High Performance Computing Symposium (HPC 2018)

2018 Spring Simulation Multiconference (SpringSim-HPC'18)

Simulation Series Volume 50 Number 4

Baltimore, MD, USA
15-18, April 2018

Editors:

Layne T. Watson
Masha Sosonkina
Karl Rupp
William I. Thacker
Josef Weinbub

ISBN: 978-1-5108-6016-2

WELCOME

Welcome to SpringSim'18

Welcome from the SpringSim'18 Conference Chairs

On behalf of the Organizing Committee, it is our pleasure to welcome you to the 2018 Spring Simulation Multi-conference in Baltimore, Maryland. Baltimore is the second-largest seaport in the Mid-Atlantic and home to numerous Universities and Colleges. The conference is organized by the Society for Modeling and Simulation International (SCS), the World's first-born international M&S society, which, from its foundation in 1952, has effectively engaged our community and continues to play a significant role in advancing research and its contribution to practice. SpringSim'18 covers state-of-the-art developments in M&S methodology, technology, performances and application in disciplines as diverse as applied computing, industrial systems, communications and networking, medicine and service systems.

We have an exciting program to offer our attendees this year. This includes presentations of peer-reviewed original research papers, work in progress, student demo and posters, keynote speeches, featured speeches, and tutorials delivered by experts. This year's conference consists of the following eight symposia:

Agent-Directed Simulation Symposium (Chaired by Yu Zhang and Gregory Madey), Annual Simulation Symposium (Erika Frydenlund, Shafagh Shafer, and Hamdi Kavak), Communications and Networking Symposium (Abdolreza Abhari and Hala ElAarag), High Performance Computing Symposium (Layne Watson, Masha Sosonkina, Will Thacker and Josef Weinbub), Symposium on Modeling and Simulation in Medicine (Jerzy Rozenblit and Johannes Sametinger), Symposium on Theory of Modeling and Simulation (Fernando Barros and Xiaolin Hu), Model-driven Approaches for Simulation Engineering confirming its success for its 2nd year (Andrea D'Ambrogio and Umut Durak) and finally M&S and Complexity in Intelligent, Adaptive and Autonomous Systems Symposium appearing for its 3rd year (Saurabh Mittal, Jose L. Risco Martin, and Marco Luetzenberger)

We would like to thank the organizers of the symposia, their respective technical program committees and reviewers for their effort in putting together the program. As a Multi-conference, our success depends entirely on their contribution.

We have an exciting line-up of distinguished keynote speakers this year again; we would like to express our gratitude to Dr. Alan Liu and Dr. Jerzy Rozenblit for accepting our invitation to deliver keynote speeches. Dr. Alan Liu is the Director of the Virtual Medical Environments Laboratory at the Val G. Hemming Simulation Center, Uniformed Services University of the Health Sciences. His talk will highlight the design, construction, and use of the Wide Area Virtual Environment, a unique training resource capable of simulating a very wide range of medical scenarios for both military and civilian applications. Dr. Jerzy Rozenblit is a Distinguished Professor and a Raymond J. Oglethorpe Endowed Chair in the Electrical and Computer Engineering at The University of Arizona. His talk will highlight Enhancing Patients' Safety through Simulation Modeling, High Technology, and Human Skills.

Welcome to SpringSim’18

This year we are continuing two initiatives launched in recent years: Featured Speakers and Student M&S Demo Session. The Featured Speakers series spotlights authors of invited papers in selected symposia. Featured Speaker is designed to emphasize state-of-the-art contributions in chosen fields, as considered by the symposium Chairs. This year our Featured Speakers are Xudong Zhang, Lin Zhang and Chungman Seo. We thank our Featured Speakers for exposing new topics in Modeling and Simulation. The Demo Session is in its second year and led by Salim Chemlal and Mohammad Moallemi.

It encourages students to display their running simulations that they have authored in contributed papers. This year you will find topics such as M&S interface on a tablet device, to complicated M&S tools running on a distributed platform. Some of this year's demos are within the realm of healthcare, supply chain management, and dense crowd simulation.

We would like to thank our sponsors who have donated funds, software licenses and books and which has made it possible for us to recognize best papers in the conference, support for student travel, and provided an enhanced conference experience for our delegates. We sincerely thank the Virginia Modeling, Analysis & Simulation Center (VMASC) and VMASC Industry Association. Our sincere gratitude goes to our Organization Committee. We would like to thank Jose Padilla and Paul Antoine Bisgambiglia (Proceedings Co-Chairs), Saikou Diallo (Sponsorship Chair), Saurabh Mittal (Awards Chair), Youssef Bouanan and Joachim Denil (Publicity Co-Chairs), Deniz Cetinkaya, and Marina Zapater (Tutorial Co-Chairs), Hamdi Kavak and Omer F. Keskin (WIP Co-Chairs) and Salim Chemlal and Mohammad Moallemi (Student M&S Demo Session Co-Chairs).

We would also like to thank SCS Executive Director, Oletha Darensburg for conference coordination activities, Carmen Ramirez for Website and Mike Chinni for his help with the proceedings and digital libraries.

Finally, as social events, attendees will tour the John Hopkins Simulation Center in order to appreciate the simulation activities in the Baltimore neighbourhood as well as a Cruise on the Bay by Watermark to learn about historical Baltimore, Maryland and enjoy the heart of the city of Baltimore.

Thank you for making SprimSim’18 a success through your participation. We look forward to your continued participation in SpringSim’19.

Gregory Zacharewicz
General Chair
University of Bordeaux, France

[illegible]

Andrea D’Ambrogio
Vice-General Chair
University of Rome Tor Vergata, Italy

[illegible]

Umut Durak
Program Chair
German Aerospace Center (DLR)
Germany

[illegible]

HPC'18 General Chair's Message

Welcome to the 26th High Performance Computing Symposium (HPC 2018), one of the longest running symposia within SpringSim. This year we have 13 high quality papers, covering algorithms, performance modeling of high end computing systems, hybrid parallel programming, visualization, quantum chemistry, scheduling, parallel I/O, resilience, and dynamical systems. We thank you for your participation in HPC 2018, and hope that you enjoy your stay in Baltimore, perhaps finding time to attend some talks from the other symposia.

Layne T. Watson

Layne T. Watson, General Chair

CDBB: AN NVRAM-BASED BURST BUFFER COORDINATION SYSTEM FOR PARALLEL FILE SYSTEMS

Ziqi Fan

Computer Science & Engineering
University of Minnesota, Twin Cities
Minneapolis, MN 55455
fanxx234@umn.edu

Fenggang Wu

Computer Science & Engineering
University of Minnesota, Twin Cities
Minneapolis, MN 55455
wuxx0835@umn.edu

Jim Diehl

Computer Science & Engineering
University of Minnesota, Twin Cities
Minneapolis, MN 55455
jdiehl@umn.edu

David H.C. Du

Computer Science & Engineering
University of Minnesota, Twin Cities
Minneapolis, MN 55455
du@umn.edu

Doug Voigt

Storage Division
Hewlett Packard Enterprise
Boise, ID 83714
doug.voigt@hpe.com

ABSTRACT

For modern HPC systems, failures are treated as the norm instead of exceptions. To avoid rerunning applications from scratch, checkpoint/restart techniques are employed to periodically checkpoint intermediate data to parallel file systems. To increase HPC checkpointing speed, distributed burst buffers (DBB) have been proposed to use node-local NVRAM to absorb the bursty checkpoint data. However, without proper coordination, DBB is prone to suffer from low resource utilization. To solve this problem, we propose an NVRAM-based burst buffer coordination system, named collaborative distributed burst buffer (CDBB). CDBB coordinates all the available burst buffers, based on their priorities and states, to help overburdened burst buffers and maximize resource utilization. We built a proof-of-concept prototype and tested CDBB at the Minnesota Supercomputing Institute. Compared with a traditional DBB system, CDBB can speed up checkpointing by up to 8.4x under medium and heavy workloads and only introduces negligible overhead.

Keywords: burst buffer, non-volatile memory, parallel file system, coordination system

1 INTRODUCTION

For high-performance computing (HPC), with its countless hardware components and complex software stacks, failures become the norm rather than exceptions. For a supercomputer with tens of thousands of nodes, the mean time between failures (MTBF) can be in the order of hours (Bautista-Gomez et al. 2011). However, many scientific HPC applications (e.g., simulations, modeling, and mathematical calculations) need to run days or even months before completion. As one of the most popular fault tolerance techniques, checkpoint/restart (C/R) is used to periodically store the intermediate application states (usually some files) into parallel file systems (PFSs). Then, if failures do happen, the application can be restarted by reading back those intermediate files and resuming from there instead of starting from scratch.

Many studies show that checkpoint (CKPT) activity is the dominating contributer to total HPC I/O traffic and application run time (Rajachandrasekar et al. 2013, Wang et al. 2016). Even though PFSs are designed to provide high aggregate I/O throughput, the large amount of bursty writes generated during checkpointing means that PFSs alone are not sufficient and become the bottleneck of the whole HPC system.

To improve checkpointing speed, an intermediate layer, called a burst buffer (BB), is often used to alleviate the burden on PFSs. BBs consist of fast storage media and/or dedicated software and network stacks that can absorb checkpoint data orders of magnitude faster than PFSs. Then the buffered data will be drained to PFSs in the background if necessary. Traditional burst buffers mostly consist of solid state drives, but newly developed NVRAM technologies (e.g., 3D Xpoint, PCM, and NVDIMM) are better candidates due to their better performance (Fan et al. 2015). In this paper, we will focus on these emerging NVRAM-based BBs.

There are two types of burst buffer architectures: centralized BB or distributed BB. In a centralized BB architecture, a big BB appliance or multiple BB appliances will absorb checkpoint data from all the compute nodes (Liu et al. 2012, Wang et al. 2014, Thapaliya et al. 2016). The checkpoint data must be transmitted through a network to reach the centralized BB. On the contrary, in the more popular distributed BB architecture, each BB is smaller capacity and put closer, or even attached directly, to each compute node (Wang et al. 2016, Rajachandrasekar et al. 2013, Wang et al. 2015). Under the distributed BB architecture, the absorption of checkpoint data is much quicker than using networks since BBs are closer to the data origin. It is also more scalable and flexible to add/remove distributed BBs to/from compute nodes as needed. However, the downside of the distributed BB architecture is potentially low BB resource utilization; without proper scheduling and coordination, some BBs are overburdened while others might be idle.

By observing HPC application execution patterns and experimenting on the *Itasca* HPC cluster (described in Section 4.2), we find there are opportunities to optimize the distributed BB architecture to improve BB resource utilization. Here is a summary of our observations: 1) Multiple HPC applications are running concurrently instead of few; 2) Compute nodes running the same application are at the same HPC phase (e.g., reading data, computation, checkpointing); 3) Compute nodes running different applications could be in distinct HPC phases; 4) Some applications (hence their compute nodes) do not perform checkpointing; 5) Write throughput to peer compute nodes (1.83 GB/s) is much higher than write throughput to the PFS (0.52 GB/s).

As mentioned above, while the distributed BB architecture has plenty of advantages it can suffer low resource utilization. This problem is particularly severe for NVRAM-based BBs since NVRAM is much more expensive than other storage media (e.g., SSD), which makes NVRAM much more valuable and scarce (Fan et al. 2014). Based on our observations of HPC application execution patterns and experimentations on HPC systems, we propose a novel BB coordination system, named collaborative distributed burst buffer (CDBB), to improve resource utilization and further increase HPC checkpointing speed. Specifically, we design a BB coordinator to monitor and control all BBs to make them work collaboratively. When an application performs checkpointing, instead of only relying on local BBs, the BB coordinator will globally select available remote BBs (based on their priority and on-the-fly status) in nodes running other applications to contribute and alleviate the burden of those local BBs. We have built a proof-of-concept CDBB prototype and evaluated it on the *Itasca* HPC cluster at the Minnesota Supercomputing Institute. Compared with a traditional distributed burst buffer system using local BBs only, the results show that under a light workload, CDBB only introduces negligible overhead, and under medium and heavy workloads, CDBB can improve CKPT speed by up to 8.4×.

The rest of the paper is organized as follows. In Section 2, we present some background and related work about checkpoint/restart tools, HPC application characteristics, and NVRAM technologies. Section 3 gives a detailed description of our proposed CDBB coordination system followed by evaluations in Section 4. Finally, Section 5 concludes our work.

2 BACKGROUND AND RELATED WORK

2.1 Checkpoint/Restart

There are two types of C/R tools: application-level C/R tools and system-level C/R tools. Application-level C/R tools come with applications themselves; only data needed for restart are stored, so the checkpoint data size could be very small. System-level C/R tools are transparent to applications and usually checkpoint the whole memory space touched by the applications; thus, the checkpoint data size could be much larger. System-level C/R tools are used to checkpoint applications without innate C/R functionalities.

Here, we use a very popular system-level C/R tool, DMTCP (Distributed MultiThreaded CheckPointing) (Ansel et al. 2009), as a reference to explain how C/R tools work. DMTCP is in user space, does not require root privilege, and is independent of system kernel version, which makes it very flexible and user-friendly. DMTCP has a `dmtcp_coordinator` process which must be started before operating `dmtcp_checkpoint` or `dmtcp_restart`. Checkpoints can be performed automatically on an interval, or they can be initiated manually on the command line of the dmtcp_coordinator. Once issued a checkpoint request, the dmtcp_coordinator will inform all the corresponding processes to halt, and each process will generate a checkpoint image individually. At the same time, a script is created for restart purposes.

2.2 HPC Application Characteristics

In a typical HPC cluster with hundreds or thousands of compute nodes, usually there are tens or hundreds of applications running concurrently. We used the `showq` command to show the job queue of the *Mesabi* cluster at the Minnesota Supercomputing Institute and found that 636 active jobs were running. Also, the online real-time job queue report of the *Stampede* supercomputer at the Texas Advanced Computing Center showed 699 active jobs were running.

Figure 1 is a high-level simplified illustration of HPC application execution patterns. As shown in the figure, many applications, which start at different times, are running in the cluster. These applications need to read data (usually from PFSs) and perform computation. Applications with C/R requirements will perform checkpointing with frequencies set by the applications or users. After one checkpointing operation is done, the computation resumes. This pattern repeats until either the application is finished or any failures happen, in which case the applications will restart from the latest checkpointing image.

Figure 1 clearly shows that the execution patterns of compute nodes assigned to the same application are quite similar to each other whereas that of the compute nodes assigned to different applications could be quite distinct. For example, when the compute nodes running *Application A* are performing checkpointing, the compute nodes running *Application B* are doing computation. In addition, some applications do not perform checkpointing at all, so they will continuously do computation until the end (Figure 1 *Application*

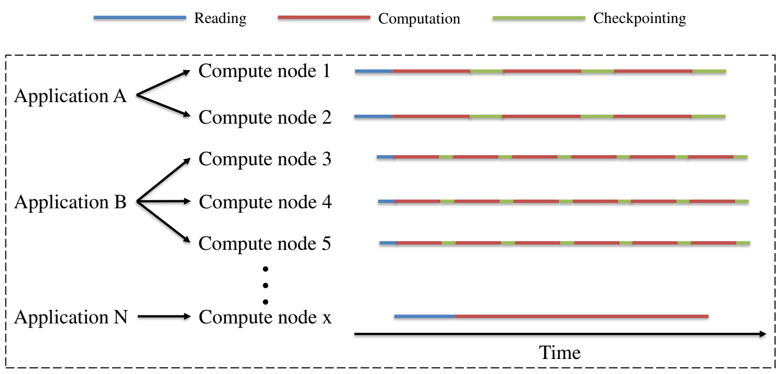


Figure 1: An example of HPC application execution patterns

N). These insights give CDBB opportunities to perform optimization on BB utilization. If there is only one application running in the whole cluster or all the applications in the cluster happen to have the exact same execution patterns, then CDBB would not contribute too much since all the BBs are either being used or idle at the same time.

2.3 Non-volatile Memory

Current memory technologies such as DRAM and SRAM face technological limitations to continued improvement (Coughlin and Grochowski 2015). As a result, there are intense efforts to develop new DRAM-alternative memory technologies. Most of these new technologies are non-volatile memories, because non-volatility can provide additional advantages such as new power saving modes for quick wakeup as well as faster power-off recovery and restart for HPC applications (Coughlin and Grochowski 2015). These new technologies include PCM, STT-RAM, MRAM, RRAM, and 3D XPoint.

Phase Change Memory (PCM) is one of the most promising new NVRAM technologies and can provide higher scalability and storage density than DRAM (Raoux et al. 2014). In general, PCM still has a 5–10× longer latency than DRAM. To overcome PCM's speed deficiency, various system architectures have been designed to integrate PCM into current systems without performance degradation (Lee et al. 2009, Qureshi et al. 2009, Fan et al. 2011). Magnetic RAM (MRAM) and Spin-Torque Transfer RAM (STT-RAM) are expected to replace SRAM and DRAM within the next few years (Gallagher and Parkin 2006, Kawahara et al. 2012, Wu et al. 2017). STT-RAM reduces the transistor count and, consequently, provides a low-cost, high-density solution. Many enterprise and personal devices use MRAM for an embedded cache memory. Resistive RAM (RRAM) is considered a potential candidate to replace NAND Flash memory (Waser et al. 2009, Wu et al. 2016). SanDisk and Hewlett Packard Enterprise are actively developing next generation RRAM technology. Micron and Intel recently introduced 3D XPoint non-volatile memory technology that is presently considered another DRAM alternative (Micron 2015). 3D Xpoint technology has high endurance, high density, and promising performance that is much better than NAND Flash but slightly slower than DRAM. Thus, it is expected to target high-performance in-memory processing (Intel 2015).

3 OUR PROPOSED APPROACH: CDBB

Collaborative distributed burst buffer (CDBB) is a coordination system to maximize the utilization of all available burst buffers and increase checkpointing speed. We will use some concepts in DMTCP (introduced in Section 2.1) as assistance to describe our design, but CDBB is designed as a general framework that does not have any dependencies on the particular implementation or design of any C/R tools.

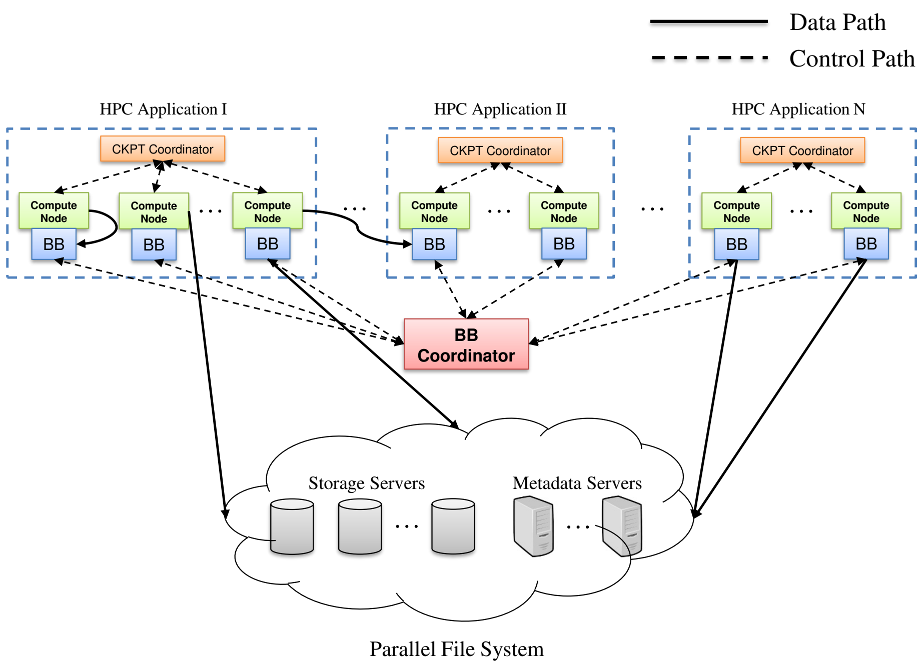


Figure 2: An overview of the CDBB coordination system

3.1 CDBB Overview

Figure 2 is an overview of our proposed CDBB coordination system. It depicts a typical HPC cluster with hundreds or thousands of compute nodes running various types of HPC applications. CKPT coordinators (e.g., dmtcp_coordinator) will control compute nodes running applications with C/R needs. Each compute node is equipped with a small NVRAM-based BB. All the BBs are communicating with, and coordinated by, a BB coordinator. CKPT data are either written to BBs and drained to PFSs in the background or written to PFSs directly. The PFS usually consists of multiple storage servers controlled by one or more metadata servers.

To illustrate the checkpointing workflow of CDBB, we simplify the whole system into Figure 3. As shown in the figure, there is one BB coordinator monitoring and coordinating all the BBs. Each BB (one per compute node) will absorb CKPT data generated by CKPT writers. A CKPT writer represents a CKPT process which generates CKPT data periodically and writes to either a BB or the PFS. Before a CKPT writer sends the real CKPT data (e.g., the CKPT writer in Figure 3 Node 1), it will first negotiate with the BB coordinator (Figure 3 Path 1) to determine the best place to write. There are three possible places: the local BB (through Figure 3 path 2.a), a remote BB (through Figure 3 path 2.b), and the PFS (through Figure 3 path 2.c). Note that the local BB refers to the BB located in the same compute node as the CKPT writer. BBs will drain their buffered data to the PFS in the background (Figure 3 path 3) and report their latest status to the BB coordinator (Figure 3 path 4). Details about the BB coordinator, BBs, and CKPT writers will be presented in the following sections.

3.2 BB Coordinator

The BB coordinator is the brain behind CDBB. It coordinates every individual BB to make globally optimized decisions about where the CKPT data should go. A process flowchart of the BB coordinator is shown in Figure 4.

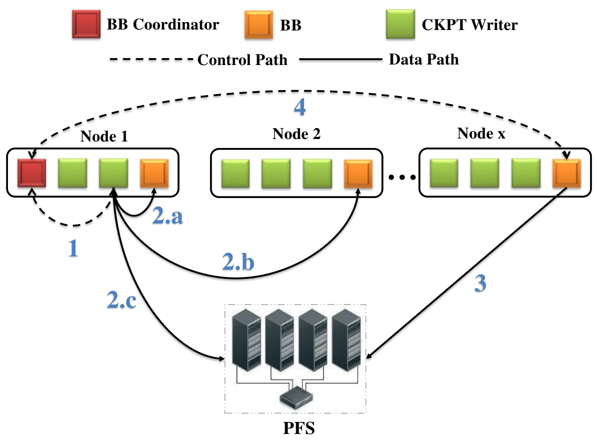


Figure 3: A high-level illustration of CDBB checkpointing workflow

A work cycle of the BB coordinator starts with the arrival of control data. The control data could be sent from either a BB or a CKPT writer. If it is from a BB, the BB coordinator will update its *StatusStore* with the latest status of the sending BB. The *StatusStore* stores the status of all BBs. For the current design, only the space utilization is stored, since it is the only metric the BB coordinator uses to make decisions. As our future work, more metrics such as compute nodes' CPU utilization and data sharing information between processes will be added into the *StatusStore* to make CDBB smarter.

