 3.1% | 2.1% |
| [24,48) | 0.5% | 0.4% | 0.6% |
| [48,96) | 4.8% | 0.0% | 15.5% |
| [96,128) | 0.1% | 0.0% | 0.1% |

These applications generally have the lowest memory demands on ARCHER, with over 90% of usage requiring less than 12 GiB/node (0.5 GiB/core). Even for larger jobs, only 20% of jobs require more than 12 GiB/node. This ties in with

the results from NAMD and GROMACS in Zivanovic *et al.* [1]. Figure 3 shows the heatmap of memory usage versus job size for NAMD. Heatmaps for the other applications show similar trends. Each of these applications have a class of large calculations that have higher memory demands (48–96 GiB/node, around four times higher than the majority of jobs). This is particularly prominent in the NAMD heatmap (Fig. 3). We plan to contact users to understand what this use case is and why it has such a large memory requirement. It is worth noting that these jobs with a larger memory requirement only represent 0.5% of the total node hours used on ARCHER in the period.

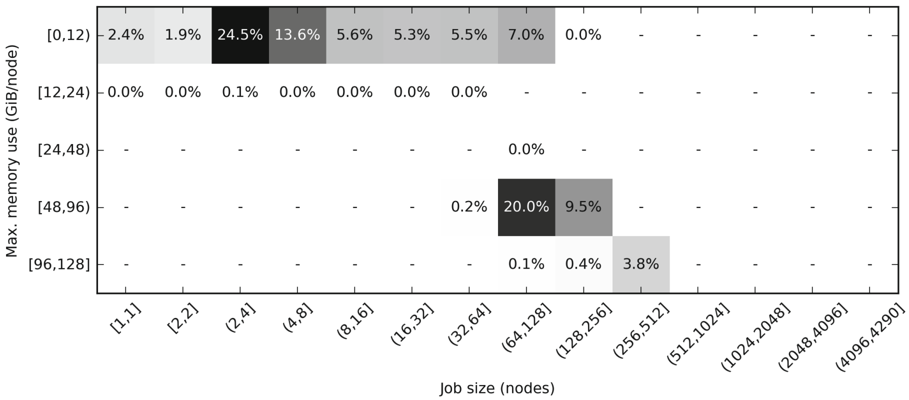



**Fig. 3.** Usage heatmap of maximum memory versus job size for NAMD jobs in the period.

### 3.4 Grid-Based Climate Modelling Applications

Both of the grid-based climate modelling applications analysed (Met Office UM and MITgcm) show a very different profile from the other application classes studied in this paper. As shown in Table 4, a much higher proportion of jobs use large amounts of memory, and that use of higher memory is almost always for the largest jobs. The heatmap for the MET Office UM (Fig. 4) clearly reveals a very distinct split, with two classes of job existing: small jobs (less than 32 nodes) with low memory requirements (24 GiB/node or less), and very large jobs (above 128 nodes) with very large memory requirements (96–128 GiB/node for Met Office UM). MITgcm shows a similar usage peak for large jobs at 24–48 GiB/node for 256–512 node jobs. We have performed initial investigations into this phenomenon for the Met Office UM jobs and found that it is due to asymmetrical memory use across parallel processes in the jobs. These jobs have a small number of parallel processes that have much higher memory requirements. These high-memory processes work as asynchronous I/O servers that write data to the file system while other processes continue the computational work.

### 3.5 Grid-Based Computational Fluid Dynamics (CFD) Applications

Finally, we look at the grid-based CFD applications. Two applications appear in the top ten on ARCHER: SBLI and OpenFOAM. Table 5 reveals that they do not follow the same memory usage trend as the climate modelling applications, even though both classes of application use grid-based methods and both have the same drive to higher resolution and, generally, larger jobs. The usage heatmap for SBLI (Fig. 5) shows that the large jobs can have a larger memory requirement (24–96 GiB/node), but this is not always required (as was seen for the climate applications), as a reasonable proportion of the large jobs also have low memory requirement (up to 12 GiB/node). We plan to contact SBLI users to understand the differences between the jobs that have large memory requirements and those having low memory requirements. The OpenFOAM data show no clear link between increasing job size and increased memory requirements, with 94% of usage requiring less than 24 GiB/node.