If the control data is from a CKPT writer, the BB coordinator will be notified of how much data the CKPT writer wants to write. Then the BB coordinator will check the *StatusStore* and reply to the CKPT writer with the best place to write. To make a decision, the BB coordinator will check the availability in the following priority order: the local BB→remote BBs→the PFS. Specifically, the status of the local BB will be checked first. If the local BB's remaining space is larger than the incoming CKPT data size, the BB coordinator will reply to the CKPT writer and let it write to the local BB. If the local BB does not have enough space, then the BB coordinator will check whether the remote BB with the most space left has enough space. If so, the remote BB will be selected as the destination. Note that whenever a BB is chosen to absorb the incoming CKPT data, the *StatusStore* will be updated accordingly to reflect that BB's increased space utilization. Finally, if none of the BBs have enough space, the CKPT writer will be notified to write to the PFS directly. Some corresponding location information of the CKPT data will be stored in a *LocationStore*, which will be used if the CKPT data is needed for restart purposes (not shown in the flowchart).

3.3 BB

Individual BBs are the building blocks of CDBB. We design and implement each BB instance using a classic producer-consumer model. We create two data structures to assist the management of each BB: a *DataStore* is the space storing CKPT data, and a *MetaStore* stores the corresponding metadata (e.g., data size, offset, CKPT data ID, and writing location) of the CKPT data for data draining and application restart purposes. For the producer of a BB, as long as the *DataStore* has enough space to accommodate the incoming CKPT data, it will insert the data into the *DataStore* and the *MetaStore*. For the consumer, as long as there are any CKPT data needing to be drained, it will use the information from the *MetaStore* to write the data in the *DataStore* to the PFS. Note that the CKPT data in the *DataStore* will be drained in a first-in-first-out (FIFO) manner controlled by the *MetaStore*. As long as one batch of CKPT data has been written to the PFS successfully, the BB will send its latest status (e.g., space utilization) to the BB coordinator to let it know more space is available in this BB.

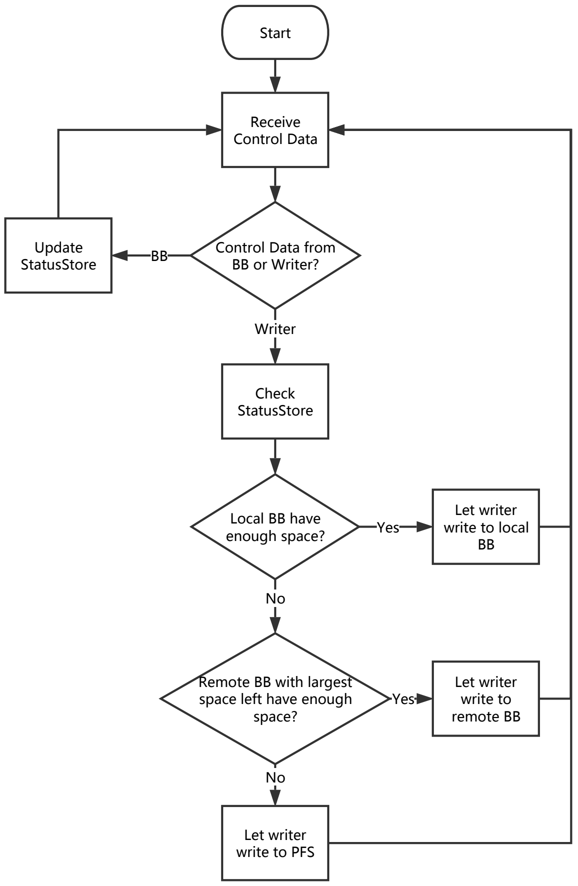

Start
Receive Control Data
Control Data from BB or Writer?
BB
Update StatusStore
Writer
Check StatusStore
Local BB have enough space?
Yes
Let writer write to local BB
No
Remote BB with largest space left have enough space?
Yes
Let writer write to remote BB
No
Let writer write to PFS

Figure 4: The BB coordinator checkpointing workflow

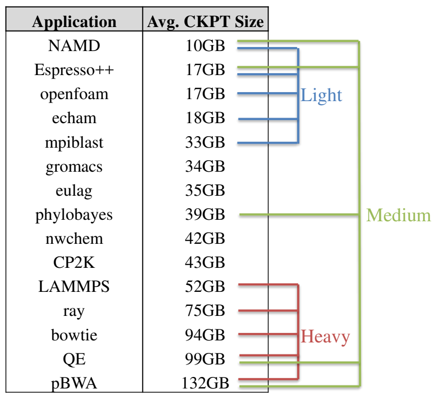
3.4 CKPT Writer

Each CKPT writer represents an HPC checkpointing process. Once the CKPT writers receive a checkpoint request from their CKPT coordinator, they will generate CKPT data by collecting the corresponding data associated with the application processes. Then CKPT writers will contact the BB coordinator to get directions about where to send the CKPT data. Each CKPT coordinator might have different CKPT frequency as specified by the application or the system administrator. CKPT tools, application types, and CKPT frequencies might affect the size of CKPT data.

4 PERFORMANCE EVALUATION

4.1 Implementation

To evaluate the performance of CDBB, we built a proof-of-concept prototype using C with the Message Passing Interface (MPI). Rank 0 is dedicated as the BB coordinator. *Rank* is an MPI term to denote each process. The last rank in each compute node acts as the local BB. The rest of the ranks in each compute node are CKPT writers. For each BB rank, it spawns two threads: one as the producer and the other as the consumer. Each application could have a different number of CKPT writers which will be awoken at the same time to generate CKPT data.



Application	Avg. CKPT Size
NAMD	10GB
Espresso++	17GB
openfoam	17GB
echam	18GB
mpiblast	33GB
gromacs	34GB
eulag	35GB
phylobayes	39GB
nwchem	42GB
CP2K	43GB
LAMMPS	52GB
ray	75GB
bowtie	94GB
QE	99GB
pBWA	132GB

Light
Medium
Heavy

Figure 5: Applications used for Light, Medium, and Heavy experiments

As the baseline comparison, we implement a traditional burst buffer prototype system in which each CKPT writer will only utilize its local BB. If the local BB is full, CKPT writers will write to the PFS directly. We call this prototype system the local distributed burst buffer (LDBB). For LDBB, similar to CDBB, one compute node has one BB and several CKPT writers. But LDBB does not have the BB coordinator. LDBB is implemented with C and MPI as well.

4.2 Testbed

We evaluate the performance of CDBB on the *Itasca* cluster located at the Minnesota Supercomputing Institute. *Itasca* is an HP Linux cluster with 1091 compute nodes, 8728 total cores, and 26 TB of aggregated RAM space. Each compute node is equipped with two quad-core 2.8 GHz Intel Nehalem processors and 24 GB of memory. It can achieve 3 GB/s node-to-node communication through a QDR InfiniBand interconnection. The back end storage is a Panasas ActiveStor 14 data-storage system with 1.281 PB of usable storage capacity and peak performance of 30 GB/s read/write and 270,000 IOPS.

Note that since there is no real NVRAM in our testbed system, we reserve 4 GB of memory on each compute node to emulate NVRAM-based burst buffers.

4.3 Evaluation Setup

We use the statistics collected by Kaiser *et al.* (Kaiser et al. 2016), as shown in Figure 5, to emulate multiple HPC applications running concurrently in an HPC cluster. These CKPT data were generated using the DMTCP tool (introduced in Section 2.1) with a frequency of every ten minutes. When creating CKPT data, DMTCP's compression feature was disabled. Almost all these applications were run for two hours with the exception that *bowtie* (after 50 minutes) and *pBWA* (after 110 minutes) finished earlier. Each application was distributed among 64 cores. A detailed description of all the applications can be found in (Kaiser et al. 2016). We design three representative experiments from the statistics to emulate scenarios under a light workload, a medium workload, and a heavy workload. We use the "Avg. CKPT Size" as the metric to describe applications listed in Figure 5. The *Light* experiment consists of the five smallest applications. The *Medium* experiment consists of the two smallest applications, one in the middle, and the two largest. The *Heavy* experiment consists of the five largest applications.

We run each experiment, *Light*, *Medium*, and *Heavy*, using 46 nodes (368 cores in total) from the cluster. Among them, 320 cores will act as CKPT writers, 46 cores will act as BBs (one BB per node), and one

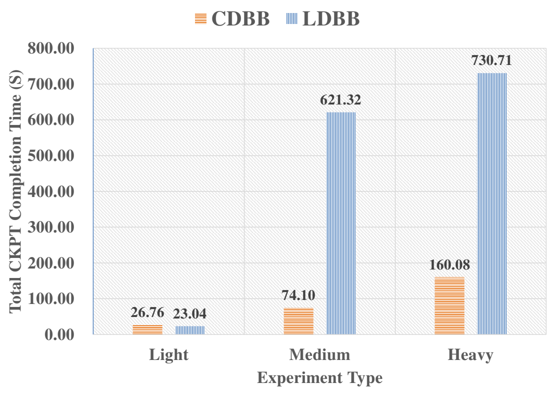


Figure 6: Combined total CKPT completion time for each experiment

core will act as the BB coordinator (on rank 0). There is one core left doing nothing. Among the 320 CKPT writers, every 64 of them will represent one application, which is the same configuration as the statistics collected in (Kaiser et al. 2016). For each experiment, there are five emulated applications. Each application is started randomly within the first ten minutes and runs for one hour. Once an application is started, it will perform CKPT every ten minutes. For each CKPT operation, 64 CKPT writers running the same application will each write the same amount of data such that their sum is equal to the "Avg. CKPT size" of that application as listed in Figure 5.

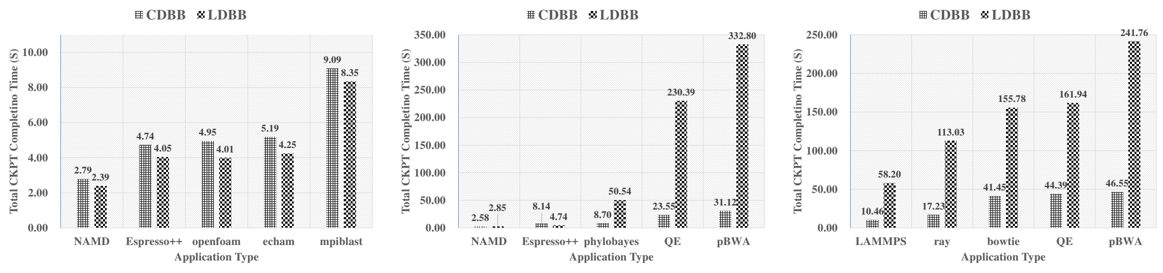
4.4 Evaluation Results

We measure each application's CKPT completion time for each CKPT operation. This time is measured as the difference between the ending time of the application's slowest finishing CKPT writer (among 64 CKPT writers) and the starting time of the CKPT operation. A CKPT writer finishes when its CKPT data are written completely, either to a BB or the PFS.

Figure 6 shows the combined total CKPT completion time for the three experiments. The total CKPT completion time is the sum of the CKPT completion times of all the applications' CKPT operations. In the *Light* experiment, CDBB, surprisingly at first look, takes 3.72 seconds longer than LDBB. However, this result is expected, since under the light workload, all CKPT data could be absorbed by local BBs, and CDBB's coordination capabilities do not help yet cause some overhead. Note that each application ran for one hour, so the overhead of 3.72 seconds is negligible ($\sim$0.1%).

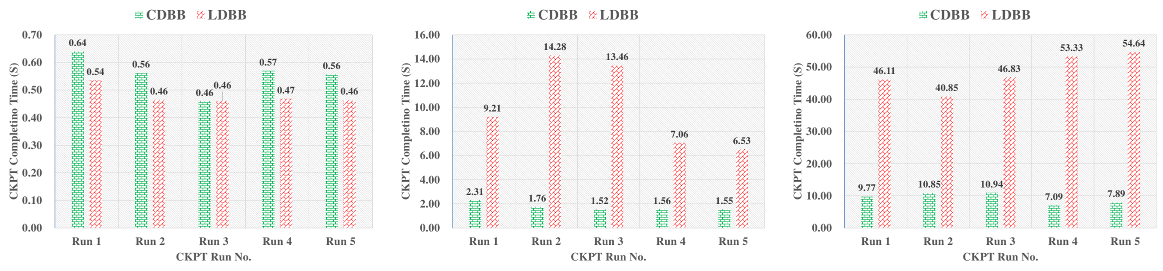
In the *Medium* and *Heavy* experiments, compared with LDBB, the results show that CDBB significantly shortens total CKPT completion time by 8.4$\times$ and 4.6$\times$, respectively. For CDBB, the total CKPT completion times of the three experiments are almost proportional to the total amount of CKPT data needing to be checkpointed. This relationship is ascribed to CDBB's ability to coordinate all available BBs to help absorb CKPT data. On the contrary, for LDBB, we find that when the local BBs are insufficient to accommodate all the incoming CKPT data, its CKPT speed becomes much slower since it has to wait until all the PFS writes are finished.

In addition, we plot total CKPT time by application (i.e., the sum of all of one application's CKPT operations) in Figure 7. For CDBB, similar to the above observation, each application's total CKPT time is proportional to its CKPT data size. For LDBB, an interesting finding is that the total CKPT times of the same application, *QE*, in the *Medium* (230.39 s) and *Heavy* (161.94 s) experiments are quite different. It is the same case for application *pBWA* (332.80 s versus 241.76 s). One possible reason is that the throughput



(a) The *Light* experiment (b) The *Medium* experiment (c) The *Heavy* experiment

Figure 7: Total CKPT completion time for each application. Note that y-axes are in different scales for the three figures.



(a) Application *NAMD* (b) Application *phylobayes* (c) Application *pBWA*

Figure 8: CKPT completion time of each CKPT operation for application *NAMD*, *phylobayes*, and *pBWA*. Note that y-axes are in different scales for the three figures.

of the PFS is quite unstable due to I/O contention caused by other running jobs in the *Itasca* cluster. In LDBB, application *QE* and *pBWA* need to write to the PFS, so their CKPT completion time will be affected. We further plot the CKPT completion time for each CKPT operation (CKPT run) in Figure 8. Here we only select three applications to plot due to space limitations: the smallest application, *NAMD*, from the *Light* experiment; the middle application, *phylobayes*, from the *Medium* experiment; and the largest application, *pBWA*, from the *Heavy* experiment. From Figure 8, we can see that for each run, CDBB's completion time variation for the same application is small whereas LDBB's completion time variation is larger due to the PFS I/O contention as mentioned above.

5 CONCLUSION

Slow checkpointing speed is the Achilles' heel of HPC systems due to the limited bandwidth of parallel file systems. To increase checkpointing speed, adding NVRAM into compute nodes as burst buffers has been previously proposed and studied. However, current HPC systems relying only on local burst buffers could waste precious NVRAM resources. In this paper, by maximizing burst buffer utilization, our newly proposed burst buffer coordination system, named collaborative distributed burst buffer (CDBB), can further speed up checkpointing. By building a proof-of-concept prototype, we demonstrate the potential of CDBB. Under a light workload, CDBB only introduces negligible overhead, and under medium and heavy workloads, CDBB can improve CKPT speed by up to 8.4×.

ACKNOWLEDGMENT

This work is partially supported by the following NSF awards: 1305237, 1421913, 1439622 and 1525617. This work is also supported by Hewlett Packard Enterprise.

REFERENCES

Ansel, J., K. Arya, and G. Cooperman. 2009. "DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop". In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pp. 1–12. Washington, DC, USA, IEEE Computer Society.

Bautista-Gomez, L., S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. 2011. "FTI: High Performance Fault Tolerance Interface for Hybrid Systems". In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pp. 32:1–32:32. New York, NY, USA, ACM.

Coughlin, T., and E. Grochowski. 2015. "Emerging Non-Volatile Memory and Spin Logic Technology and Memory Manufacturing Report". Technical report, Coughlin Associates.

Fan, Z., Y. Chen, L. Wang, L. Shu, and T. Hara. 2011, Dec. "Removing Heavily Curved Path: Improved DV-hop Localization in Anisotropic Sensor Networks". In *2011 Seventh International Conference on Mobile Ad-hoc and Sensor Networks*, pp. 75–82.

Fan, Z., D. H. Du, and D. Voigt. 2014. "H-ARC: A non-volatile memory based cache policy for solid state drives". In *MSST*, pp. 1–11.

Fan, Z., A. Haghdoost, D. H. Du, and D. Voigt. 2015. "I/O-Cache: A Non-volatile Memory Based Buffer Cache Policy to Improve Storage Performance". In *MASCOTS*, pp. 102–111.

Gallagher, W. J., and S. S. P. Parkin. 2006. "Development of the Magnetic Tunnel Junction MRAM at IBM: From First Junctions to a 16-Mb MRAM Demonstrator Chip". *IBM Journal of Research and Development* vol. 50 (1), pp. 5–23.

Intel 2015. "3D XPoint Unveiled: The Next Breakthrough in Memory Technology". Technical report, Intel Corporation.

Kaiser, J., R. Gad, T. Süß, F. Padua, L. Nagel, and A. Brinkmann. 2016, Sept. "Deduplication Potential of HPC Applications Checkpoints". In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 413–422.

Kawahara, T., K. Ito, R. Takemura, and H. Ohno. 2012. "Spin-transfer torque RAM technology: review and prospect". *Microelectronics Reliability* vol. 52 (4), pp. 613–627.

Lee, B. C., E. Ipek, O. Mutlu, and D. Burger. 2009. "Architecting Phase Change Memory as a Scalable DRAM Alternative". In *ISCA*, pp. 2–13.

Liu, N., J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. 2012, April. "On the role of burst buffers in leadership-class storage systems". In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–11.

Micron 2015. "3D XPoint Technology". Technical report, Micron Technology.

Qureshi, M. K., V. Srinivasan, and J. A. Rivers. 2009. "Scalable High Performance Main Memory System Using Phase-change Memory Technology". In *ISCA*, pp. 24–33.

Rajachandrasekar, R., A. Moody, K. Mohror, and D. K. D. Panda. 2013. "A 1 PB/s File System to Checkpoint Three Million MPI Tasks". In *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pp. 143–154. New York, NY, USA, ACM.

Raoux, S., F. Xiong, M. Wuttig, and E. Pop. 2014. “Phase change materials and phase change memory”. *MRS Bulletin* vol. 39 (08), pp. 703–710.

Thapaliya, S., P. Bangalore, J. Lofstead, K. Mohror, and A. Moody. 2016, Aug. “Managing I/O Interference in a Shared Burst Buffer System”. In *2016 45th International Conference on Parallel Processing (ICPP)*, pp. 416–425.

Wang, T., K. Mohror, A. Moody, K. Sato, and W. Yu. 2016. “An Ephemeral Burst-buffer File System for Scientific Applications”. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’16, pp. 69:1–69:12. Piscataway, NJ, USA, IEEE Press.

Wang, T., S. Oral, M. Pritchard, B. Wang, and W. Yu. 2015, Sept. “TRIO: Burst Buffer Based I/O Orchestration”. In *2015 IEEE International Conference on Cluster Computing*, pp. 194–203.

Wang, T., S. Oral, Y. Wang, B. Settlemyer, S. Atchley, and W. Yu. 2014, Oct. “BurstMem: A high-performance burst buffer system for scientific applications”. In *2014 IEEE International Conference on Big Data (Big Data)*, pp. 71–79.

Waser, R., R. Dittmann, G. Staikov, and K. Szot. 2009. “Redox-based resistive switching memories–nanoionic mechanisms, prospects, and challenges”. *Advanced materials* vol. 21 (25-26), pp. 2632–2663.

Wu, F., Z. Fan, M. C. Yang, B. Zhang, X. Ge, and D. H. C. Du. 2017, Nov. “Performance Evaluation of Host Aware Shingled Magnetic Recording (HA-SMR) Drives”. *IEEE Transactions on Computers* vol. 66 (11), pp. 1932–1945.

Wu, F., M.-C. Yang, Z. Fan, B. Zhang, X. Ge, and D. H. Du. 2016. “Evaluating Host Aware SMR Drives”. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*. Denver, CO, USENIX Association.

AUTHOR BIOGRAPHIES

ZIQI FAN received his Ph.D. at the University of Minnesota, Minneapolis. His research interests include non-volatile memory, data deduplication, and distributed systems. He currently works at NovuMind Inc. His email address is fanxx234@umn.edu.

FENGGANG WU is currently a Ph.D. student at the University of Minnesota, Minneapolis. His research interests include SMR drives, persistent memory, and key-value store. His email address is wuxx0835@umn.edu.

JIM DIEHL is currently a Ph.D. student at the University of Minnesota, Minneapolis. He is interested in tiered storage and new storage technologies including Kinetic key-value drives. His email address is jdiehl@umn.edu.

DAVID H.C. DU is currently the Qwest Chair Professor of Computer Science and Engineering at the University of Minnesota, Minneapolis. He is the Center Director of the NSF multi-university I/UCRC Center of Research in Intelligent Storage (CRIS). His email address is du@umn.edu.

DOUG VOIGT is a Distinguished Technologist with Hewlett Packard Enterprise. With over 30 years experience in HPE’s storage business, Doug is involved with storage strategy, architecture and intellectual property. His email address is doug.voigt@hpe.com.

PREDICTING SYSTEM PERFORMANCE BY INTERPOLATION USING A HIGH-DIMENSIONAL DELAUNAY TRIANGULATION

Tyler H. Chang

Dept. of Computer Science
Virginia Polytechnic Institute
& State University (VPI&SU)
Blacksburg, VA 24061
thchang@vt.edu

Layne T. Watson

Depts. of Computer Science,
Mathematics, and
Aerospace & Ocean Engineering
VPI&SU, Blacksburg, VA 24061

Thomas C. H. Lux

Dept. of Computer Science
VPI&SU, Blacksburg, VA 24061

Jon Bernard

Dept. of Computer Science, VPI&SU
Blacksburg, VA 24061

Bo Li

Dept. of Computer Science, VPI&SU
Blacksburg, VA 24061

Li Xu

Dept. of Statistics, VPI&SU
Blacksburg, VA 24061

Godmar Back

Dept. of Computer Science, VPI&SU
Blacksburg, VA 24061

Ali R. Butt

Dept. of Computer Science, VPI&SU
Blacksburg, VA 24061

Kirk W. Cameron

Dept. of Computer Science, VPI&SU
Blacksburg, VA 24061

Yili Hong

Dept. of Statistics, VPI&SU
Blacksburg, VA 24061

ABSTRACT

When interpolating computing system performance data, there are many input parameters that must be considered. Therefore, the chosen multivariate interpolation model must be capable of scaling to many dimensions. The Delaunay triangulation is a foundational technique, commonly used to perform piecewise linear interpolation in computer graphics, physics, civil engineering, and geography applications. It has been shown to produce a simplex based mesh with numerous favourable properties for interpolation. While computation of the two- and three-dimensional Delaunay triangulation is a well-studied problem, there are numerous technical limitations to the computability of a high-dimensional Delaunay triangulation. This paper proposes a new algorithm for computing interpolated values from the Delaunay triangulation without computing the complete triangulation. The proposed algorithm is shown to scale to over 50 dimensions. Data is presented demonstrating interpolation using the Delaunay triangulation in a real world high performance computing system problem.

Keywords: Delaunay triangulation, multivariate interpolation, performance, performance variability, high-dimensional data

1 INTRODUCTION

High performance computing (HPC) system performance is a topic of significant importance in the scientific community. There are many measures by which HPC performance can be evaluated, such as throughput, energy consumption, and compute time for a given task. Additionally, since each of these measures tends to fluctuate stochastically between independent runs, various summary statistics such as the mean, median, maximum, minimum observed value, or variance can be collected. Furthermore, each of these measures is generally affected by numerous factors, such as system specifications, system configuration, application type, and application dependent inputs. In this way, each measure can be thought of as a function of various system and application level parameters.

For a performance evaluator benchmarking a machine, each of these measures could easily be collected for a finite set of system and application "signatures" by running several applications over various machine configurations (with multiple runs at each configuration). After collecting the data, suppose an evaluator wanted to predict performance measures for some *different* signature(s). Assuming that there is an underlying relationship between the measured statistics and the set of system and application parameters that contribute to the signature, it is reasonable to make this prediction with a multivariate interpolatory model.

Given a set of data points, an interpolant is an approximation of the underlying function f that exactly matches all given values. A *multivariate* interpolant is characterized by the presence of more than one input variable. Therefore, a multivariate interpolant of f is a function over d-tuples of input parameters in the domain of f, where the domain of f has typically been mapped to a subset of $\mathbb{R}^d$. There are many parameters that can affect the performance of a HPC system, and therefore the dimension of the space $\mathbb{R}^d$ could be quite large.

A triangulation of a set of points P in $\mathbb{R}^d$ is a division of the convex hull of P, denoted $CH(P)$, into d-simplices with vertices in P. The Delaunay triangulation of P, denoted $DT(P)$, is a particular triangulation defined in terms of the empty circumsphere property, and is commonly used as an interpolatory mesh in fields such as geographic information systems (GIS), civil engineering, physics, and computer graphics. Two- and three-dimensional Delaunay triangulations (Delaunay triangulations of points in $\mathbb{R}^2$ and $\mathbb{R}^3$) are readily computable, but there are theoretical limitations to the computability of a high-dimensional Delaunay triangulation.

The rest of the paper is organized as follows. Section 2 will discuss background related to the theoretical advantages and computability of the Delaunay triangulation. Section 3 presents a new algorithm for interpolating high-dimensional data with the Delaunay triangulation. Section 4 contains performance data demonstrating the scalability of the proposed algorithm. Section 5 details the predictive performance of the proposed algorithm for a four-dimensional HPC system interpolation problem. Section 6 outlines future work.

2 BACKGROUND

2.1 The Delaunay Interpolant

The Delaunay triangulation is defined as the geometric dual of the Voronoi diagram, or equivalently, as any triangulation satisfying the empty circumsphere property defined below (**?**, **?**, **?**). See Figure 1 for a visual.

Definition 1. *A Delaunay triangulation $DT(P)$ of a set of points P in $\mathbb{R}^d$ is any triangulation of P such that for each d-simplex $s \in DT(P)$, the interior of the sphere circumscribing s contains no point in P.*

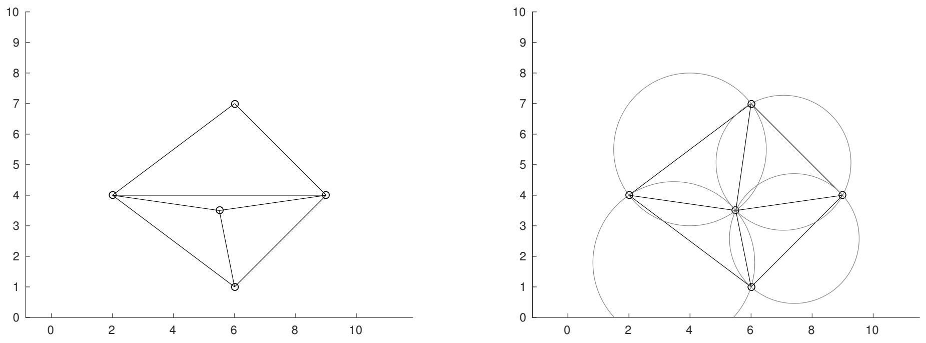


Figure 1: A triangulation in $\mathbb{R}^2$ (left) and the Delaunay triangulation (right).

The Delaunay triangulation exists for every nonempty finite set of points P in $\mathbb{R}^d$ that do not all lie in the same lower-dimensional linear manifold in $\mathbb{R}^d$ (or equivalently, if $CH(P)$ has nonzero volume). This is a direct corollary to the Delaunay triangulation's duality with the Voronoi diagram. Note that the existence of $DT(P)$ requires that n (the size of P) be greater than or equal to $d+1$, otherwise all n points will lie in the same $n-1$-dimensional linear manifold trivially. In the context of interpolation, the degenerate case where $DT(P)$ does not exist and $n \geq d+1$ can be seen as an over-parameterization of the underlying function f. The points are said to be in *general position* if the Delaunay triangulation exists, and no $d+2$ points in P lie on the same $d-1$-sphere; in this case the Delaunay triangulation of P is unique (**?**; **?**).

Given a triangulation T of P and function values $f_i = f(x_i)$ for all $x_i \in P$, f can be interpolated at any point $q \in CH(P)$ using a piecewise linear interpolant $\hat{f}$ defined as follows: Let s be a d-simplex in T with vertices $s_1, s_2, \ldots, s_{d+1}$ and $q \in s$. Then $q = \sum_{i=1}^{d+1} w_i s_i$, $\sum_{i=1}^{d+1} w_i = 1$, $w_i \geq 0$, $i = 1, \ldots, d+1$, and

$$\hat{f}(q) = f(s_1)w_1 + f(s_2)w_2 + \ldots + f(s_{d+1})w_{d+1}. \tag{1}$$

It is common to define $\hat{f}$ using $DT(P)$ since the Delaunay triangulation enjoys several properties that are considered optimal over all simplex based interpolation meshes (**?**, **?**). Consequently, the Delaunay triangulation finds wide use as a model for multivariable piecewise linear interpolation in fields such as GIS, civil engineering, physics, and computer graphics (**?**, **?**).

2.2 Related Works and Challenges

The two- and three-dimensional Delaunay triangulation (triangulations of points in $\mathbb{R}^2$ and $\mathbb{R}^3$) can be efficiently computed in $\mathcal{O}(n \log n)$ time (**?**). In order to compute an interpolated value $\hat{f}(q)$ for some two- or three-dimensional point $q \in CH(P)$, one must also locate the simplex in $DT(P)$ containing q. This simplex can be located using the "jump-and-walk" algorithm in sublinear expected time (**?**). Given this simplex, it is trivial to compute $\hat{f}(q)$ using (1).

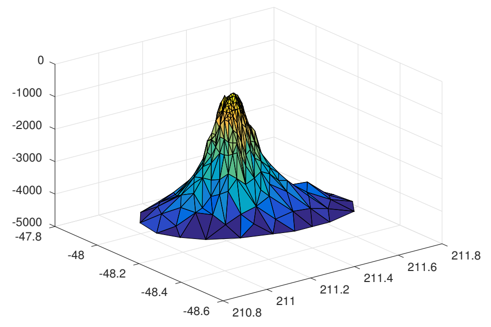


Figure 2: The 2D Delaunay triangulation of height data for a sea mountain is used to construct a piecewise linear model of the mountain in `Matlab`.

However, in higher dimensions, there is a theoretical limitation to the computability of the Delaunay triangulation. In the worst case, a d-dimensional Delaunay triangulation of n points could contain up to $\mathscr{O}(n^{\lceil d/2 \rceil})$ simplices (**?**). Therefore, the lower bound on the worst case time complexity for computing $DT(P)$ is $\mathscr{O}(n^{\lceil d/2 \rceil})$. Furthermore, in the worst case, $\mathscr{O}(n^{\lceil d/2 \rceil})$ space is required to store $DT(P)$. It is important to note that in the common case, significantly fewer than $n^{\lceil d/2 \rceil}$ simplices will be contained in $DT(P)$. However, the common case size still tends to grow exponentially with respect to d (**?**).

Current state-of-the art Delaunay triangulation techniques generally scale to six or seven dimensions for large input sets and focus on either minimizing compute time in the general case (**?**) or minimizing the storage overhead (**?**). Others have aimed to avoid the scalability problem by computing alternative simplical complexes with still favourable topological properties and improved scalability (**?**).

3 PROPOSED ALGORITHM

In order to interpolate over large data sets where the independent variables span many dimensions (such as system performance data sets), a more scalable approach to the Delaunay triangulation is required. Since it is intractable to compute the *entire* Delaunay triangulation in large dimensions, the following observation is useful: Given a set of points P in $\mathbb{R}^d$, one can compute the Delaunay interpolant $\hat{f}(q)$ at a point $q \in CH(P)$ knowing only the simplex $s \in DT(P)$ such that q is in s. It should be noted that q could lie on a face shared by multiple simplices in $DT(P)$. However, in these cases, $\hat{f}(q)$ will be the same regardless of the simplex chosen since only the vertices shared by *all* containing simplices will contribute to $\hat{f}(q)$. This observation reduces the problem of computing the entire Delaunay triangulation to that of computing a specific simplex in the Delaunay triangulation.

3.1 Algorithm Outline

It is possible to grow an initial Delaunay simplex using any point in P as the starting vertex. This approach is described as the basis for both *incremental construction* and *divide and conquer* algorithms in **?**. Given a point $q \in CH(P)$ to interpolate at and a simplex s defined by points in P, it must be the case that q is either in s, or q is across the hyperplane defined by at least one of the facets of s. Therefore, given any s not containing q, one can construct a new Delaunay simplex s^* closer to q by completing a facet of s that is shared with s^*. The technique used for completing an open facet of a simplex is outlined in **?**.

Using this methodology and a deterministic variant of the "walk" described in **?**, it is possible to continually advance toward the simplex containing q via an iterative process. Note, that for every d-simplex $s \in DT(P)$

and $q \in \mathbb{R}^d$, q is a *unique* affine combination of the vertices of s. Furthermore, if q is *in* s, then this affine combination will also be convex by the definition of a simplex. Therefore, one can terminate based on the nonnegativity of the affine weights for q with respect to the vertices of the current simplex s. Conveniently, upon termination, the convex weights for q are the exact weights needed to linearly interpolate at q within s, as defined previously in (1). Pseudo code for the proposed algorithm is provided below.

Algorithm.

Let P be an array of points in $\mathbb{R}^d$.
Let $q \in CH(P)$ be a point to interpolate at.
Let $f_i = f(x_i)$ be known for all $x_i \in P$, and let $f(s) = \left(f_{i_1}, f_{i_2}, \ldots, f_{i_{d+1}}\right)$ for d-simplex s with vertices $x_{i_1}, x_{i_2}, \ldots, x_{i_{d+1}}$.
Let MakeFirstSimplex(P) be a function that constructs an initial Delaunay simplex from points in P.
Let CompleteSimplex(σ, q, P) be a function that completes the facet σ with a point from P that is on the same side of the hyperplane containing σ as q.
Let AffineWeights(q, s) be a function that returns the affine weights that give q as a combination of the vertices of s.
Let MinIndex(w) return the index of the most negative element w_i.
Let DropVertex(s, i) return the facet of s that results from dropping vertex i.
The following algorithm computes the interpolant $\hat{f}(q)$ using the Delaunay triangulation of P.

```
$s$ = MakeFirstSimplex($P$);
$w$ = AffineWeights($q$, $s$);
while $w_i < 0$ for some $1 \le i \le d+1$ do
   $i$ = MinIndex($w$);
   $\sigma$ = DropVertex($s$, $i$);
   $s$ = CompleteSimplex($\sigma$, $q$, $P$);
   $w$ = AffineWeights($q$, $s$);
end while
return $\hat{f}(q) = \langle w, f(s) \rangle$ (inner product);
```

3.2 Complexity Analysis

3.2.1 Time Complexity

The construction of the first simplex can be formulated as a sequence of least squares (LS) problems ranging in size from $2 \times d$ to $d \times d$. Using `DGELS` from `LAPACK`, each LS problem can be solved in $\mathcal{O}(d^3)$ time. At all $d-1$ sizes, one must solve up to n LS problems, taking the point that produces the minimum residual as the next vertex. Therefore, the total computation time for the first simplex will be $\mathcal{O}(nd^4)$.

To complete a simplex (one iteration of the walk) requires at most n linear solves, performed with `DGESV` from `LAPACK` in $\mathcal{O}(nd^3)$ total time. In each iteration, a flip toward q is performed by dropping a vertex in s to get an open facet σ, then completing σ with a point on the opposite halfspace (as defined by the hyperplane containing σ). In the expected case, for uniformly distributed points, the simplex containing q is located in $\mathcal{O}(n^{1/d})$ iterations (**?**, **?**). Therefore, the worst case time complexity for performing the "walk" from the initial simplex to the simplex containing q is expected to be $\mathcal{O}(n^{1+\frac{1}{d}}d^3)$.

From the previous analysis, using the proposed algorithm to interpolate at m points from a set of n points in $\mathbb{R}^d$ will take $\mathcal{O}(mn^{1+\frac{1}{d}}d^3 + mnd^4)$ total time. This is a significant improvement over the non polynomial worst case for computing the complete triangulation.

3.2.2 Space Complexity

Recall from Section 3.1 that the size of the Delaunay triangulation grows exponentially with the dimension. Therefore, space complexity is equally as concerning as time complexity since, for large d, one cannot store the exponentially sized triangulation in memory. Another advantage of the proposed algorithm is that any computed simplex that does not contain any interpolation point can be discarded immediately.

Therefore, the required space for computing the Delaunay interpolant is reduced to:

- $\mathcal{O}(nd)$ space for the n input points in $\mathbb{R}^d$;
- $\mathcal{O}(md)$ space for storing the m interpolation points in $\mathbb{R}^d$, the m containing simplices of size $d+1$ each, and the m convex coordinate vectors of size $d+1$ each;
- $\mathcal{O}(d^2)$ space for storing the $d \times d$ matrices involved in performing linear solves;
- Other temporary storage arrays that require $\mathcal{O}(d)$ space.

This makes the total space complexity $\mathcal{O}(nd+md+d^2)$. Since no triangulation can exist unless $n > d$, the space complexity can be further reduced to $\mathcal{O}(nd+md)$, which is approximately the same size as the input.

3.3 Optimizations

There are two optimizations that are easily implemented to improve the performance of the proposed algorithm. First, one could identify the point $p \in P$ that is a nearest neighbor to q with respect to Euclidean distance in $\mathcal{O}(n)$ time and build the first simplex off of p instead of an arbitrarily chosen point. For interpolating at a single point, this typically leads to location of the simplex containing q in $\mathcal{O}(d \log d)$ iterations of the walk. Therefore, the expected time complexity is further reduced to $\Theta(mnd^4 \log d)$.

Furthermore, when interpolating many points, it is often the case that some simplex or simplices in the iterative process contain interpolation points that have not yet been resolved. With no increase in total time complexity, it is possible to check if the current simplex s contains *any* of the unresolved interpolation points. If the points being interpolated are tightly clustered, it is typical for them to all be contained in a small number of Delaunay simplices, significantly reducing total computation time.

3.4 Extrapolation

Up until this point, the proposed algorithm has only covered interpolation cases (when q is in $CH(P)$). Often, however, it is reasonable to make a prediction about some extrapolation points Z that are *slightly* outside $CH(P)$. In these cases, it is most reasonable to project each $z \in Z$ onto $CH(P)$ and interpolate at each projection $\hat{z}$, provided the residual $r = \|z-\hat{z}\|_2$ is small. The projection of z onto $CH(P)$ can be easily reformulated as an inequality constrained least squares problem, whose efficient solution is described in **?**.

4 RUNTIME ANALYSIS

A serial implementation of the proposed algorithm has been coded in ISO Fortran 2003 with extra machinery added for numerical stability and degenerate case handling. This code was tested for correctness against `Qhull` on both real world and pseudo-random data sets in up to four dimensions, with a mix of degenerate and general position input points. `Qhull` is an industry standard used for computing high-dimensional Delaunay triangulations in `Matlab`, `SciPy`, and `R`. An analysis of the `Qhull` algorithm is in **?**. After confirming correctness with lower dimensional data, run times were gathered for pseudo-randomly generated point sets in up to 64 dimensions.

The following run times were gathered on an Intel i7-3770 CPU @3.40 GHz running CentOS release 7.3.1611. All run times were averaged over a sample size of 20 runs, each performed on a unique pseudo-randomly generated input data set (generated with the Fortran intrinsic random number generator). Times were recorded with the Fortran intrinsic `CPU_TIME` function, which is accurate up to microsecond resolution. Table 1 details average run times for interpolating at uniformly spaced interpolation points using a five-dimensional input data set ranging in size from $n = 2000$ points to $n = 32,000$ points. Table 2 details average run times for interpolating at interpolation points that were clustered within a hypercube with 10% of the original point-set's diameter using a five-dimensional input data set ranging in size from $n = 2000$ points to $n = 32,000$ points. Table 3 details average run times for interpolating at a single point in $d = 2$ up to $d = 64$ dimensions over input data sets ranging in size from $n = 2,000$ points up to $n = 32,000$ points. For reference, the data in Tables 1 and 2 could be compared to the computation times for the complete five-dimensional Delaunay triangulation presented in **?**.

Table 1: Average runtime in seconds for interpolating at uniformly spaced interpolation points for 5D pseudo-randomly generated input points.

	$n = 2000$	$n = 8000$	$n = 16,000$	$n = 32,000$
32 interp. pts	0.3 s	2.7 s	9.6 s	35.7 s
1024 interp. pts	2.5 s	11.6 s	28.9 s	79.1 s

Table 2: Average runtime in seconds for interpolating at clustered interpolation points for 5D pseudo-randomly generated input points.

	$n = 2000$	$n = 8000$	$n = 16,000$	$n = 32,000$
32 interp. pts	0.2 s	2.2 s	8.4 s	33.0 s
1024 interp. pts	0.2 s	2.5 s	9.2 s	35.2 s

Table 3: Average runtime in seconds for interpolating at a single point in up to 64 dimensional space for pseudo-randomly generated input points.

	$n = 2000$	$n = 8000$	$n = 16,000$	$n = 32,000$
$d = 2$	0.1 s	1.7 s	6.8 s	27.0 s
$d = 8$	0.2 s	2.5 s	9.6 s	37.9 s
$d = 32$	1.4 s	9.5 s	29.7 s	101.1 s
$d = 64$	13.2 s	60.1 s	138.6 s	349.1 s

A keen reader may recall from Section 3.3 that with optimizations, linear scaling with respect to n (the input point size) is expected. However, in Table 3, a quadratic relationship is observed below 64 dimensions. This is actually an artifact of the implementation, which includes extra code to promote elegant error handling and numerical stability. Among other functions, the added code checks the diameter and closest-pair distance of the data-set in $\mathscr{O}(n^2)$ time and must perform at least one check for rank deficiency using the singular-value decomposition of a $d \times d$ matrix. If the input is assumed to consist of legal values with all points in general position, then these checks can be dropped, significantly improving the run time.

5 PREDICTIVE ACCURACY

There are many theoretical advantages to interpolating using the Delaunay triangulation. To demonstrate its predictive power, in this section, the code described in Section 4 is applied to an HPC system performance problem.

5.1 Problem Summary

For this problem, the performance measure chosen is *throughput variance*, which is a quantifier of performance variability. Performance variability refers to the inherent "jitter" observed in performance values between multiple independent runs of an identical application. This variability can be of concern, particularly in large scale systems such as HPC and Cloud systems (**?**, **?**, **?**). Specifically for this work, throughput variability is considered with respect to various instances of the `IOzone` benchmark being run over a cluster of identical machines with identical system and application level parameters. Unless otherwise stated, all system and application level parameters are assumed to be fixed at some reasonable value.

5.2 Data Collection

Data for these experiments has been gathered at Virginia Tech on a homogeneous cluster of shared-memory nodes running Ubuntu 14.04 LTS on a dedicated 2TB HDD. Each node is a 2 socket, 4 core hyperthreaded Intel Xeon E5-2623 v3 (Haswell) processor with 32 GB DDR4. Data has been gathered by running the `IOzone` benchmark. The `IOzone` benchmark measures read/write throughputs by reading and writing files of configurable size, broken up into configurable record sizes, utilizing a configurable number of threads. For more information on `IOzone`, see `www.iozone.org`. Up to 13 different variations of read and write tasks can be tested, but in this paper only the `fread` task is considered. To generate each data point, the `IOzone` benchmark has been run independently 40 times with identical settings, and the variance has been computed for the maximum throughput of the `fread` task using

$$\sigma^2 = \frac{\sum_{i=1}^{40}(t_i - \mu)^2}{39} \tag{2}$$

where t_i represents the throughput for each of the 40 runs, and μ is the mean observed throughput over all 40 runs. The variance is modeled as a function of four system and application parameters, chosen because of their relevance to the `IOzone` benchmark. These parameters are thread count, CPU frequency, file size, and record size. Note that the parameters thread count, file size, and record size are specifiable as `IOzone` inputs, while CPU frequency must be manually set for each run using system tools. To avoid biasing the data, the CPU cache has been purged between each run of an `IOzone` test.

For each parameter, several values have been chosen spanning a range of reasonable values, and the observed variance has been calculated for each combination of settings using (2). For the data presented, the values chosen are in Table 4. Note that some combinations of parameters are not viable (specifically, the record size cannot be greater than the file size). These combinations have been omitted when collecting data. Observe that there are 6 valid combinations of file and record size, 9 thread counts, and 16 distinct CPU frequencies. This results in $6 \times 9 \times 16 = 864$ total data points.

Table 4: Values for adjusted parameters. Note, the frequency 3.001 GHz results from overclocking.

	values
file size (KB)	64, 256, 1024
record size (KB)	32, 128, 512
thread count	1, 2, 4, 8, 16, 32, 64, 128, 256
frequency (GHz)	1.2, 1.4, 1.5, 1.6, 1.8, 1.9, 2.0, 2.1, 2.3, 2.4, 2.5, 2.7, 2.8, 2.9, 3.0, 3.001

The relationship between variance and each individual problem dimension has been observed to exhibit highly nonlinear behavior, making a simple linear fit an extremely poor solution to this problem. The Delaunay interpolant is better able to accommodate this nonlinearity since the Delaunay interpolant is only piecewise linear.

5.3 Model Evaluation

From the collected data, the following points in (3) have been selected for testing the Delaunay prediction.

$$\begin{aligned} q_1 &= (\text{fsize} = 256, \text{rsize} = 128, \text{threads} = 2, \text{freq} = 1.9) \\ q_2 &= (\text{fsize} = 256, \text{rsize} = 128, \text{threads} = 2, \text{freq} = 2.9) \\ q_3 &= (\text{fsize} = 1024, \text{rsize} = 32, \text{threads} = 4, \text{freq} = 2.9) \end{aligned} \tag{3}$$

Note that because of the restriction on valid combinations of file and record size, it is not possible to select points strictly inside the convex hull. Consequently, all the points in (3) are on the boundary of the convex hull. Various percentages of the remaining points are used by the Delaunay interpolant to predict the value of the throughput variance $f(q_i)$ for all q_i in (3). For each of these "training percentages," up to 200 Delaunay interpolations are calculated using different pseudo-random samplings from the complete set of data points, with a bias toward uniformly distributed samplings. These samplings are constrained in that the prediction of $f(q_i)$ cannot be based off a sampling that includes q_i, and q_i cannot be outside the convex hull of the selected points. Also, repeated use of the same sampling in a single batch of 200 samplings has been forbidden. Note that because of the constraints, in some cases, less than 200 samplings could be gathered.