**Table 4.** % usage breakdown by maximum memory use per node for Met Office UM and MITgcm jobs run on ARCHER during the analysis period. (Small: 32 nodes or less; Large: more than 32 nodes.)

| Max. memory use (GiB/node) | Usage | | |
|---|---|---|---|
| | All | Small | Large |
| (0,12) | 53.5% | 66.4% | 18.8% |
| [12,24) | 25.0% | 33.1% | 3.3% |
| [24,48) | 6.0% | 0.2% | 21.5% |
| [48,96) | 0.2% | 0.2% | 0.0% |
| [96,128) | 15.3% | 0.0% | 56.4% |

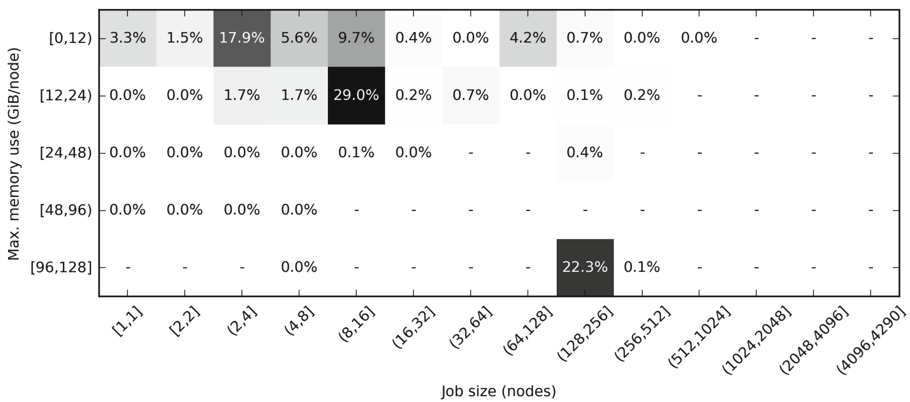



**Fig. 4.** Usage heatmap of maximum memory versus job size for Met Office UM jobs in the period.

**Table 5.** % usage breakdown by maximum memory use per node for SBLI and OpenFOAM jobs run on ARCHER during the analysis period. (Small: 32 nodes or less; Large: more than 32 nodes.)

| Max. memory use (GiB/node) | Usage | | |
|---|---|---|---|
| | All | Small | Large |
| (0,12) | 64.2% | 71.8% | 62.2% |
| [12,24) | 14.8% | 8.6% | 16.4% |
| [24,48) | 13.4% | 5.6% | 15.4% |
| [48,96) | 7.7% | 14.0% | 6.0% |
| [96,128) | 0.0% | 0.0% | 0.0% |

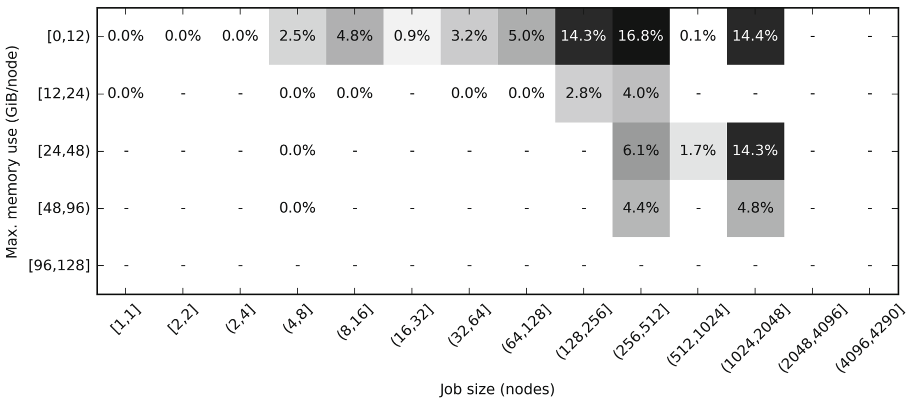



**Fig. 5.** Usage heatmap of maximum memory versus job size for SBLI jobs in the period.

## 4 Conclusions and Future Work

Our initial analysis of memory use of applications running on ARCHER has shown that a large amount of use (80%) is under 24 GiB/node (1 GiB/core), with a significant fraction (60%) using less than 12 GiB/node (0.5 GiB/core). There seems to be a trend to increased memory requirements as jobs get larger, although some of this increase may be due to asymmetric memory use across processes. Another possible reason for this phenomenon is that larger jobs are usually larger simulations, and so the memory requirement may generally be larger. These results are generally in line with those reported for specific applications benchmarks in Zivanovic *et al.* [1], with the exception that we do not see large memory requirements for small jobs as reported in their study.

We also illustrated one weakness in our current analysis, when memory use between parallel processes is very asymmetric. As the analysis is based on maximum process memory use extrapolated to a per-node value, parallel processes with very different memory use within the same application can produce

misleading estimated memory use figures. We plan to refine our analysis methodology to take this type of asymmetric memory use into account for future work.