Figures 3, 4, and 5 show box plots of the relative errors observed when using the Delaunay interpolant to predict q_1, q_2, and q_3 at each training percentage. The relative error was computed using: $\left|\hat{f}(q_i) - f(q_i)\right| / f(q_i)$ for $i = 1, 2, 3$.

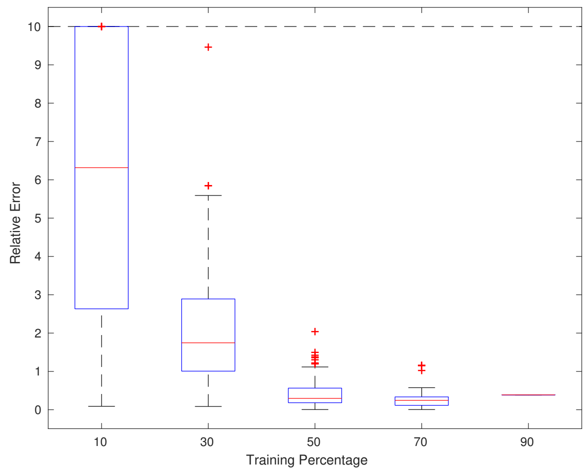


Figure 3: Box plot of the relative error for up to 200 variance predictions at q_1 (3) using the Delaunay interpolant with various percentages of the total available data. Note: Relative errors greater than 10 (1000%) have been truncated.

Using 90% of the 863 remaining data points (after excluding the point q_i that is being interpolated at) the Delaunay interpolant is seen to be fairly accurate, taking into consideration the difficulty of the problem. The Delaunay interpolant remains relatively accurate using as little as 50% of the remaining data. Note that for the large training percentages, the boxes in Figures 3, 4, and 5 are very narrow. This is because large samplings have a relatively low probability for dropping a vertex from the Delaunay simplex containing q_i, and unless some vertex of the containing Delaunay simplex is dropped, the interpolated value will not change for the Delaunay interpolant.

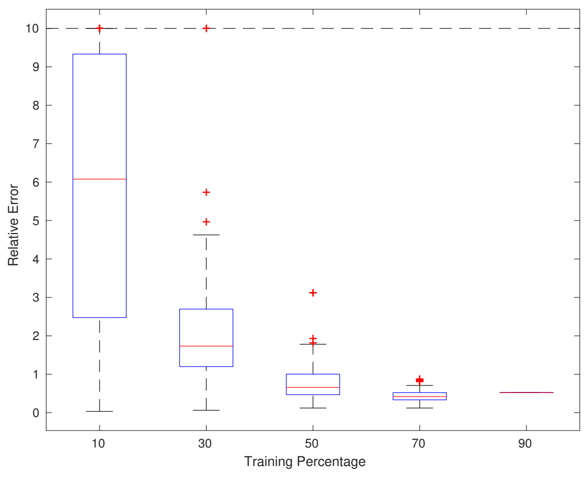


Figure 4: Box plot of the relative error for up to 200 variance predictions at q_2 (3) using the Delaunay interpolant with various percentages of the total available data. Note: Relative errors greater than 10 (1000%) have been truncated.

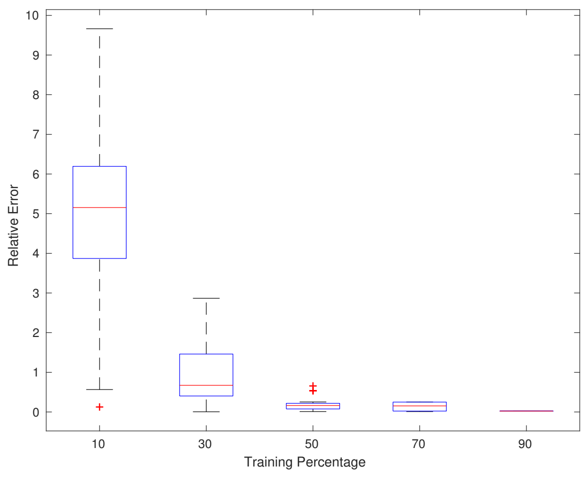


Figure 5: Box plot of the relative error for up to 200 variance predictions at q_3 (3) using the Delaunay interpolant with various percentages of the total available data.

6 CONCLUSION AND FUTURE WORK

In this paper, a new Delaunay triangulation algorithm is proposed for interpolating in point clouds in arbitrary dimension d. This is achieved by computing a relatively small number of simplices from the complete Delaunay triangulation in polynomial time. The proposed algorithm scales linearly with respect to the size of the input in the expected case, regardless of the dimension. The described algorithm was demonstrated on a system performance problem, where it was used to make accurate predictions about throughput variance. It should be noted that the Delaunay triangulation is not limited to interpolation and is widely used in mesh generation, principle component analysis, and topological data analysis. Its geometric dual, the Voronoi diagram, can also be used to make rapid nearest neighbor queries and is the basis for another interpolation technique, *natural neighbor interpolation*. In future work, the methods used to construct and locate a single

Delaunay simplex could potentially be extended for other Delaunay triangulation applications. In particular, knowledge of a Voronoi cell can be achieved by computing the star of simplices incident at a given vertex.

REFERENCES

AUTHOR BIOGRAPHIES

TYLER H. CHANG is a Ph.D. student at VPI&SU studying computer science under Dr. Layne Watson.

LAYNE T. WATSON (Ph.D., Michigan, 1974) has interests in numerical analysis, mathematical programming, bioinformatics, and data science. He has been involved with the organization of HPCS since 2000.

THOMAS C. H. LUX is a Ph.D. student at VPI&SU studying computer science under Dr. Layne Watson.

JON BERNARD is a Ph.D. student at VPI&SU studying computer science under Dr. Kirk Cameron.

BO LI is a senior Ph.D. student at VPI&SU studying computer science under Dr. Kirk Cameron.

LI XU is a Ph.D. student at VPI&SU studying statistics under Dr. Yili Hong.

GODMAR BACK (Ph.D., University of Utah, 2002) has broad interests in computer systems, with a focus on performance and reliability aspects of operating systems and virtual machines.

ALI R. BUTT (Ph.D., Purdue, 2006) has interests in cloud computing and distributed systems.

KIRK W. CAMERON (Ph.D., Louisiana State, 2000) has interests in computer systems design, performance analysis, and power and energy efficiency.

YILI HONG (Ph.D., Iowa State, 2009) has interests in engineering statistics, statistical modeling, and data analysis.

FAST CONVOLUTION KERNELS ON PASCAL GPU WITH HIGH MEMORY EFFICIENCY

Qiong Chang

Grad. of Systems and Information Engineering
University of Tsukuba
1-1-1 Ten-nou-dai Tsukuba Japan
cq@darwin.esys.tsukuba.ac.jp

Masaki Onishi

National Institute of Advanced Industrial Science and Technology (AIST)
1-1-1 Umezono Tsukuba Japan
onishi@ni.aist.go.jp

Tsutomu Maruyama

Grad. of Systems and Information Engineering
University of Tsukuba
1-1-1 Ten-nou-dai Tsukuba Japan
maruyama@darwin.esys.tsukuba.ac.jp

ABSTRACT

The convolution computation is widely used in many fields, especially in CNNs. Because of the rapid growth of the training data in CNNs, GPUs have been used for their acceleration and memory-efficient algorithms have been the focus of attention due to their high performance. In this paper, we propose two convolution kernels for single-channel and multi-channel convolution respectively. Our two methods achieve high performance by hiding the access delay of the global memory efficiently, and achieving high ratio of floating point Fused Multiply-Add operations per fetched data from the global memory. In comparison to the latest Cudnn library developed by Nvidia aimed to accelerate the deep-learning computation, the average performance improvement of our research is 2.6X for the single-channel, and 1.4X for the multi-channel.

Keywords: Convolution, GEMM, GPU

1 INTRODUCTION

Convolution is widely used as a fundamental operation in many applications such as computer vision, natural language processing, signal processing. Especially, the Convolution Neural Network (CNN), a popular model used for deep-learning, is widely used in many applications such as image recognition (Poznanski and Wolf 2016), (Simonyan and Zisserman 2014), video analysis (Yu et al. 2016), natural language processing (Zhang and Wallace 2017), and has yielded remarkable results.

Recently, many CNN models have been proposed, such as AlexNet (Krizhevsky et al. 2017), GoogleNet (Szegedy et al. 2015), VGGNet (Simonyan and Zisserman 2014), ResNet (He et al. 2016), etc. They are used in many areas, and are improved steadily. The sizes of their networks have grown larger, which has led to the increase of their processing time. The CNNs have several layers, but a large portion of the total processing time is used for the convolution layers. Because of the high inherent parallelism of the convolution algorithms, many researchers are exploring to use GPUs to accelerate them. They can be divided into four categories: 1) *Direct-based* method (Li et al. 2015), 2) *FFT-based* method (Mathieu et al. 2013), 3) *Winograd-based* method (Lavin and Gray 2016) and 4) *General matrix multiplication (GEMM) based* method (Tan et al. 2011). Recently, many of the algorithms and their modified versions have been aggregated into a public library called Cudnn by Nvidia, which aims to accelerate the deep-learning platforms like Chainer (Tokui et al. 2015), Caffe (Jia et al. 2014), etc. As the volume of the processing data used in deep-learning increases, the memory-efficient algorithms play an increasingly more important role, resulting in the constant proposal of many improved versions. Among them, *Implicit-GEMM* (Chetlur et al. 2014) has been included in the Cudnn because of its high efficiency. It divides the feature map and filter data into many sub-blocks and converts them to many sub-matrices only using on-chip memory of GPU. This method is very memory-efficient, and achieved a high ratio of floating point Fused Multiply-Add (FMA) operations per data transferred from the global memory. In (Chen et al. 2017), two memory-efficient methods were proposed. Both of which are faster than the Implicit-GEMM. However, their performances are negatively affected when the feature map size is smaller than 32, because it fixes the amount of the data assigned to each *SM*, which sometimes is not suitable to the small feature map. In modern CNN models as described above, more than half of the convolution layers are used for the calculation of the images smaller than 32 (such as 28, 14, 7), meaning that (Chen et al. 2017) cannot handle the modern CNN models efficiently.

In this paper, we propose two methods to accelerate the convolution kernels for single-channel and multi-channel. In these methods, the memory access is optimized to achieve higher performance. For the single-channel kernel, our approach follows the computation method proposed in (Chen et al. 2017), however, the data is divided and assigned to each *SM* carefully to hide the access delay of the global memory considering the input data size and the hardware features of our target GPUs (Pascal GPUs). For the multi-channel kernel, we propose a *stride-fixed block* method. This method aims to maximize the number of FMA operations per loaded data because the total amount of data that have to be loaded to each *SM* is much larger than the single-channel convolution. This allows the access delay to be hidden by *data prefetching*.

2 OPTIMIZATION METHODS

In this section, we first introduce the CNN models, and then discuss what kind of optimization method is applicable on Pascal GPUs.

2.1 The Convolution Models

The multi-channel convolution can be defined as follows:

$$\begin{aligned} O^m(x,y) &= \sum_{ch=1}^{C}\sum_{i=0}^{K-1}\sum_{j=0}^{K-1} I^{ch}(x+i,y+j)\cdot F^{ch,m}(i,j), \\ \textit{where} \quad & x\in[0,W_x-K+1), y\in[0,W_y-K+1), m\in[1,M]. \end{aligned} \tag{1}$$

Here, I is the input feature map set and F is the input filter set. O is the output feature map set which is generated from I and F. x and y are the coordinates of the pixels of the feature maps. W_x is the width and W_y is the height of the input feature map. i and j are the offsets, and are added to the coordinates. Their upper bound K determines the size of filter. ch represents the channel of the input in the range of $[1,C]$ (C is the number of channels), and all of the convolution results are added along the dimension ch. m represents the filter number, and each filter has C channels. M is the number of filters, which is defined in each convolution layer. When $C=1$, it is called single-channel convolution, and its definition is given the following equation.

$$O^m(x,y) = \sum_{i=0}^{K-1}\sum_{j=0}^{K-1} I(x+i,y+j)\cdot F^m(i,j) \tag{2}$$

2.2 Acceleration models on GPU

Here, we discuss the acceleration methods of the convolution calculation. However, this discussion is not restricted to the convolution, and can be applied to other applications.

In GPUs, the on-chip memory including registers and shared memory is supported, as well as the off-chip memory, in which the global memory is mainly supported. The register is the fastest, while the global memory is the slowest and largest. To fully utilize this hierarchy, many studies such as (Chen et al. 2017) have been proposed. However, throughput computation, the data loading time from the global memory to the on-chip memory is most critical, and hiding the latency of the global memory is the most crucial point for the acceleration.

To hide the latency of the global memory, two methods can be considered:

1. keep the operation units busy (mainly Fused Multiply-Add (FMA) operation units in convolution) by executing more than N_{FMA} operations (the lowest value to make the units busy) in each *SM* for the current data set until the next data set arrive from the global memory by data prefetching, and
2. transfer a large volume of data (V_s) from the global memory continuously.

In most cases, the first approach is preferable, because the data loading overhead from the global memory can be relatively reduced more by executing more operations per loaded data. In the multi-channel convolution, the data size is large enough, and it is possible to find the division of the feature maps and filters that makes it possible to execute more than N_{FMA} operations in each *SM*. However, in the single-channel convolution, when the size of feature maps is small, the number of executable operations becomes less than N_{FMA} even with the data prefetching, and the second approach is required. Thus, it is necessary to make it clear under what conditions which method shows better performance.

Table 1 shows several parameters of GTX 1080Ti and its performance for accessing single precision data. As shown in Table 1, in GTX 1080Ti, 2 FMA operations can be executed in one clock cycle in each core, namely 256 FMA operations in each *SM* (each *SM* has N_{cores} = 128 cores). According to the method proposed in (Mei and Chu 2017), the global memory latency of the GTX 1080Ti is 258 clock cycles. In order to hide these 258 clock cycles, $N_{FMA} = 66,048$ FMA operations ($66,048 = 258 \times N_{cores} \times 2$) are required in each *SM* for the current data set, the set of divided feature maps and filters.

The volume size V_s can be calculated as follows. The Geforce GTX 1080Ti has a base clock of 1480 MHz and the bandwidth of 484 GB/s, which means the transfer rate is roughly 327 bytes per clock cycle. Therefore, the volume size needed to hide the latency (258 clock cycles) becomes 84,366 = 327 $\times$ 258 bytes. To accommodate the data transfer of this size, 21,120 (= 84,366 / 4) threads are required because each thread fetches a 4 byte data in single precision. Thus, in each of 28 *SMs* ($N_{sm} = 28$ is the total number of *SMs* in the GTX 1080Ti), 755 threads are required to fetch one 4-byte word respectively. Here, because 32 threads in a warp execute the same instruction at the same time, the number of threads required is often kept at an integer multiple of 32. Therefore, instead of 755, 768 threads are preferred (in total, it becomes 768 $\times 4 \times$ 28= 86,016 > 84,366). This means that the minimum volume size to make the global memory busy is V_s = 86,016 bytes. For dividing the feature maps and filters, and assigning them to each *SM*, the following procedure should be taken:

1. Divide the feature maps and filters so the total size of data that are assigned to each *SM* is smaller than the size of the shared memory S_{shared} (96KB in GTX 1080Ti).
2. Evaluate the number of FMA operations that can be executed for the data in each *SM*.
3. If it is larger than N_{FMA}, use the first method which is based on the data prefetching.
4. If not, redivide the feature maps and filters so that the total size of data that are transferred to all *SMs* become larger than V_s, and use the second approach.

Table 1: Parameters to access single precision data

	GTX 1080Ti
Architecture	Pascal
Global Memory Latency (clock cycles)	258
Bandwidth (GB/s)	484
Base clock cycle (MHz)	1480
SM	28
Transmission Rate (Byte/clock cycle)	327
Data Requirement (bytes)	84,366
Thread Requirement/SM	768
Warp Requirement/SM	24
Data Requirement/SM (bytes)	3072
Flops/clock cycle/core	2

Additionally, to access global memory, it is necessary to confirm that the starting address and the size of the sequential accessing segment is a multiple of *32-byte*. In Pascal GPU, although the performance of multiple of *128-byte* shows best (NVIDIA 2018), the performance for *32-byte* and *64-byte* are also acceptable.

2.3 Data Mapping

As shown in Fig.1(a), in the single-channel convolution ($C = 1$), the size of each filter is $K \times K \times 4$-*byte*, and they are continuously stored in the global memory. With this data mapping, the filters can be divided only along the dimension *m*, and the filters can be efficiently loaded from the global memory because they are stored continuously. Three approaches can be considered.

1. Only the filters are divided and are assigned to each *SM*. In each *SM*, the assigned filters are applied to the whole feature maps (the feature map is processed sequentially against each filter).
2. Only the feature maps are divided and are assigned to each *SM*. In each *SM*, the assigned feature maps are processed by all filters sequentially.
3. Both feature maps and filters are divided, and they are assigned to each *SM* (the combination of the first and the second approach).

By using different approach, the amount of data that has to be loaded to the shared memory from the global memory, and the number of FMA operations that can be executed in parallel become different. Therefore, finding a good balance between the size of divided feature maps and filters becomes a key point.

In the multi-channel convolution ($C > 1$), which is the typical case in the convolution layers of the CNN. The data size becomes much larger than the single-channel convolution. Fig.1(b) shows how the filters are stored in the global memory. They are stored along the dimension *ch* first, and then along the dimension *m*. In this case, the dividing method of the filters along the dimension *m* used for the single-channel convolution cannot be applied as is, because the data size of each filter is normally not a multiple of *32-byte*. Especially when $K = 1$, the filter size is only 4 bytes and are accessed as *4-byte* segments which causes serious performance reduction due to the *non-coalescing memory access*.To solve this problem, several approaches can be considered. Fig.2(a) shows the whole data structure before the division.

1. In Fig.2(b), both the filters and the feature maps are divided along the dimension *ch*, and the data for $C' = C/N_{sm}$ channels are assigned to each *SM* (N_{sm} is the total number of *SMs*). With this

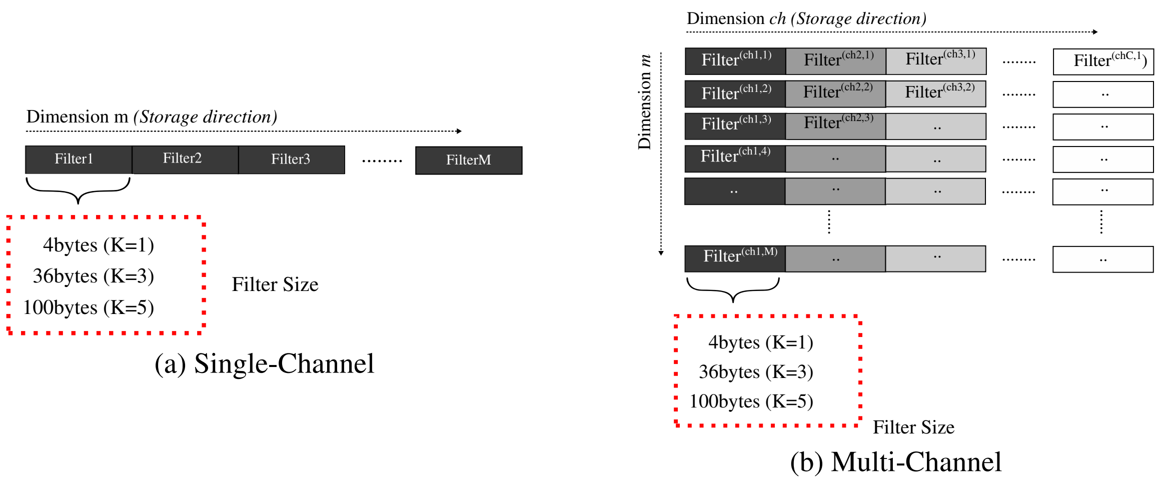


Figure 1: Memory storage form of the Filter

division, the data calculated in each thread have to be summed along dimension *ch*, meaning that add operations among *SMs* are required, and $W_x \times W_y \times C' \times 4\text{-}byte$ byte in the global memory are used for this summation. The global memory accesses to this area and the synchronous operations required for this summation considerably reduce the overall performance.

2. In Fig.2(c), only the filters are divided along the dimension *m*. $M' \times C = M/N_{sm} \times C$ filters are assigned to each *SM*, and the whole feature map is loaded to *SMs* from the global memory. In this approach, if the total size of the filters are less than the total size of shared memory ($K \times K \times C \times M \times 4\text{-}byte < N_{sm} \times S_{shared}$), the divided filters can be cached in each *SM*, and no additional access to the global memory is required.
3. On the other hand, in Fig.2(d), only the feature maps are divided along dimension *y*. The divided feature maps are assigned to each *SM*, and the whole of the filters are loaded to *SMs* from the global memory. In this case, if the total size of the feature maps is smaller than the total size of the shared memory, the divided feature maps can be cached in each *SM*, and no additional access to the global memory is required.
4. However, the total size of the filters and feature maps are larger than the total size of the shared memory in general. Thus, as shown in Fig.2(e), both the filters and feature maps have to be divided respectively, and each divided segments are cached in each *SM* or loaded from the global memory. In this case, there are many alternatives for how to divide the filters and features maps.

According to our preliminary evaluation, the performance with the data dividing method along the dimension *ch* (Fig.2(b)) is obviously slower than other dividing methods because of the additional access to the global memory for the addition. For achieving higher performance, it is necessary to choose other dividing methods considering the data size and the hardware features of the target GPUs so that in each *SM* the number of FMA operations that can be executed per loaded data from the global memory is maximized.

3 GPU IMPLEMENTATION

According to the discussion in Section 2, in both the single-channel and multi-channel convolution, it is important to make the number of FMA operations per pre-fetched data higher than N_{FMA} in order to improve performance. However, in some single-channel convolution cases, like when the size of feature maps is small, the number of FMA operations cannot be kept high enough by data prefetching. This means that, in case of single channel convolution, according to the size of input data, we need to choose one of the two methods described in Section 2.2: data prefetching or data transfer larger than V_s.

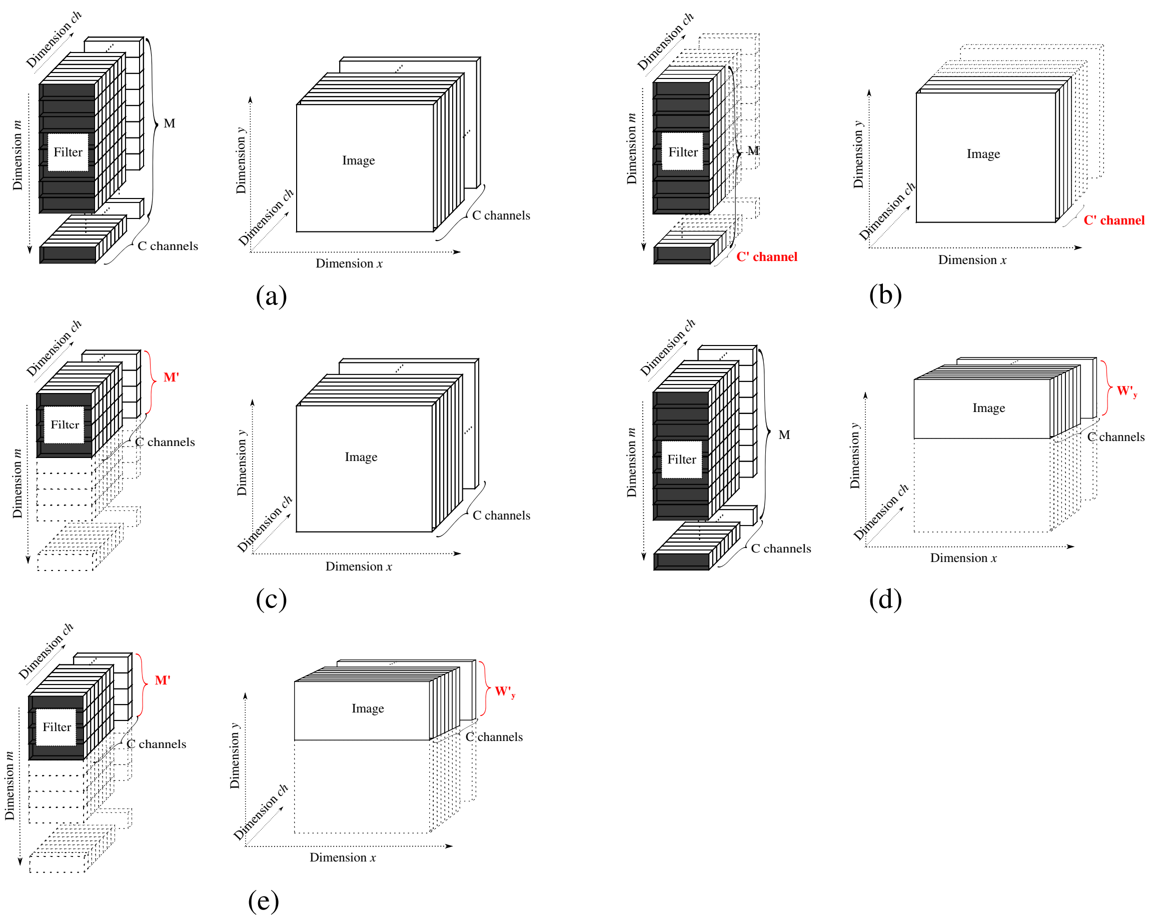


Figure 2: Assignment of the input data

In the multi-channel convolution, the size of input data is large enough, so the number of FMA operations can be kept high enough by data prefetching. However, the performance can be improved more by achieving higher FMA operation ratio for the fetched data, because to fetch the data from the global memory, each thread has to issue the instruction to read the data, and the clock cycles are spent for issuing these read instructions. Therefore, in the multi-channel convolution, to find the data dividing method that maximizes the number of FMA operations for each divided data is the key to achieve higher performance.

3.1 Single-Channel Convolution

Here, we describe how to divide the input data to improve the performance in the single channel convolution. As for the convolution calculation in each *SM*, we follow the method proposed in (Chen et al. 2017).

The total amount of the input data is given by:

$$D_{input} = D_{filter} + D_{map} = (K \times K \times M + W_x \times W_y) \times 4\,Bytes. \tag{3}$$

Let N_{sm} be the number of *SMs*. There are two ways to divide the input data and assign them to each *SM*. In the first method, the input data is divided along the dimension m of the filter. D_1, the size of input data assigned to each *SM*, becomes

$$D_1 = \frac{D_{filter}}{N_{sm}} + D_{map} = (K \times K \times \lceil \frac{M}{N_{sm}} \rceil + W_x \times W_y) \times 4\,Bytes \tag{4}$$

In general, D_{map} is too large to be stored in the on-chip memory of each SM. Thus, D_{map} is divided into P pieces along the dimension y. The size of each piece becomes $D_{Inc1} = D_{map}/P$. Here, for each line of feature map, since the convolution requires additional $K-1$ lines, the amount of data that have to be held in the on-chip memory becomes

$$D_1 = \frac{D_{filter}}{N_{sm}} + D_{Inc1} + (K-1) * W_x = (K \times K \times \lceil \frac{M}{N_{sm}} \rceil + (\lceil \frac{W_y}{P} \rceil + K - 1) \times W_x) \times 4 Bytes \tag{5}$$

and the number of FMA operation that can be executed for these data in each SM is given by

$$Th_1 = \frac{D_{filter}}{N_{sm}} \times D_{Inc1} = K \times K \times \lceil \frac{M}{N_{sm}} \rceil \times \lceil \frac{W_y}{P} \rceil \times W_x. \tag{6}$$

In the second method, the input data is divided along the dimension y of the feature map. In this case, D_2, the amount of the input data assigned to each SM, becomes

$$D_2 = D_{filter} + \frac{D_{map}}{N_{sm}} = (K \times K \times M + (\lceil \frac{W_y}{N_{sm}} \rceil + K - 1) \times W_x) \times 4 Bytes. \tag{7}$$

D_{filter} is too large to be stored in the on-chip memory in general, and it is divided into Q pieces. The size of each piece becomes $D_{Inc2} = D_{filter}/Q$. Then, D_2 becomes

$$D_2 = D_{Inc2} + \frac{D_{map}}{N_{sm}} = \frac{D_{filter}}{Q} + \frac{D_{map}}{N_{sm}} = (K \times K \times \lceil \frac{M}{Q} \rceil + (\lceil \frac{W_y}{N_{sm}} \rceil + K - 1) \times W_x) \times 4 Bytes \tag{8}$$

and the number of FMA operation that can be executed for these data in each SM is given by

$$Th_2 = D_{Inc2} \times \frac{D_{map}}{N_{sm}} = K \times K \times \lceil \frac{M}{Q} \rceil \times (\lceil \frac{W_y}{N_{sm}} \rceil) \times W_x. \tag{9}$$

The values of P and Q are decided considering if D_1 or D_2 is smaller than S_{shared}, and if Th_1 or Th_2 is larger than N_{FMA}. If $P = 1$ or $Q = 1$, the feature maps or the filters are not divided, and they are transferred to the on-chip memory at a time. If $P > 1$ or $Q > 1$, the feature maps or the filters are divided into several pieces, and the pieces are transferred to each SM by using the data prefetching. With smaller P and Q, D_1, D_2 and Th_1, Th_2 become larger. The lower bound of P and Q is given by the requirement that D_1 and D_2 have to be smaller than S_{shared}, and the upper bound is given by the requirement that Th_1 and Th_2 should be larger than N_{FMA}. P and Q should be chosen so that these requirements can be satisfied. In our implementation, P and Q are decided as follows.

1. Th_1 or Th_2 should be larger than the number of FMA operation N_{FMA}.

   $Th_1 \geq N_{FMA}$ and $Th_2 \geq N_{FMA}$

   Thus, the upper bound of P and Q (they must be smaller than W_y and M respectively) is given as follows

   $$P \leq \frac{K \times K \times \lceil \frac{M}{N_{sm}} \rceil \times W_y \times W_x}{N_{FMA}} and P \leq W_y, \qquad Q \leq \frac{K \times K \times M \times \lceil \frac{W_y}{N_{sm}} \rceil \times W_x}{N_{FMA}} and Q \leq M$$

2. D_1 and D_2 must be smaller than the size of on-chip memory. The lower bound of P and Q is given as follows.

   $$P \geq \frac{4 \times W_y \times W_x}{S_{shared} - 4 \times K \times K \times \lceil \frac{M}{N_{SM}} \rceil + (1-K) \times 4 \times W_x}, \qquad Q \geq \frac{4 \times M \times K \times K}{S_{shared} - 4 \times W_x \times (\lceil \frac{W_y}{N_{SM}} + K - 1)}$$

   Actually, there exist one more requirement to decide this lower bound. The number of required registers for the computation must be smaller than that supported in each SM. Its detail is not shown here, but considering this requirement, the lower bound is calculated.

3. If there exist P and Q (P and Q must be an integer) in the range specified by (1) and (2), any of them can be used. In our current implementation, the minimum ones are chosen as P and Q, because the smaller values means less division, and make the processing sequence simpler. If no value exists, P and Q are set to 1.
4. Using the obtained P and Q, D_1 and D_2 are calculated and compared. If D_1 is smaller than D_2, Q is reset to 1 to use the first dividing method described above, and otherwise, P is reset to 1 to use the second one. Both methods can be used because they both satisfy the requirements, but for safety and leaving more memory space on the on-chip memory, the smaller one is chosen.

Following this procedure, the input data are divided and allocated to each *SM* in the best balance.

3.2 Multi-Channel Convolution

As described above, in the multi-channel convolution, both feature maps and filters are divided, and prefetching is used to transfer them to each *SM* from the global memory. Recent block-based methods show high performance in convolution due to their continuous and simple memory access sequence. Fig.3 shows the data mapping of the filters and feature maps, and how they are divided and calculated in each *SM*. In the block-based method, as shown in Fig.3(a), the following data are loaded to the on-chip memory in each *SM*.

1. S bytes of each filter along the dimension ch (called *segment* in the following discussion) of M' filters ($S \times M'$ bytes in total), and
2. a part of feature map, $W'_x \times W'_y \times 4\,bytes$ in the same channel (W'_x is an arbitrary value that is decided by the size of on-chip memory, but W'_y is specified as $\lceil \frac{S}{K \times 4bytes} \rceil$, because when S bytes are fetched along the dimension ch, $\lceil \frac{S}{K \times 4bytes} \rceil$ lines in the feature map are required to apply the filter).

Then, the convolution is calculated for these data, and the next data (next $S \times M'$ bytes of filters and $W'_y \times W'_x$ bytes of feature maps are loaded by data prefetching. In (Chen et al. 2017), the filter size is chosen as S ($S = K \times K \times 4\,bytes$), and only the filters of the target channel and a part of feature map of the same *channel* are loaded to the on-chip memory. However, the filter size $K \times K$ is usually odd and often small, and the performance is seriously degraded because of *non-coalescing memory access*. (Tan et al. 2011) tried to solve this problem by extending S to *128-bytes*. By fetching continuous 128 bytes on the global memory, the highest memory throughput can be achieved in GPUs. In this method, the filters of several channels (and a part of the next channel) are fetched at the same time, and are kept in the on-chip memory. First, only the filters of the first channel are used for the computation, then, the filters of the next channel are used. With this larger S, M' has to be kept small because of the limited size of on-chip memory, and smaller M' means less parallelism (M' filters are applied in parallel to the feature map of the same channel). In (Chen et al. 2017), higher parallelism comes first, while in (Tan et al. 2011), lower access delay has a higher priority.

Here, we propose a *stride-fixed block* method not only to maintain the efficient global memory access, but also to achieve high parallelism in each *SM*.

1. S is set to a multiple of 32-bytes. Actually, 32 or 64 is used. Small S allows larger M', namely higher parallelism, under the limited size of on-chip memory. When S is 32 or 64 bytes, the memory throughput from the global memory becomes a bit worse than $S = 128$ bytes (the highest throughput), but it is acceptable. $S = 32$ is the minimum value to maintain efficient global memory access.
2. Next, fixed W'_x. W'_x pixels in the feature map are fetched along dimension x from the global memory. Thus, W'_x should be a multiple of 128-bytes to achieve the highest memory throughput. Larger W'_x is preferable because it increases the Instruction Level Parallelism (ILP), which can improve the performance of the convolution.

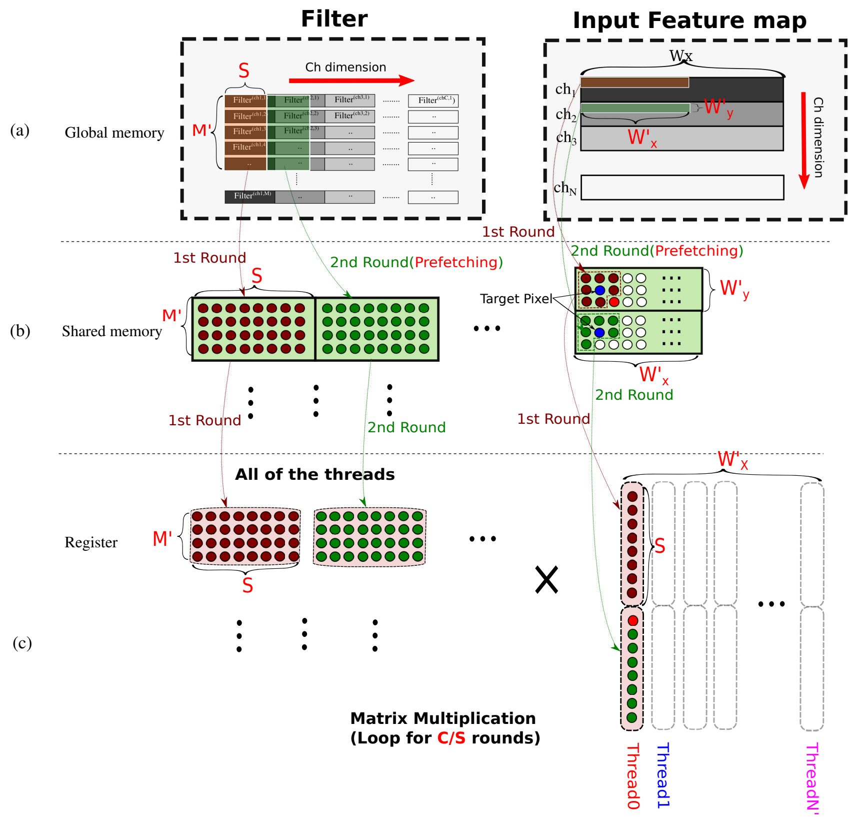


Figure 3: Multi-Channel Convolution Kernel

3. After deciding the values of S and W'_x, the most suitable M' can be found by the requirements of the number of FMA operations.
   $M' \geq \frac{N_{FMA} \times 4\text{-}bytes}{S \times W'_x}$.
4. Because the data prefetching is used to fetch the next data set while the current data set is being used for the current calculation, the size of the data set cannot exceed half of the shared memory. Thus, $(S \times M' + \lceil \frac{S}{K \times 4bytes} \rceil \times W'_x) \leq \frac{S_{shared}}{2}$

   Here, $\lceil \frac{S}{K \times 4bytes} \rceil = W'_y$ is the number of feature maps required for the calculation.

With this approach, for given S, W'_x and M' to improve performance based on *block method* can be obtained.

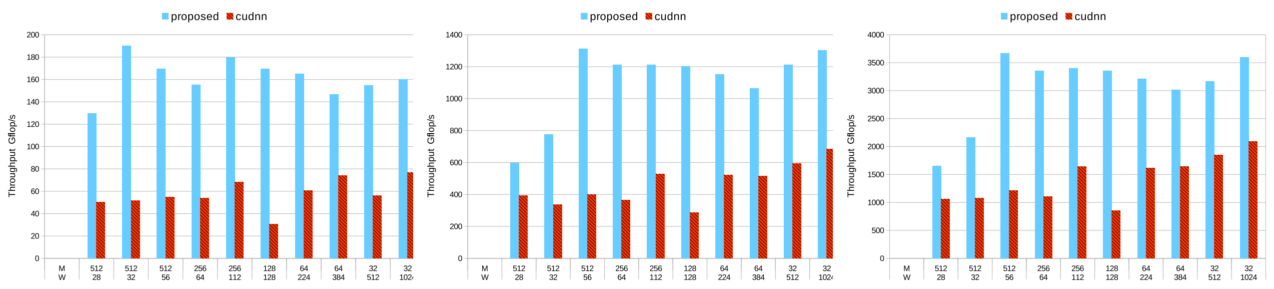
From here, we describe how the convolution calculation is executed in each SM. As shown in Fig.3(a)(b), first, each SM loads $S bytes$ of M' filters to the shared memory. At the same time, W'_x pixels on $\lceil \frac{S}{K \times 4bytes} \rceil = W'_y$ lines of the feature maps are also loaded. After the first round loading of these data, the same size of data for the next round are pre-fetched: the next $S \times M'$ bytes along the dimension ch, and the next W'_x pixels of the W'_y lines. During the second round loading, the convolutions for the first round data set are calculated on the chip as shown in Fig.3(b)(c). On the chip, each thread corresponds to one target pixel of the feature map as shown in Fig.3(b). Because the accessing speed of registers is faster than that of the shared memory, it is required to transfer each data in the shared memory to the registers in order to achieve high performance. In the convolution computation, the target pixel and its neighbors in the feature map are sent to the corresponding registers by each thread. Here, one important point is that only $S/4bytes$ pixels in the feature map have to be loaded onto the registers. The rest pixels, the red ones in Fig.3(b), are just held in the shared memory for the next round. The filter data is also transferred to the registers by the corresponding thread, but in this case, all data is transferred to the registers because all of them are used. After that, each pixel data is multiplied by the corresponding filter data, and their products are added. When all computations for the data stored in on-chip memory has been finished, data prefetching for the third round is started. During this loading, the convolution for the second round data is calculated.

By using this method, the size of S can be kept small, and the number of filters M' can be increased. This ensures that more filters can be applied in parallel to the same feature map. This does not increase the number of data loading of the feature maps, and hides the latency caused by global memory access.

4 EXPERIMENTAL ANALYSIS

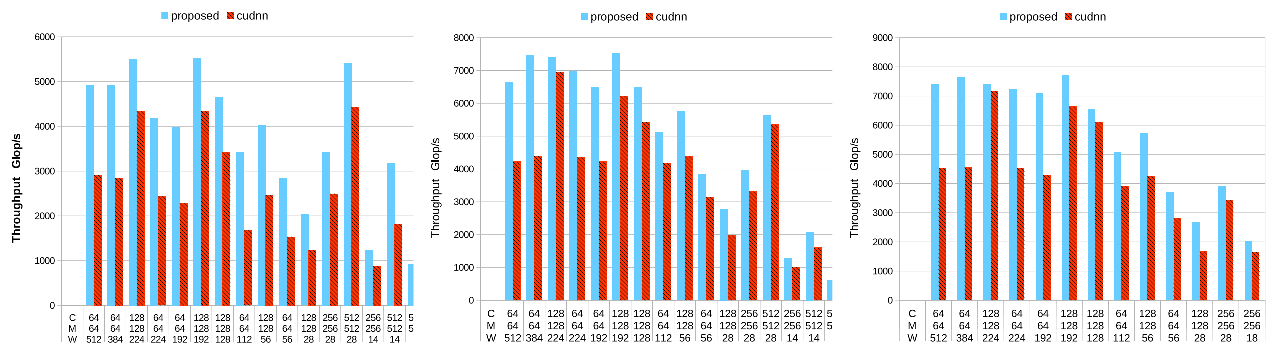
We implemented our two convolution kernels on Pascal series GPU Geforce GTX 1080Ti by using the CUDA 8.0. Their performances were evaluated using many convolutions which are commonly used in popular CNN models, and compared with the latest public library Cudnn v7.1 (Chetlur et al. 2014).

In the single-channel convolution, we changed the sample size of the feature maps from 28 to 1K and the size of the corresponding channels from 512 to 32. The filter size is 1, 3 or 5, which is common in many CNN models. In CUDA programming, by assigning more number of blocks to each SM, the SMs can be kept busy. In our current implementation, $N_{block} = 2 \times N_{SM} = 2 \times 28$ blocks are used. Two blocks are assigned to each SM, and 512 threads are assigned to each block. Thus, the maximum number of registers for each thread is constrained to 128. For each tested case, P and Q are decided following the method



Filter Size = 1 Filter Size = 3 Filter Size = 5

Figure 4: Performance of the Single-Channel Convolution Kernel



Filter Size = 1 Filter Size = 3 Filter Size = 5

Figure 5: Performance of the Multi-Channel Convolution Kernel

described in Section 3.1. Fig.4 shows the results of the single-channel convolution. Our method is faster than Cudnn v7.1 in all tested cases. The performance gain is 1.5X to 5.6X, and its average is 2.6X. In the multi-channel convolution, we changed the sample size of the feature maps from 7 to 512, and the size of the corresponding channels from 64 to 512. The filter size is also 1, 3, or 5. As discussed in Section 3.2, larger M' is preferable for making *data prefetching* more effective. Therefore, we fixed the segment size S as 32 or 64 bytes, and M' and W'_x are decided following the method described in Section 3.2. According to our preliminary evaluation, when $M' = 64$ and $W'_x = 128$, the performance is best, and we used these values for this comparison. As shown in Fig.5, our method is faster than Cudnn in all tested cases, and the throughput has been increased by 1.05X to 2X, with an average increase of 1.39X. In (Chen et al. 2017), a different GPU is used, and a direct comparison is not possible. However, when $K = 3$, our performance is 4X faster and the peak performance of which is 2.4X faster than that used in (Chen et al. 2017).

We also implemented our two kernels on Maxwell series GPU GTX Titan X, and it also showed that our performance is faster than Cudnn on the same GPU by 1.3X to 3.7X in the single-channel convolution and 1.08X to 1.8X in the multi-channel convolution.

5 CONCLUSIONS

In this paper, we proposed two convolution kernels on Pascal series GPUs for single-channel and multi-channel respectively. For single-channel convolution, we introduced an effective method of data mapping, which can hide the access delay of the global memory efficiently. For multi-channel convolution, we introduced a method that not only guarantees the memory access efficiency, but also achieves high FMA operation ratio per loaded data. Performance comparison with the public library Cudnn shows that our approaches are faster in all tested cases: 1.5X to 5.5X in the single-channel convolution and 1.05X to 2X in the multi-channel convolution. Our approaches was designed assuming Pascal architecture, but the performance is also faster than Cudnn on Maxwell architecture. This practice shows that our approaches can be applied to the wide range of CNN models on various GPUs. In our current implementation, the throughput is still lower than the theoretical maximum, meaning that the convolution kernel still has room for improvement, which will be our main future work.

ACKNOWLEDGMENTS

This paper is based on results obtained from a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

REFERENCES

Chen, X., J. Chen, D. Z. Chen, and X. S. Hu. 2017. “Optimizing Memory Efficiency for Convolution Kernels on Kepler GPUs”. In *54th Annual Design Automation Conference, DAC*, pp. 68:1–68:6.

Chetlur, S., C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. 2014. “cuDNN: Efficient Primitives for Deep Learning”. *CoRR* vol. abs/1410.0759.

He, K., X. Zhang, S. Ren, and J. Sun. 2016. “Deep Residual Learning for Image Recognition”. In *IEEE Conference on Computer Vision and Pattern Recognition,CVPR*, pp. 770–778.