Our analysis leads us to conclude that there is an opportunity for exploiting emerging, high bandwidth memory technologies for most of the research on ARCHER. Many applications from a broad range of research areas have performance that is currently bound by memory bandwidth and would therefore potentially see significant performance improvements from this type of technology. The data in this paper suggests that, even memory was as low as 0.5 GiB/core, two-thirds of the current workload on ARCHER would be in a position to exploit this, without any software changes. Expanding to 1.0 GiB/core would address nearly 80% of ARCHER's current workload. Our results (and results from previous studies) suggest that a future ARCHER service could even benefit from architectures where HBM-like technologies with limited capacity *replace* main memory, rather than using a hybrid solution (such as the MCDRAM+DRAM seen on the Intel Xeon Phi). The reasoning here is that using HBM technologies as a main memory replacement allows applications to access the best performance *without application code modifications* whereas in the hybrid approach the only way to use the HBM without code modification is as an additional, large cache level, which can limit the performance gains available [6]. Another option would be to use a combination of processors with high memory bandwidth alongside processors with high memory capacity.

In addition to refining our analysis technique using this new data from ARCHER, we need to work with the user community to understand the different memory use classes for particular applications and research problems. This work will help us make sure that future UK national supercomputing services provide the best resource for researchers.

In future we plan to work with other HPC centres worldwide to understand the variability in memory use profile across different services. We have already opened discussions with other European and US HPC centres on this topic.

## References

1. Zivanovic, D., Pavlovic, M., Radulovic, M., Shin, H., Son, J., Mckee, S.A., Carpenter, P.M., Radojković, P., Ayguadé, E.: Main memory in HPC: Do we need more or could we live with less? ACM Trans. Archit. Code Optim. **14**(1), March 2017. Article 3, https://doi.org/10.1145/3023362
2. PRACE. Unified European Applications Benchmark Suite (2013). http://www.prace-ri.eu/ueabs/. Accessed 21 Sep 2017
3. XC30 Series System Administration Guide (CLE 6.0.UP01) S-2393. https://pubs.cray.com/content/S-2393/CLE%206.0.UP01/xctm-series-system-administration-guide-cle-60up01/resource-utilization-reporting. Accessed 21 Sep 2017
4. Booth, S.: Analysis and reporting of Cray service data using the SAFE. In: Cray User Group 2014 Proceedings. https://cug.org/proceedings/cug2014_proceedings/includes/files/pap135.pdf. Accessed 21 Sep 2017

5. ARCHER. Memory usage on the UK national supercomputer, ARCHER: analysis of all jobs and leading applications (2017). http://www.archer.ac.uk/documentation/white-papers/. Accessed 21 Sep 2017
6. Radulovic, M., Zivanovic, D., Ruiz, D., de Supinski, B.R., McKee, S.A., Radojković, P., Ayguadé, E.: Another trip to the wall: How Much will stacked DRAM benefit HPC? In: Proceedings of the 2015 International Symposium on Memory Systems (MEMSYS 2015), pp. 31–36. ACM, New York, NY, USA (2015). https://doi.org/10.1145/2818950.2818955

# Author Index

Allen, Tyler 236
Aupy, Guillaume 44
Austin, Brian 236

Badawy, Abdel-Hameed A. 114
Balogh, G. D. 22
Bird, Robert 114

Cavelan, Aurélien 158
Chabbi, Milind 221
Chennupati, Gopinath 114
Chien, Andrew A. 158

Daley, Christopher S. 236
DeLeon, Robert L. 197
Denoyelle, Nicolas 91
Doerfler, Douglas 236

Eidenbenz, Stephan 114

Faizian, Peyman 136
Fang, Aiman 158
Fatica, Massimiliano 67
Fèvre, Valentin Le 44
Furlani, Thomas R. 197

Gainaru, Ana 44
Gallo, Steven M. 197
Gawande, Nitin A. 3
Giannozzi, Paolo 67
Goglin, Brice 91

Haftka, Raphael T. 179
Hoisie, Adolfy 3

Ilic, Aleksandar 91
Innus, Martins D. 197

Jeannot, Emmanuel 91
Jones, Matthew D. 197

Kim, Nam H. 179
Kumar, Nalini 179

Lam, Herman 179
Lang, Michael 136

McIntosh-Smith, Simon 250
Misra, Satyajayant 114
Mollah, Md Atiqul 136
Mudalige, G. R. 22

Neelakantan, Aravind 179

Pakin, Scott 136
Park, Chanyoung 179
Patra, Abani K. 197
Phillips, Everett 67

Rahman, Md Shafayat 136
Reguly, I. Z. 22
Robert, Yves 158
Romero, Joshua 67
Ruetsch, Gregory 67

Santhi, Nandakishore 114
Siegel, Charles 3
Simakov, Nikolay A. 197
Sousa, Leonel 91
Spiga, Filippo 67

Tallent, Nathan R. 3
Thulasidasan, Sunil 114
Turner, Andy 250

Vishnu, Abhinav 3

White, Joseph P. 197
Wright, Nicholas J. 236

Yoga, Adarsh 221
Yuan, Xin 136

Zhang, Yiming 179