Jia, Y., E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. B. Girshick, S. Guadarrama, and T. Darrell. 2014. “Caffe: Convolutional Architecture for Fast Feature Embedding”. In *ACM International Conference on Multimedia, MM*, pp. 675–678.

Krizhevsky, A., I. Sutskever, and G. E. Hinton. 2017. “ImageNet classification with deep convolutional neural networks”. *Commun. ACM* vol. 60 (6), pp. 84–90.

Lavin, A., and S. Gray. 2016. “Fast Algorithms for Convolutional Neural Networks”. In *IEEE Conference on Computer Vision and Pattern Recognition,CVPR*, pp. 4013–4021.

Li, S., Y. Zhang, C. Xiang, and L. Shi. 2015. “Fast Convolution Operations on Many-Core Architectures”. In *17th IEEE International Conference on High Performance Computing and Communications, HPCC*, pp. 316–323.

Mathieu, M., M. Henaff, and Y. LeCun. 2013. “Fast Training of Convolutional Networks through FFTs”. *CoRR* vol. abs/1312.5851.

Mei, X., and X. Chu. 2017. “Dissecting GPU Memory Hierarchy Through Microbenchmarking”. *IEEE Trans. Parallel Distrib. Syst.* vol. 28 (1), pp. 72–86.

NVIDIA 2018. “CUDA C Programming Guide V9.1”. Http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

Poznanski, A., and L. Wolf. 2016. “CNN-N-Gram for HandwritingWord Recognition”. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pp. 2305–2314.

Simonyan, K., and A. Zisserman. 2014. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. *CoRR* vol. abs/1409.1556.

Szegedy, C., W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. 2015. “Going deeper with convolutions”. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pp. 1–9.

Tan, G., L. Li, S. Triechle, E. H. Phillips, Y. Bao, and N. Sun. 2011. “Fast implementation of DGEMM on Fermi GPU”. In *Conference on High Performance Computing Networking, Storage and Analysis, SC*, pp. 35:1–35:11.

Tokui, S., K. Oono, S. Hido, and J. Clayton. 2015. “Chainer: a Next-Generation Open Source Framework for Deep Learning”. In *Workshop on Machine Learning Systems (LearningSys), NIPS*.

Yu, D., W. Xiong, J. Droppo, A. Stolcke, G. Ye, J. Li, and G. Zweig. 2016. “Deep Convolutional Neural Networks with Layer-Wise Context Expansion and Attention”. In *Interspeech 2016, 17th Annual Conference of the International Speech Communication Association*, pp. 17–21.

Zhang, Y., and B. C. Wallace. 2017. “A Sensitivity Analysis of (and Practitioners’ Guide to) Convolutional Neural Networks for Sentence Classification”. In *8th International Joint Conference on Natural Language Processing, Volume 1: Long Papers*, pp. 253–263.

AUTHOR BIOGRAPHIES

QIONG CHANG received his Master degree in Engineering from the University of Tsukuba in 2013. He is currently a PhD student at the Graduate School of Systems and Information Engineering, University of Tsukuba. His research interest is in reconfigurable parallel computing systems. His email address is cq@darwin.esys.tsukuba.ac.jp.

MASAKI ONISHI recevied his PhD from the Osaka Prefecture University in 2002. He was a research scientist at the National Institute of Advanced Industrial Science and Tehnology (AIST). His research interests are computer vision, video surveillance, and human robot interaction. His email address is onishi@ni.aist.go.jp.

TSUTOMU MARUYAMA received his PhD in Engineering from the University of Tokyo in 1987. He is currently a professor at the Graduate School of Systems and Information Engineering, University of Tsukuba. His research interest is in reconfigurable parallel computing systems. His email address is maruyama@darwin.esys.tsukuba.ac.jp.

ON TUNING THE SYMMETRIC SPARSE MATRIX VECTOR MULTIPLICATION WITH CSR AND TJDS

Edward Aymerich

Department of Computer Science
University of Central Florida
4000 Central Florida Boulevard
Orlando, FL, USA
edward.aymerich@ucf.edu

Alexandre Duchateau*

Pure Storage
650 Castro St #400
Mountain View, CA, USA
lex@purestorage.com

Euripides Montagne**

Department of Computer Science
University of Central Florida
4000 Central Florida Boulevard
Orlando, FL, USA
eurip@ucf.edu

Frank Plochan

Department of Computer Science
University of Central Florida
4000 Central Florida Boulevard
Orlando, FL, USA
researcherfp@gmail.com

ABSTRACT

In this work we present a heuristic to select the appropriate compressed storage format when computing the symmetric SpMV multiplication sequentially. A subset of symmetric sparse matrices were selected from the SPARSITY benchmark suite and extended with other matrices we consider complement them. All matrices were collected from Matrix Market and UF matrix collection. Experimental evidence shows that given a symmetric sparse matrix, predicting what is the more convenient format to use for computing the symmetric SpMV multiplication could be possible. According to our findings, and good rule of thumb, if the average number of non zero coefficients per column (row) is less than 3.5, then the symmetric SpMV multiplication runs up to 1.6× faster using the TJDS format compared to CSR.

Keywords: : Storage format, CSR, TJDS, data structures; sparse matrix vector product, symmetric sparse matrix.

1 INTRODUCTION

When solving a sparse system of linear equations using iterative methods, most of the computation time is dominated by the sparse matrix vector product (SpMV) and this has brought up a significant amount of research into sparse matrix reorganization and mapping the SpMV onto sequential, parallel and distributed systems. Research into sparse matrix representation has developed various static

* Formerly at University of Illinois at Urbana-Champaign

** Most of this author contribution was carried out while visiting University of Illinois at Urbana-Champaign storage reduction schemes such as Compressed Sparse Row (CSR), Compressed Column Storage (CCS), Jagged Diagonal Storage (JDS), Transpose Jagged Diagonal Storage format (TJDS), and Block Compressed Sparse Row format (BCSR) (Dongarra 2000, Montagne and Ekambaran 2004, Saad 1989, Saad 1996, Pinar and Heath 1999) among others. Compressed storage formats focus on avoiding direct representation of zero elements. They achieve that by only storing non zero elements in a linear array, and rely on additional arrays of meta data to infer where they sit in the original sparse matrix. The quantity and shape of meta data varies with each storage format. While those storage formats achieve their primary objective in lowering the memory footprint of sparse matrices, they introduce their own performance challenges for SpMV due to the complexity of the inner loop and indirections in memory access. For example, with BCSR, there is a tradeoff between reducing indirect addressing and the presence of some zeros inside the blocks when these are not dense.

The rest of this paper is organized as follows: Section 2 presents a review of the related works. Section 3, introduces symmetric matrices and presents the benchmark matrix suite. Section 4 describes the algorithm and data structure needed to store symmetric sparse matrices using CSR and TJDS. Section 5 shows the performance of the symmetric SpMV multiplication using CRS and TJDS and the type of matrices that benefit from each format. Finally, in Section 6 we give some final remarks.

2 RELATED WORKS

Several formatting options have been reported to compress sparse matrices to gather the non zero values into a data structure that allows us to run the SpMV product computation efficiently without handling the zero values. Compressed Row Storage (CSR) is a widely used format and a mandatory reference. In addition, many optimizations techniques have been developed for it, in particular regarding SpMV performance (Im et al. 2004, Vuduc. 2004,Williams et al. 2009). Unfortunately, due to the variety of non zero elements distribution in sparse matrices, this format is not always the most successful. On this regard, Sedaghati (Sedaghati et al. 2015) presents an interesting decision model using machine learning to predict the best format to compute SpMV, depending on some matrix features and using three different high-end GPUs. They showed that prediction is possible.

A variant of CSR, known as the Block Compressed Sparse Row format (BCSR) can be considered an improvement on CSR because it reduces the number of indirect accesses and therefore it has received special attention (Pinar and Heath 1999, Lee et al. 2003, Vuduc. 2004). Pinar and Heath proposed a reordering heuristic based on the Traveling Salesman Problem combined with the Block Compressed Row Storage format (BCSR). Their method is aimed to reduce the number of memory accesses by permuting rows and columns to create dense blocks and/or enlarge dense blocks already present. As a result, they reduce the number of indirections from the number of non-zero values to one indirection per block (Pinar and Heath 1999). Another method to decrease memory traffic is register blocking optimization combined with a blocking scheme where explicit zeros are added to the blocks to create dense tiles (Im et al. 2004). In this approach, they eliminate loads and store operations by reusing values already stored in registers. In SPARSITY, they report speedups up to a maximum of $4\times$ compared to the baseline CSR.

Some times, a specific distribution of values in a matrix may not benefit from a straightforward application of a specific storage format and some reordering is necessary. In this regard, Cuthill and Mckee (Cuthill and McKee 1969) present a reordering algorithm to reduce the bandwidth of a sparse matrix. Basically, they represent the sparse matrix non zero coefficients as the vertices of a graph and through a series of rows and columns permutations, they relabel the vertices of the graph. The relabeled graph represents a matrix where the number of the non zero elements cluster near the diagonal. When using this approach, a permutation vector must be saved to permute the source vector x before the computation of the SpMV multiplication begins, and another permutation must be done in the resulting vector to obtain the final result. This means that if Cuthill-McKee is applied, then independently of the compressed storage format, we must compute $P_2 y = A_{cm} P_1 x$ instead of $y = Ax$, where P_1 and P_2 are permutation arrays and A_{cm} is the Cuthill-Mckee reduced band matrix. The TJDS format we are presenting in this work also requires a permutation array but even with the permutation array, TJDS uses less space than CSR.

Vuduc proposed the row segmented diagonals (RSDIAG) format as an alternative to exploit matrices with irregular diagonal structures such as diagonals that grow progressively longer (Vuduc. 2004). In these cases BCSR combined with uniform register blocking is difficult to apply. The RSDIAG method basically consists of dividing the matrix into row segments and selecting a tuning parameter called the unrolling depth u. Then in each segment we move along the diagonals selecting u elements per diagonal and copying them back to back into a linear array. Using RSDIAG, Vuduc reported speedups of $2\times$ for matrices with irregular diagonal structure.

The variable band storage proposed by Jennings (Jennings 1966) also known as profile storage or Skyline is a more flexible storage scheme than diagonal storage and works well for symmetric sparse matrices. It has the advantage of using only two vectors: one for the matrix coefficients and another to identify the beginning of each row. The column index array is not necessary because the column indices can be easily derived (Duff 1977). TJDS resembles Jennings' format when dealing with banded systems but there are two differences: 1) The variable band storage allows fill in zeros in between the diagonal coefficients and the last nonzero coefficient in each row while TJDS does not. 2) Because TJDS does not allow fill in zeros the index array is required. However, in the handling of banded matrices, TJDS is able to easily derive the indices as well.

In solving the SpMV kernel using parallel and distributed systems, a generous amount of study has addressed the problem of mapping the SpMV kernel into different types of architectures in order to optimize its performance. But recently some attention has been focused on mapping Symmetric SpMV on multicore architecture because there is a tendency to adopt multicore processors as de facto standard for parallel processing (Krotkiewski and DabrowskiArrow 2010, Bulu et al. 2011). Mapping the symmetric SpMV multiplication onto a multicore architecture is not an easy task and we currently have an investigation on the possibilities of using TJDS in this arena. However, in this work we are reporting how TJDS is an adequate choice to compute the sequential symmetric SpMV multiplication for a class of matrices.

3 SYMMETRIC MATRICES

Probably the type of symmetric sparse matrices we are most acquainted with is the symmetric definite band matrix. These are matrices whose elements outside a band along the main diagonal are all zeros. We can characterize them by an integer $m < n$ (Schwarz 1973) such that:

$$a_{ij} = 0 \;\; \forall i, j \; with \;\; |i - j| > m \tag{1}$$

in equation (1) n is the matrix dimension and m defines the bandwidth as $b = 2m + 1$. For instance for a given matrix A, $m = 0$ tells us that A is a diagonal matrix, $m = 1$ means that matrix A is tridiagonal, $m = 2$ says that A is a band matrix that consist of five adjacent diagonals, and so forth. However, there are many other types of symmetric sparse matrices that range from a diagonal matrix as defined above to any combination made of a diagonal matrix and some off diagonals nonzero values distributed randomly.

One of the advantages of working with symmetric sparse matrices is that only half of the matrix has to be stored alleviating thus the pressure on memory traffic because each matrix coefficient is accessed once and

Table 1: Matrix benchmark suite from Matrix Market.

	Name *(Application Area)*	**Dim. ($\frac{nnz}{dim}$)**	**Nnz (nnz stored in file)**
1	dense *(symmetric dense matrix)*	1000 (500.5)	1000000 (500500)
2	3dtube *(3-D pressure tube)*	45330 (35.95)	3213618 (1629474)
3	crystk03 *(FEM crystal free vibration)*	24696 (35.95)	1751178 (887937)
4	crystk02 *(FEM crystal free vibration)*	13965 (35.18)	968583 (491274)
5	gupta1 *(Linear programming)*	31802 (34.53)	2164210 (1098006)
6	raefsky4 *(Bucking problem)*	19779 (34.09)	1328611 (674195)
7	ct20stif *(CT20 engine block)*	52329 (26.28)	2698463 (1375396)
8	bcsstk35 *(Stiff matrix automobile frame)*	30237 (24.48)	1450163 (740200)
9	vibrobox *(Structure of vibroacustic problem)*	12328 (14.40)	342828 (177578)
10	pwt *(INDEED Test Matrix [DC-m]))*	36519 (4.96)	326107 (181313)
11	finan512 *(Finantial porfolio optimization)*	74752 (4.49)	596992 (335872)
12	saylr4 *(Oil reservoir modelling)*	3564 (3.63)	22316 (12940)
13	penta_diagonal *(Banded)*	1000 (2.99)	4994 (2997)
14	bcspwr10 *(Representation US power network)*	5300 (2.56)	21842 (13571)
15	shallow_water2 *(Fluid dynamics problem)*	81920 (2.50)	327680 (204800)
16	LFAT5000 *(Model reduction problem)*	19994 (2.50)	79966 (49980)
17	1138_bus *(Power network problem)*	1138 (2.28)	4054 (2596)
18	494_bus *(Power network problem)*	494 (2.19)	1666 (1080)
19	largetridiagonal *(Banded)*	10180 (1.99)	30538 (20359)
20	Chem97ZtZ *(Statistical/mathematical problem)*	2541 (1.95)	7361 (4951)
21	bcsstm39 *(Structural problem)*	46772 (1.00)	46772 (46772)

used twice. An additional saving is obtained by storing only the matrix nonzero elements using a compressed storage format.

The work of Lee et al. (Lee et al. 2003, Lee et al. 2004) is, to our knowledge, the most comprehensive and in depth study on optimization strategies to speed up the symmetric SpMV product. In their work, they selected a subset of the SPARSITY benchmark suite and applied the following optimizations: symmetric storage, register blocking, and vector blocking. Their experiments were carried out in four different architectures and for each architecture they reported nine implementations and bounds. The performance summary, they present, is divided in three groups or bands: dense matrix, matrices arising from FEM applications , and matrices from other applications. In the case of dense matrices a performance upper bound was found when they reported gains up to 4$\times$ compared to the baseline implementation. Indeed, dense matrices adapt themselves to any register blocking size and padding zeros are not necessary therefore good performance outcomes can be expected. They also reported significant gains in the group of FEM matrices, but an interesting and curious result is the modest gain in performance they reported in the group of matrices labeled as other applications.

This curious result motivated our study of symmetric matrices with few off diagonals elements and banded matrices. In our experimental study we handled most of the symmetric matrices taken from Lee's matrix benchmark suite used in (Lee et al. 2003, Lee et al. 2004), some additional symmetric sparse matrices taken from the SPARSITY benchmark suite (Im et al. 2004), plus some other matrices we considered representative of the domain of symmetric sparse matrices. These new additions are: diagonal (bcsstm39), symmetric tridiagonal, symmetric five diagonal, and matrices with few off diagonal elements. Table 1 shows the name, application area, dimension , r factor ($r = \frac{nnz}{dim}$), number of non zero values (nnz), and number of stored values for each matrix of the benchmark suite divided in groups. The first two group shows a set of matrices from (Lee et al. 2003, Lee et al. 2004) where there is not doubt that CSR is the format to be used to compute the SpMV multiplication. Dense is the only matrix in group one and its the epitome of performance to benefit from optimization techniques. The third group represents those matrices in (Lee et al.

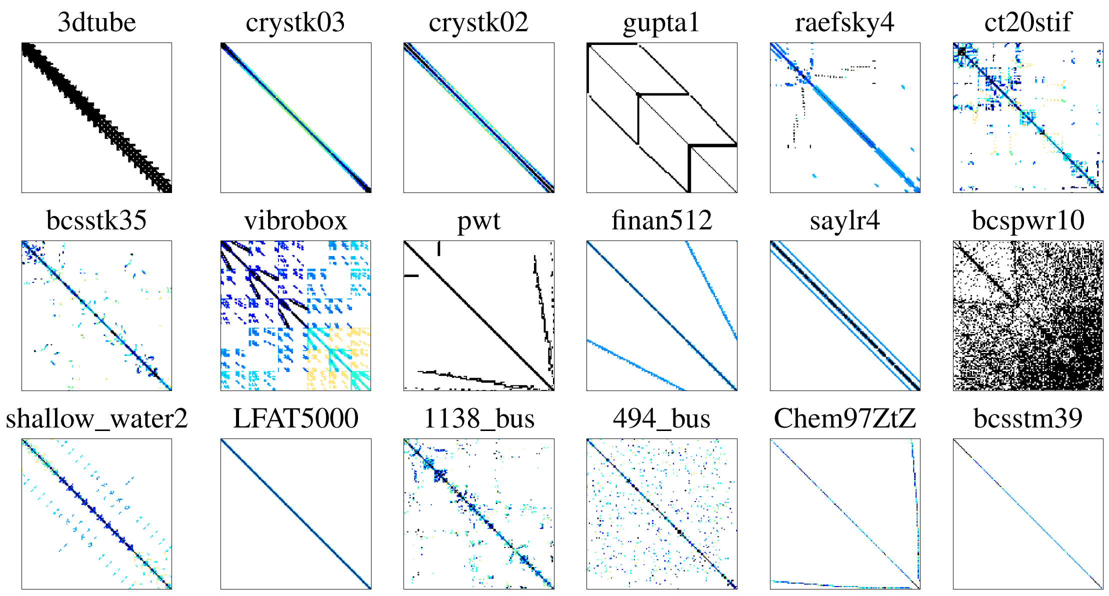


Figure 1: Matrix benchmark suite from Matrix Market.

2003, Lee et al. 2004) which did not gain a significant improve on performance when register blocking and other optimizations are carried out. This last set of matrices, characterizes the matrices where the SpMV multiplication performs better using the TJDS format instead of CSR. To visualize the matrix structure or non zero distribution we include as well a picture of each matrix in Figure 1. The picture of matrix bcsstm39 which is diagonal, also represents the symmetric banded matrices (tridiagonal and pentadiagonal). Dense matrix picture is excluded. All matrices were collected from the University of Florida Sparse Matrix Collection (Davis and Hu 2011) and Matrix Market (Boisvert et al. 1997.)

4 STORAGE FORMATS AND ALGORITHMS

When a symmetric sparse matrix, $A = A^T$, is used to compute the symmetric sparse matrix-vector multiply (SpMV) we just need to store the lower or upper triangle of the matrix depending on the compressed storage format adopted for the computation. We will use matrix A to describe the different formats and algorithms used in this experiment.

$$A = \begin{bmatrix} a_{11} & 0 & a_{13} & 0 & 0 & 0 \\ 0 & a_{22} & 0 & 0 & 0 & 0 \\ a_{31} & 0 & a_{33} & a_{34} & a_{35} & a_{36} \\ 0 & 0 & a_{43} & a_{44} & a_{45} & a_{46} \\ 0 & 0 & a_{53} & a_{54} & a_{55} & a_{56} \\ 0 & 0 & a_{63} & a_{64} & a_{65} & a_{66} \end{bmatrix}$$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} = \begin{bmatrix} a_{11} & 0 & a_{13} & 0 & 0 & 0 \\ 0 & a_{22} & 0 & 0 & 0 & 0 \\ a_{31} & 0 & a_{33} & a_{34} & a_{35} & a_{36} \\ 0 & 0 & a_{43} & a_{44} & a_{45} & a_{46} \\ 0 & 0 & a_{53} & a_{54} & a_{55} & a_{56} \\ 0 & 0 & a_{63} & a_{64} & a_{65} & a_{66} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix}$$

value_list	a_{11}	a_{13}	a_{22}	a_{33}	a_{34}	a_{35}	a_{36}	a_{44}	a_{45}	a_{46}	a_{55}	a_{56}	a_{66}
col_index	1	3	2	3	4	5	6	4	5	6	5	6	6
row_start	1	3	4	8	11	13	14						

Figure 2: CSR data structures to compute the symmetric SpMV multiplication.

Algorithm 1 Symmetric SpMV-Lee et al.

```
$i \leftarrow 1$;
for $r \leftarrow 1$ to $n$ do
    $y\_temp \leftarrow 0$;
    $x\_temp \leftarrow x[r]$;
    $row\_ptr \leftarrow row\_start[r]$;
                                                ▷ Special handling of the diagonal
    if $r = col\_index[i]$ then
        $y\_temp \leftarrow value\_list[i] * x[i]$;
        $i \leftarrow i + 1$;
        $row\_ptr \leftarrow row\_ptr + 1$;
    end if
                                                ▷ Loop over non-zeros in row i
    for $j = row\_ptr$ to $row\_start[i+1] - 1$ do
        $c \leftarrow col\_index[j]$;
        $a\_val \leftarrow value\_list[j]$;
        $y\_temp \leftarrow y\_temp + a\_val * x[c]$;
        $y[c] \leftarrow y[c] + a\_val * x\_temp$;
        $i \leftarrow i + 1$;
    end for

    $y[i] = y\_temp$;
end for
```

CSR is an efficient gather base compressed storage format which is widely used and can be considered de facto standard to compare with. Performance optimizations of the SpMV multiplication using CSR has been, and still is, the focus of a many research projects searching for improving performance on different architectures (Williams et al. 2009, Krotkiewski and DabrowskiArrow 2010, Bulu et al. 2011). We can characterize CSR as follows:

- The elements of the source vector x are accessed in a discontiguous way creating low spatial locality (gather base).
- There is also a high per row loop overhead because it generates many short vectors.
- The matrix elements are used just once and are accessed sequentially in row order producing thus low temporal locality (low reuse).
- The elements of the destination vector y are accessed sequentially and they are used many times showing optimal temporal locality(high reuse).
- The outer loop is executed n times.

To create the CSR representation of the above illustrated matrix, the non-zero coefficients of matrix A are compressed along the row and stored one row after another in a linear array that we denote as *value_list*.

Another array, *col_index*, is used to store the column indices associated to each coefficient of matrix A. A third array, *row_position* is necessary to identify the beginning of each row in the *value_list* and *col_index* arrays. Figure 2 shows the data structures required to compute the symmetric SpMV kernel using a variant of Lee's algorithm (Lee et al. 2003) as described in Algorithm 1. The algorithm assumes that before compression only the upper-triangle of A is stored. Storing the upper-triangle saves space and alleviates the presure on memory traffic because each matrix coefficient is accessed once and used twice.

TJDS is a scatter base compressed storage format which produces long vectors. We can characterize it as follows:

- The elements of the source vector x are accessed sequentially and are reused at each iteration (good locality).
- There is also a low per row loop overhead in the outer loop because it generates large vectors and the number of iterations is $num\_tjdiag \ll n$.
- The matrix elements are used just once and are accessed sequentially producing low temporal locality (low reuse).
- The elements of the destination vector y are accessed randomly and they are used many times showing low spatial locality(low reuse).
- Source array x has to be permuted before the computation.
- Destination array y have to be permuted after the computation.

To store the symmetric matrix A using the TJDS format, we proceed as follows: The non-zero coefficients of lower triangle of the symmetric matrix are compressed along the columns and reordered in decreasing order of the number of non-zero elements per column as shown below:

$$\left|\begin{array}{cccccc} \mathbf{a_{33}} & \mathbf{a_{44}} & \mathbf{a_{11}} & \mathbf{a_{55}} & \mathbf{a_{22}} & \mathbf{a_{66}} \\ a_{43} & a_{54} & a_{31} & a_{65} & 0 & 0 \\ a_{53} & a_{64} & 0 & 0 & 0 & 0 \\ a_{63} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right|$$

The final step is to permute arrays x and y using the same reordering we applied to the matrix columns. Figure 3 shows the data structures and Algorithm 2 the sequential algorithm to perform the symmetric matrix-vector multiplication, $y = Ax$, using the TJDS format. In this branchless algorithm num_tjdiag stands for the number of transpose jagged diagonals. It is worth mentioning that $num\_tjdiag \ll n$. The length of the column with the largest nnz provides the value of num_tjdiag. We include the source array x and the destination array y in Figure 3 to show the permutation applied to array x and how the results, array y, will be gathered already permuted. For instance: Let P be the permutation array, then if we need to compute $y = A^n x$ using TJDS, we must compute $Py = A^n Px$.

Then, the first transpose jagged diagonal tjd is obtained by gathering the first coefficient of each column into the linear array *value_list* (see highlighted a_{ij} values above and first segment in the *value_list* array in Figure 3). Then we proceed in a similar way with the second element of each column for the second diagonal or segment, and so forth. Another array denoted as *row_index* is used to store the row indices of the coefficients of the matrix for knowing where the non-zero elements fit into the sparse matrix. A third array, called *start_position*, stores the starting position of each tjd diagonal stored in the arrays *value_list* and *row_index*. Although in the TJDS data structure there is a permutation vector involved, the information

Y	y_3	y_4	y_1	y_5	y_2	y_6							
X	x_3	x_4	x_1	x_5	x_2	x_6							
value_list	a_{33}	a_{44}	a_{11}	a_{55}	a_{22}	a_{66}	a_{43}	a_{54}	a_{31}	a_{65}	a_{53}	a_{64}	a_{63}
row_index	3	4	1	5	2	6	2	4	1	6	4	6	6
start_position	1	7	11	13	14								

Figure 3: TJDS data structures to compute the symmetric SpMV multiplication.

Algorithm 2 Symmetric SpMV-TJDS

```
$i \leftarrow 1$;                                                    ▷ Special handling of the diagonal
for $j = Start\_position[i]$ to $Start\_position[i+1]-1$ do
    $y[j] \leftarrow value\_list[j] * x[j]$;

end for
                                                                     ▷ Loop over off diagonal elements
for $i = 2$ to $num\_tjdiag$ do
    $k \leftarrow 0$;
    for $j = Start\_position[i]$ to $Start\_position[i+1]-1$ do
        $p \leftarrow row\_index[j]$;
        $a\_val \leftarrow value\_list[j]$;
        $y[p] \leftarrow y[p] + a\_val * x[k]$;
        $y[k] \leftarrow y[k] + a\_val * x[p]$;
        $k \leftarrow k+1$;
    end for
end for
```

of the permutation array coincides with the first segment of the *row_index* therefore no additional storage is needed for the permutation array because it is implicitly located in the *row_index* array; this make TJDS a more compact format than CSR. Last but not least, we need to mention that all the indices of the *row_index* array starting from the second segment onwards are also permuted.

4.1 Symmetric banded matrices

We have explored as well the computation of the symmetric SpMV multiplication for banded matrices under the TJDS format. In our experiment we handled tridiagonals, pentadiagonals, and n-diagonals (dense matrix) and found that TJDS provides some advantages for this class of matrices. For instance, reordering after compressing the columns is not necessary because the columns are already ordered in decreasing order and this entails that the source array x and destination array y do not have to be permuted either. The data structures are illustrated in Figure 4 and the algorithm to compute the SpMV multiplication for banded matrices using the TJDS format is described in Algorithm 3. In this algorithm we have moved out the indirection of the inner loop and replaced by a double indirection before starting the inner loop. This subtle modification in the algorithm reduces the number of memory accesses to the index array from $(mn-m)$ to m, where n is the matrix dimension and m defines the band (see (1) in section 3). This a significant reduction in memory traffic. Furthermore, the column indices could be easily generated as in the variable band format (Jennings 1966, Duff 1977) eliminating thus all the indirections.

Y	y_1	y_2	y_3	y_4	y_5	y_6					
X	x_1	x_2	x_3	x_4	x_5	x_6					
value_list	a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}	a_{21}	a_{32}	a_{43}	a_{54}	a_{65}
row_index	1	2	3	4	5	6	2	3	4	5	6
start_position	1	7	12								

Figure 4: TJDS data structures to compute the symmetric SpMV multiplication for banded matrices.

Algorithm 3 Banded Symmetric SpMV-TJDS

```
$i \leftarrow 1$;                                                    ▷ Special handling of the diagonal
for $j = Start\_position[i]$ to $Start\_position[i+1]-1$ do
    $y[j] \leftarrow value\_list[j] * x[j]$;
end for
                                                                     ▷ Loop over off diagonal elements
for $i = 2$ to $num\_tjdiag$ do
    $k \leftarrow 1$;
    $p \leftarrow row\_index[Start\_position[i]]$;
    for $j = Start\_position[i]$ to $Start\_position[i+1]-1$ do
        $a\_val \leftarrow value\_list[j]$;
        $y[p] \leftarrow y[p] + a\_val * x[k]$;
        $y[k] \leftarrow y[k] + a\_val * x[p]$;
        $k \leftarrow k+1$;
        $p \leftarrow p+1$;
    end for
end for
```

5 SEQUENTIAL PERFORMANCE

In our experiment we have examined the symmetric benchmark matrix suite used in (Lee et al. 2003, Lee et al. 2004) which is a subset of the SPARSITY benchmark matrix suite, and we have added a set of symmetric matrices whose structure is a main diagonal plus few off diagonal elements and banded matrices. The matrices we have added are characterized by a factor or rule of thumb denoted as $r = \frac{nnz}{dim} < 2.5$. Indeed, we have found a threshold ($th = 3.6$) which leads us to determine beforehand the usage of CSR or TJDS to compute the symmetric SpMV multiplication.

We used the twenty one matrices from Table 1 and run the Symmetric SpMV mutiplication on Ivy Bridge. Figure 5 shows that CSR is a better choice for matrices beyond $r > 3.63$ and TJDS does it better when $r \leq 3.63$. Figure 6 shows the results of executing the symmetric SpMV in Harpertown for the same set of matrices. The outcome is similar to the one presented in Figure 5 but it can be noticed that matrix pwt ($r \leq 4.96$) behaves differently because it runs better using TJDS. Both Figures (5 and 6) show that $r = 3.63$ is the tipping point where it is recommended to switch from CSR to TJDS. This point corresponds to matrix saylr4 in our tests. We can also observe that the gap between the two plots is closer in Harpertown showing this way the impact of the architecture on the execution of the symmetric SpMV. We believe that there is a zone between $3.5 \leq r \leq 5.0$ where we can find some matrices that contradict our findings as was the case of pwt ($r \leq 4.96$) that did better using TJDS and we were expecting that CSR was the format to use. Further studies should be made on those matrices whose r value does not follows the expected behavior of our empirical assertion. In both experiments, each SpMV multiplication iterates 1000 for each matrix of the benchmark suite and each matrix was run 30 times. Table 2 shows the hardware and software configuration use in this benchmark.

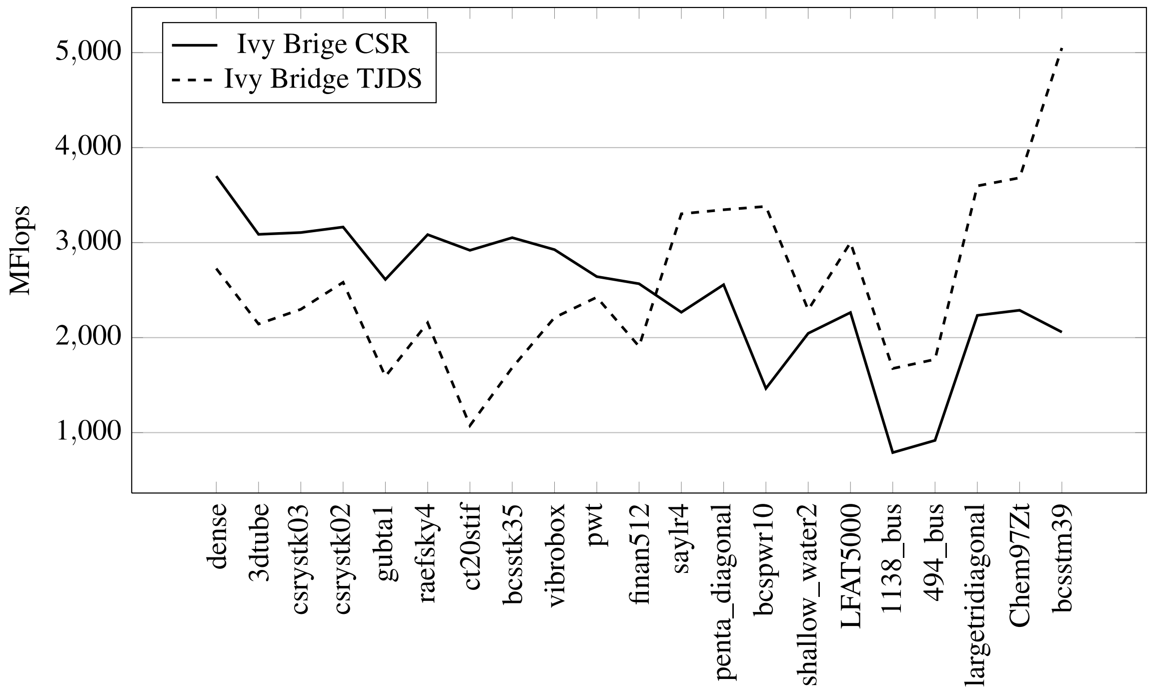


Figure 5: Mflops for the symmetric SpMV multiplication on Ivy Bridge using CSR and TJDS.

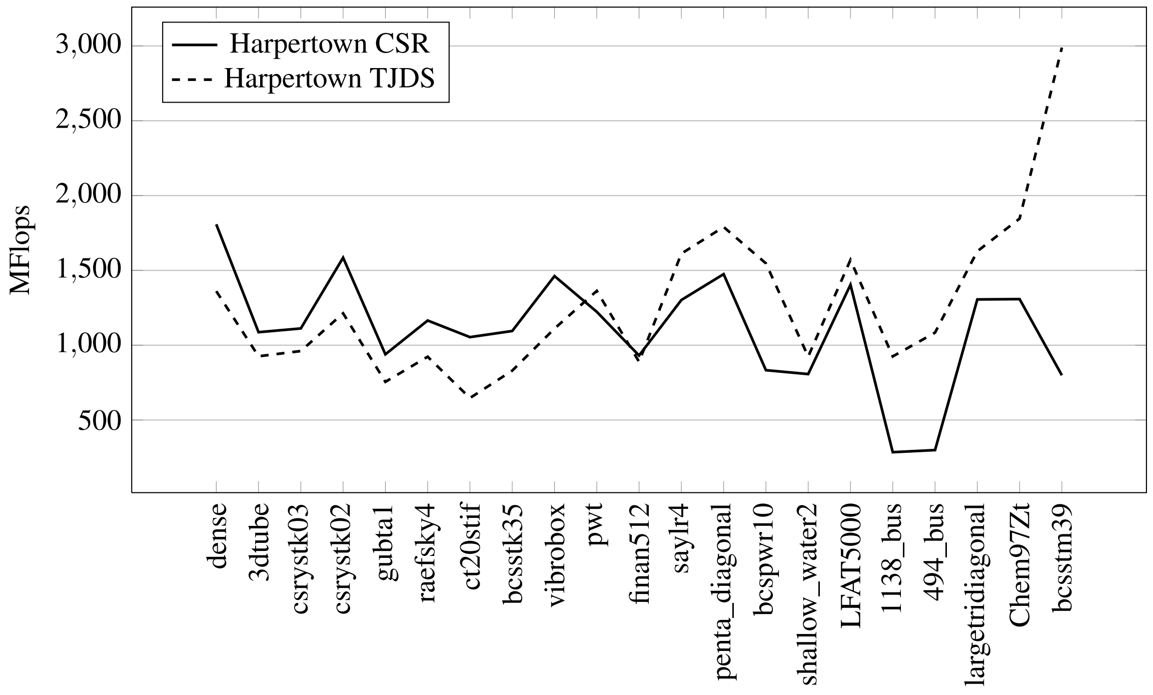


Figure 6: Mflops for the symmetric SpMV multiplication on Harpertown using CSR and TJDS.

Table 2: System Platform.

	Ivy Bridge	**Harpertown**
Model	Intel Core i5-3470	Intel Xeon E5430
Architecture	x86_64	x86_64
Microarchitecture	Intel Ivy Bridge	Intel Core
Clock freq.	3.2 GHz	2.66 GHz
L1 cache (D/I)	32KB/32KB	32KB/32KB
L2 cache	256KBx4	6MBx2
L3 cache	6MB	–
Cores/Threads	4/4	4/4
Compiler	gcc	gcc
Flags	-O3	-O3

6 CONCLUSIONS

One of the advantages of dealing with symmetric matrices to compute the SpMV multiplication is that sy mmetry allows the saving of space because only the diagonal and half of the off diagonal elements have to be stored, alleviating thus memory traffic. In addition, we have found empirical evidences that lead us to recommend the usage of the TJDS format when computing the symmetric SpMV multiplication for tridiagonals, five-diagonals, and matrices made of a main diagonal and few non zero off diagonal elements because TJDS requires less memory space and also outperform the CRS version for matrices whose r factor ($r = \frac{nnz}{dim}$) is less than 3.63. It is worth mentioning here that tridiagonals and penta-diagonals matrix are of common used in scientific computing and their r factor is 1 and 1.9 respectively.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1111407. We would like to thank the University of Illinois where this work began. David Padua and María J. Garzarán who gave scientific guidance and provided useful suggestions. We also thank the anonymous referees for their review, which helped improve the paper. Thank you Andrew Hughes who participated in discussions.

REFERENCES

Boisvert, R. F., R. Pozo, K. Remington, R. F. Barrett, and J. J. Dongarra. 1997. “Matrix Market: a web resource for test matrix collections”. In *Quality of Numerical Software, Assessment and Enhancement*, edited by B. R.F, Chapter 9, pp. pp. 125–137. London, Chapman and Hall.

Bulu, A., S. Williams, and J. Demmel. 2011. “Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication”. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 721–733.

Cuthill, E., and J. McKee. 1969. “Reducing the bandwidth of sparse symmetric matrices”. In *Proceedings of the 24th National Conference of the ACM*, pp. 157–172.

Davis, T. A., and Y. Hu. 2011, dec. “The University of Florida Sparse Matrix Collection”. *ACM Trans. Math. Softw.* vol. 38 (1), pp. 1–25.

Dongarra, J. 2000. “Sparse Matrix Storage Formats”. In *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*, edited by Z. B. et al. Philadelphia, SIAM.

Duff, I. S. 1977. “A Survey of Sparse Matrix Research”. *Proceedings of the IEEE* vol. 65, pp. 500–535.

Im, E. J., K. Yelick, and R. Vuduc. 2004. “SPARSITY: An Optimization Framework for Sparse Matrix Kernels”. *International Journal of High Performance Computing* vol. 18 (1), pp. 135–158.

Jennings, A. 1966. “A Compact Storage Scheme for the Solution of Simultaneous Equations”. *The Computer Journal* vol. 9 (3), pp. 281–285.

Krotkiewski, M., and M. DabrowskiArrow. 2010. “Parallel symmetric sparse matrix-vector product on scalar multi-core CPUs”. *Parallel Computing* vol. 36 (4), pp. 181–198.

Lee, B., R. Vuduc, J. W. Demmel, and K. A. Yelick. 2004, Aug.. “Performance Models for Evaluation and Automatic Tuning of Symmetric Sparse Matrix-Vector Multiply”. In *International Conference on Parallel Processing*, Volume 1, pp. 169–176.

Lee, B. C., R. W. Vuduc, J. W. Demmel, K. A. Yelick, M. de Lorimier, and L. Zhong. 2003. “Performance Optimizations and Bounds for Sparse Symmetric Matrix-Multiple Vector Multiply”. Technical Report UCB/CSD-03-1297, EECS Department, University of California, Berkeley.

Montagne, E., and A. Ekambaran. 2004. “An Optimal Storage Format for Sparse Matrices”. *Information Processing Letters* vol. 90 (2), pp. 87–92.

Pinar, A., and M. T. Heath. 1999. “Improving Performance of Sparse Matrix-vector Multiplication”. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, SC ’99. New York, NY, USA, ACM.

Saad, Y. 1989. “Krylov Subspace methods on Supercomputers”. *Siam Journal Scientific and Statistical Computing* vol. 10 (6), pp. 1200 –1232.

Saad, Y. 1996. *Iterative Methods for Sparse Linear Systems*. Boston, MA, Prentice Hall, Englewood Cliff.

Schwarz, H. 1973. *Numerical Analysis of Symmetric Matrices*. Prentice-Hall series in automatic computation. Prentice-Hall.

Sedaghati, N., L. Pouchet, S. Parthasarathy, and P. Sadayappan. 2015, June. “Automatic Selectionof Sparse Matrix Representation on GPUs”. In *International Conference on Supercomputing*, pp. 99–108.

Vuduc., R. W. 2004, 1. *Automatic Performance Tuning of Sparse Matrix Kernels*. Ph. D. thesis, University of California, Berkeley, The address of the publisher.

Williams, S., L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. 2009. “Optimization of sparse matrix-vector multiplication on emerging multicore platforms”. *Parallel Computing* vol. 35 (3), pp. 178–194.

AUTHOR BIOGRAPHIES

EDWARD AYMERICH works at Hewlett Packard Enterprise in Costa Rica. He holds a MSc in Computer Science from University of Central Florida. His research interests include program optimization and computer graphics. His email address is edward.aymerich@gmail.com.

ALEXANDRE DUCHATEAU works at Pure Storage. He holds a Ph.D. in Computer Science from University of Illinois at Urbana-Champaign. His research interest includes program optimization, compilers, and security. His email address is lex@purestorage.com.

EURIPIDES MONTAGNE is an Associate Lecturer in the Department of Computer Science at the University of Central Florida. He holds a Ph.D. from Universidad Central de Venezuela. His research interests include program optimization, computer architecture and irregular problems. His email address is eurip@eecs.ucf.edu.

FRANK PLOCHAN works as a consultant in California. He holds a MSc in Computer Science from University of Central Florida. His research interest includes program optimization and software Engineering. His email address is researcherfp@gmail.com

REMOTE HIGH PERFORMANCE VISUALIZATION OF BIG DATA FOR IMMERSIVE SCIENCE

Faiz Abidi

Department of Computer Science
Virginia Tech
Blacksburg, Virginia 24060
fabidi89@vt.edu

Nicholas Polys

Department of Computer Science
Virginia Tech
Blacksburg, Virginia 24060
npolys@vt.edu

Srijith Rajamohan

Advanced Research Computing
Virginia Tech
Blacksburg, Virginia 24060
srijithr@vt.edu

Lance Arsenault

Advanced Research Computing
Virginia Tech
Blacksburg, Virginia 24060
lanceman@vt.edu

Ayat Mohammed

Department of Computer Science
Virginia Tech
Blacksburg, Virginia 24060
maaayat@vt.edu

1 ABSTRACT

Remote visualization has emerged as a necessary tool in the analysis of big data. High-performance computing clusters can provide several benefits in scaling to larger data sizes, from parallel file systems to larger RAM profiles to parallel computation among many CPUs and GPUs. For scalable data visualization, remote visualization tools and infrastructure is critical where only pixels and interaction events are sent over the network instead of the data. In this paper, we present our pipeline using VirtualGL, TurboVNC, and ParaView to render over 40 million points using remote HPC clusters and project over 26 million pixels in a CAVE-style system. We benchmark the system by varying the video stream compression parameters supported by TurboVNC and establish some best practices for typical usage scenarios. This work will help research scientists and academicians in scaling their big data visualizations for remote, real-time interaction.

Keywords: Remote rendering, CAVE, HPC, ParaView, big data.

2 INTRODUCTION

Network is one of the biggest bottlenecks of remote rendering and we want to minimize the data sent and received from the server. The most commonly used method to do this is to compress the data. However, compression has its own disadvantages. Compression and decompression consumes CPU time meaning that some extra time will be added to the rendering process. Another disadvantage is that compression can lead to loss of data depending on what type of algorithms are being used (lossy versus lossless algorithms). Apart from compression, there are other factors that can help minimize network bandwidth usage at the cost of reduced render quality like JPEG quality, JPEG chrominance sub-sampling, and the amount of compression level applied. The goal of this work is to identify which of these factors is significant for remote rendering of big data over a dedicated 10 Gbps network.

2.1 Graphics rendering

Graphics rendering pipeline consists of two phases - geometry processing and rasterization. The authors in (Molnar, Cox, Ellsworth, and Fuchs 1994) have described three classes of parallel rendering algorithms; sort-first, sort-middle, and sort- last. A significant advantage of the sort-last algorithm is that the renderers implement the complete rendering pipeline and are independent until pixel merging. In general, all the three rendering algorithms suffer from some common problems like load balancing, high processing and communication costs, and highly pixelated content. In this paper, we used ParaView (Kitware 2017) to run our experiments, which implements sort-last algorithm in its code base. Samanta et al. in (Samanta, Funkhouser, Li, and Singh 2000) proposed a hybrid of sort-first and sort-last parallel polygon rendering algorithm. The hybrid algorithm outperforms sort-first and sort-last algorithms in terms of efficiency. Overall, the hybrid algorithm achieved interactive frame rates with an efficiency of 55% to 70.5% during simulation with 64 PCs. The hybrid algorithm provides a low cost solution to high performance rendering of 3D polygonal models.

Visualizing big data sets has always been challenging because of limited CPU, memory, parallelization, and network requirements and Ahrens et al. talk more about it in (Ahrens, Desai, McCormick, Martin, and Woodring 2007). Ahrens et al. (Ahrens, Brislawn, Martin, Geveci, Law, and Papka 2001) also discuss similar problems with respect to large-scale data visualization and present an architectural approach based on mixed dataset topology parallel data streaming. Moreland et al. in (Moreland, Wylie, and Pavlakos 2001) used sort-last parallel rendering algorithm to render data to large tile displays. Morland et al. in (Moreland and Thompson 2003) describe a new set of parallel rendering components for the Visualization Toolkit (VTK) (Schroeder, Martin, and Lorensen 1996), which is also used by ParaView for data visualization. They introduced components of Chromium (Humphreys, Houston, Ng, Frank, Ahern, Kirchner, and Klosowski 2002) and ICE-T (an implementation of (Moreland, Wylie, and Pavlakos 2001)) into VTK and showed that VTK can be a viable framework for cluster-based interactive applications that require remote display.

Eilemann et al. (Eilemann, Makhinya, and Pajarola 2009) introduced a system called Equalizer, a toolkit for parallel rendering based on OpenGL. Equalizer takes care of distributed execution, synchronization, and final image composting, while the application programmer identifies and encapsulates culling and rendering. This approach is minimally invasive since the proprietary rendering code is retained.

2.2 Remote Scientific Visualization

ParaView (Ahrens, Geveci, and Law 2005)(Kitware 2017) is an open-source toolkit that allows scientists to visualize large datasets. It is based on the visualization toolkit called VTK (Schroeder, Martin, and Lorensen 1996) that provides the data representations for a variety of grid types including structured, unstructured, polygonal, and image data. ParaView extended VTK to support streaming of all data types and parallel execution on shared and distributed-memory machines.

TurboVNC is a derivative of VNC (Virtual Network Computing) (Richardson, Stafford-Fraser, Wood, and Hopper 1998) that has been specifically developed to provide high performance for 3D and video workloads (TurboVNC 2017). It is an open source project that contains a modern X server code base (X.org 7.7).

VirtualGL (VirtualGL 2017) is an open source software that enables remote display software the ability to run OpenGL applications with full 3D hardware acceleration. It redirects the 3D rendering commands and the data of the OpenGL applications to a graphical processing unit (GPU) installed on the remote server and sends the rendered 3D images to the client.

A CAVE automated virtual environment (Cruz-Neira, Sandin, DeFanti, Kenyon, and Hart 1992) is a virtual reality interface that consists of walls and each wall can be driven by one or more projectors. The first CAVE was developed at the University of Illinois, Chicago Electronic Visualization Laboratory in 1992 for scientific and engineering applications and overcome the limitations of head mounted displays (HMD).

3 DESIGN AND IMPLEMENTATION

3.1 Experimental setup

ParaView consists of a data server, render server, and a client. The data and the render server can be separate machines but if they are on the same host, they are referred to as a pvserver. In our experiments, we had the render and the data server running on the same machine acting as the pvserver. We used NewRiver (Computing 2017) for remote rendering and the Hypercube machine in our lab for running the ParaView client. A vncserver was started on Hypercube and a user could use that vncsession to connect to Hypercube to run the experiments. Ganglia daemon (Sourceforge 2016) running on Hypercube was set at a polling frequency of 3 seconds and used to collect metrics like memory and network usage. By default, Ganglia sets the polling frequency at 15 seconds but we wanted to poll more frequently to collect more accurate data. However, polling every second is also not recommended since that could stress the ganglia server and therefore we decided to poll every 3 seconds instead. Our experimental pipeline consisted of Paraview, TurboVNC, and VirtualGL. Figure 1 shows the architectural overview and Figure 2 shows a flow diagram of the different components talking to each other. Figure 3 shows a 40 million point cloud dataset being analyzed in the CAVE at Virginia Tech.

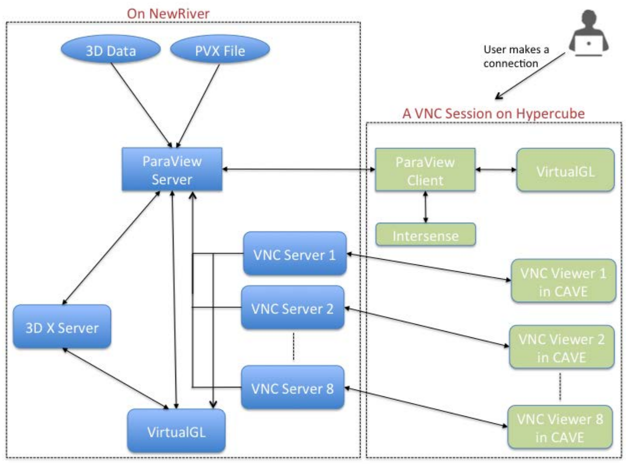


Figure 1: Overview of the experimental setup

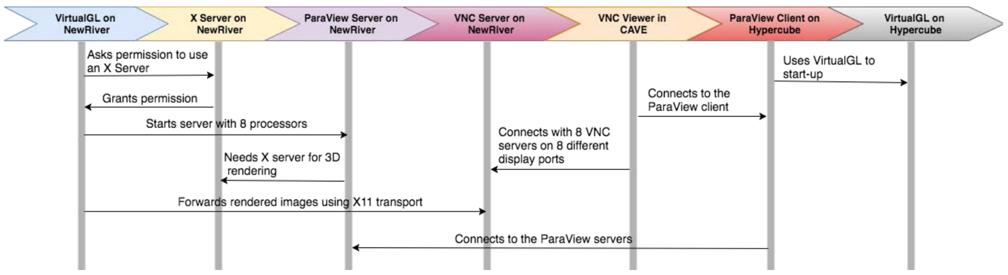

VirtualGL on NewRiver
X Server on NewRiver
ParaView Server on NewRiver
VNC Server on NewRiver
VNC Viewer in CAVE
ParaView Client on Hypercube
VirtualGL on Hypercube
Asks permission to use an X Server
Grants permission
Starts server with 8 processors
Needs X server for 3D rendering
Forwards rendered images using X11 transport
Connects with 8 VNC servers on 8 different display ports
Connects to the ParaView client
Uses VirtualGL to start-up
Connects to the ParaView servers

Figure 2: Flowdigram of the experimental setup

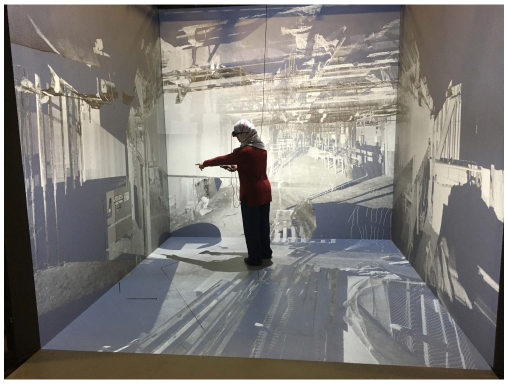


Figure 3: 40 million point cloud data set visualized using our experimental pipeline and remote HPC servers

3.2 Hardware specifications and data used

We used the NewRiver HPC cluster at Virginia Tech for remote rendering. Each host had 24 CPUs, 512 GB of memory, and 2 Tesla K80 GPUs. The Hypercube machine used to run the ParaView client had 48 CPUs, 512 GB of memory, and 4 Nvidia Quadro M6000 GPUs. Each projector used in the lab to run the CAVE runs at a frequency of 120 HZ with active stereo and a resolution of 2560x1600. We used a point cloud dataset that had more than 5 billion points in it. It is a LiDAR (Reutebuch, Andersen, and McGaughey 2005) mass scan of the Goodwin hall at Virginia Tech that has motion sensors installed at different locations. The total size of this dataset was 270 GB and it was available in the form of CSV files. This data was cleaned of the noise and converted into unstructured XML VTK format in binary also called VTU (FEM 2016) format using ParaView's D3 filter (Kitware 2017). Converting the CSV files into VTU files was necessary to make sure that the data can be distributed in parallel when running ParaView with MPI.

3.3 Data Interaction and ParaView's timer log

ParaView provides a timer log that monitors the time taken to render the data during different stages. We were interested in observing the still render time and the interactive render time. Still render measures the time from when the render is initiated to the time when the render completes. Interactive render occurs when a user moves and interacts with the 3D data using the ParaView GUI. We needed the interactions to remain consistent for all the experiments run and for this reason we decided to script the mouse movements on the client machine using a Python library called pynout (Palmér 2014). We added 1000 random scripted mouse movements on the client side interacting with the data shown on the ParaView GUI. We used the values of still and interactive renders obtained from the timer log and averaged them to obtain the Framerate Per Second (FPS) reported here.

3.4 Other compression options

ParaView provides several compression parameters that can be set with remote rendering. However, it does not allow to script the parameters and every time a user wants to make a change, they have to do it using the GUI. For this reason, we decided to not use ParaView's compression settings. VirtualGL also lets a user set compression settings but it does not perform any compression when using the X11 transport method in which case it relies on the X proxy to encode and deliver the images to the client. We did not realize this initially when we started running our experiments and were trying to vary the compression parameters supported by VirtualGL. As expected, we did not observe any difference in the results that we got and this led us to explore the compression supported by TurboVNC instead. The compression settings provided by TurboVNC were relevant for our case and we could script them for our experiments.

4 RESULTS

In this section, we discuss the different results obtained by varying the compression parameters supported by TurboVNC. We measured the maximum, minimum, and average network and memory consumed at the client side, the maximum memory consumed at the server side, and the average still and interactive frame rates. Note that we did not measure percentage CPU on the server side because when we run pvservers with MPI, we get 100% CPU utilization even when the pvserver sits idle. This is just how the MPI layer is implemented in the two most common implementations: OpenMPI (MPI 2017) and MPICH (MPICH 2017).

We created nine groups based on the different compression parameters as shown in Table 1 and all the results obtained by varying the number of VTU files and CPUs are shown in Figure 4.

Table 1: Different compression parameters used in the experiments

#	Encoding	JPEG	Quality	JPEG Subsampling	Compression Level
CASE 1	Tight	0	95	1x	0
CASE 2	Tight	0	95	1x	1
CASE 3	Tight	0	95	1x	5
CASE 4	Tight	0	95	1x	6
CASE 5	Tight	1	95	1x	0
CASE 6	Tight	1	95	1x	1
CASE 7	Tight	1	95	1x	5
CASE 8	Tight	1	80	2x	6
CASE 9	Tight	1	30	4x	7

4.1 Frame rates with 8 VTU files and 8 CPUs

For still render FPS, the average obtained as shown in Figure 4 was 6.536 with a standard deviation of 0.118 and a standard error of 0.039. CASE 1 (JPEG:0, Quality:95, JPEG Subsampling:1x, Compression Level:0) with 6.727 FPS performed the best while CASE 4 (JPEG:0, Quality:95, JPEG Subsampling:1x, Compression Level:6) with 6.305 FPS performed the worst. For interactive render FPS, the average obtained as shown in Figure 4 was 52.073 with a standard deviation of 4.393 and a standard error of 1.464. CASE 8 (JPEG:1, Quality:80, JPEG Subsampling:2x, Compression Level:6) with 55.907 FPS preformed the best while CASE 4 with 40.172 FPS performed the worst. Overall, there was not much difference between the different still and interactive render FPS obtained for the different cases except for CASE 4.

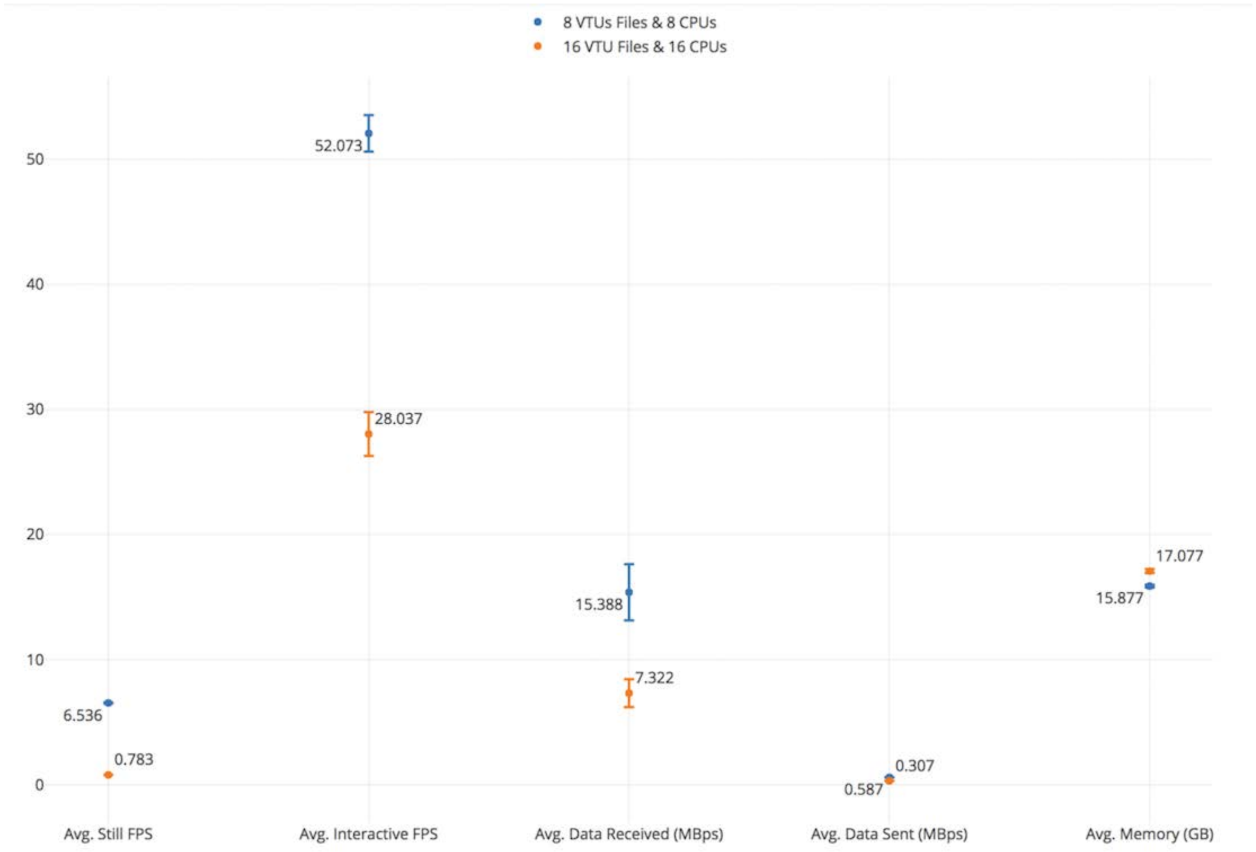


Figure 4: Experimental results obtained using different combinations

4.2 Frame rates with 16 VTU files and 16 CPUs

For still render FPS, the average obtained as shown in Figure 4 was 0.783 with a standard deviation of 0.004 and a standard error of 0.001. CASE 7 (JPEG:1, Quality:95, JPEG Subsampling:1x, Compression Level:5) with 0.792 FPS performed a little better than the others while CASE 1 (JPEG:0, Quality:95, JPEG Subsampling:1x, Compression Level:0) and CASE 2 (JPEG:0, Quality:95, JPEG Subsampling:1x, Compression Level:1) performed the worst with 0.778 FPS each. For interactive render FPS, the average obtained as shown in Figure 4 was 28.037 with a standard deviation of 5.255 and a standard error of 1.751. CASE 9 (JPEG:1, Quality:30, JPEG Subsampling:4x, Compression Level:7) with 33.946 FPS preformed the best while CASE 2 (JPEG:0, Quality:95, JPEG Subsampling:1x, Compression Level:1) with 18.699 FPS performed the worst. Overall, there was not much difference between the different still and interactive render FPS obtained for the different cases except for CASE 2 and CASE 4 (JPEG:0, Quality:95, JPEG Subsampling:1x, Compression Level:6).

4.3 Network usage with 8 VTU files and 8 CPUs

Figure 4 shows the average data received by the client at 15.388 MBps with a standard deviation of 6.723 and a standard error of 2.241 while the average data sent by the client was at 0.587 MBps with a standard deviation of 0.017 and a standard error of 0.005. The maximum data received by the client at any given point in time during the rendering process was almost the same for all cases except CASE 3 (JPEG:0, Quality:95, JPEG Subsampling:1x, Compression Level:5) at 37 MB/sec and CASE 4 (JPEG:0, Quality:95, JPEG Subsampling:1x, Compression Level:6) at 40.7 MB/sec. On an average, CASE 1 (JPEG:0, Quality:95, JPEG Subsampling:1x, Compression Level:0) and CASE 3 received the maximum amount of data at 24.4 MB/sec each while the least average data was received by CASE 9 (JPEG:1, Quality:30, JPEG Subsampling:4x, Compression Level:7) at 6 MB/sec. The average data sent from the client to the server was almost the same for all the cases but it was the least for CASE 7 (JPEG:1, Quality:95, JPEG Subsampling:1x, Compression Level:5) at 0.554 MB/sec.

4.4 Network usage with 16 VTU files and 16 CPUs

Figure 4 shows the average data received by the client at 7.322 MBps with a standard deviation of 3.333 and a standard error of 1.111 while the average data sent by the client was at 0.307 MBps with a standard deviation of 0.023 and a standard error of 0.007. The maximum data received by the client at any given point in time during the rendering process was almost the same for all the cases (52.8 MB/sec) except for CASE 7 (JPEG:1, Quality:95, JPEG Subsampling:1x, Compression Level:5) at 41.8 MB/sec. On an average, CASE 1 (JPEG:0, Quality:95, JPEG Subsampling:1x, Compression Level:0) at 12.4 MB/sec received the maximum amount of data while the least average data was received by CASE 9 (JPEG:1, Quality:30, JPEG Subsampling:4x, Compression Level:7) at 2.8 MB/sec. The average data sent from the client to the server was least for CASE 6 (JPEG:1, Quality:95, JPEG Subsampling:1x, Compression Level:1) at 0.246 MB/sec and maximum for CASE 5 (JPEG:1, Quality:95, JPEG Subsampling:1x, Compression Level:0) at 0.329 MB/sec.

4.5 Memory usage with 8 VTU files and 8 CPUs

Figure 4 shows the average memory consumed at the client side as 15.877 GB with a standard deviation of 0.332 and a standard error of 0.110. The peak memory was consumed by CASE 9 (JPEG:1, Quality:30, JPEG Subsampling:4x, Compression Level:7) at 23 GB while CASE 5 (JPEG:1, Quality:95, JPEG Subsampling:1x, Compression Level:0) consumed the least memory at 17.7 GB. On an average, CASE 2 (JPEG:0, Quality:95, JPEG Subsampling:1x, Compression Level:1) and CASE 7 (JPEG:1, Quality:95, JPEG Subsampling:1x, Compression Level:5) consumed the maximum memory at 16.3 GB each while CASE 3 (JPEG:0, Quality:95, JPEG Subsampling:1x, Compression Level:5) consumed the least memory at 15.4 GB. On the server side, there was not much difference in the peak memory usage for any test case with CASE 4 (JPEG:0, Quality:95, JPEG Subsampling:1x, Compression Level:6) at 50.85 GB consuming the maximum while CASE 1 (JPEG:0, Quality:95, JPEG Subsampling:1x, Compression Level:0) at 50.65 GB consuming the least.

4.6 Memory usage with 16 VTU files and 16 CPUs

Figure 4 shows the average memory consumed at the client side as 17.077 GB with a standard deviation of 0.456 and a standard error of 0.152. The peak memory was consumed by CASE 8 (JPEG:1, Quality:80, JPEG Subsampling:2x, Compression Level:6) at 22.4 GB while CASE 1 (JPEG:0, Quality:95, JPEG Subsampling:1x, Compression Level:0) and CASE 2 (JPEG:0, Quality:95, JPEG Subsampling:1x, Compression Level:1) at 17.7 GB each consumed the least memory. On an average, CASE 3 (JPEG:0, Quality:95, JPEG Subsampling:1x, Compression Level:5) at 18.1 GB consumed the maximum memory while CASE 6 (JPEG:1, Quality:95, JPEG Subsampling:1x, Compression Level:1) at 16.3 GB consumed the least memory. On the server side, there was not much difference in the peak memory usage for any test case with CASE 2 and CASE 7 (JPEG:1, Quality:95, JPEG Subsampling:1x, Compression Level:5) at 108.5 GB each consuming the maximum while CASE 5 (JPEG:1, Quality:95, JPEG Subsampling:1x, Compression Level:0) at 108.39 GB consuming the least.

5 DISCUSSION

5.1 Welch Two Sample t-tests

We had several combination of variables that we varied and got different results. In order to understand which of these results are significant and dependent on the variables, we ran 8 Welch Two Sample t-tests. We created two groups as shown in Table 2. The t-tests were run for 8 variables - a) average data received; b) maximum data received; c) average data sent; d) average memory on client; e) maximum memory on client; f) maximum memory on the server; g) average still render frame rates; and h) average interactive render frame rates.

Table 2: Groups created for Welch Two Sample t-tests

Group	# Processors	# VTU files
I	8	8
II	16	16

Table 3: Significant results obtained from the t-tests

Variable name	p-value
Avg. data received	0.005267
Avg. data sent	1.96E-14
Avg. still render frame rate	4.14E-15
Avg. interactive render frame rate	2.03E-08

Only 4 out of the 8 p-values that we got were significant as shown in Table 3. This means that the other 4 variables (max. data received, avg. memory on the client, max. memory on the client, and max. memory on the server) were not affected by changing the number of processors and VTU files.

5.2 ANOVA Analysis

To further analyze the effects of the different compression parameters on the variables, we created sub-groups as shown in Table 4 and ran 8 ANOVAs for each of the 8 variables. The results we got are shown in Table 6. All the results except for maximum memory on the client and maximum data received were affected by the number of processors and the VTU files. We can see from the results that the number of processors and the VTU files were the main factors affecting the variables and there was no variable that was affected if JPEG was enabled or disabled. Within each of the two main groups (8 and 16 processors and VTU files), we created sub-groups and used 3 predictors: JPEG, Quality, and Chrominance (see Table 5). ANOVA did not show significant results for any variable except for one as shown in Table 7.

Table 4: Groups created for ANOVA for the entire data

Group	# Processors and VTU files	JPEG
I	8	Disabled (0)
II	8	Enabled (1)
III	16	Disabled (0)
IV	16	Enabled (1)

Table 5: Sub-groups created for ANOVA

Group	JPEG Compression	Quality	Chrominance
I	Disabled (0)	95	1
II	Enabled (1)	95	1
III	Enabled (1)	80	2
IV	Enabled (1)	30	4

Table 6: Significant results obtained from ANOVA for all the data

Variable name	Significant factor	p-value
Avg. data received	Processors and VTU files	0.01209
Avg. data sent	Processors and VTU files	3.25E-13
Avg. memory on client	Processors and VTU files	2.68E-05
Max. memory on server	Processors and VTU files	<2e-16
Avg. still render FPS	Processors and VTU files	<2e-16
Avg. interactive render FPS	Processors and VTU files	5.169e-08

Table 7: Significant results obtained from ANOVA of sub-groups

Variable name	Significant factor	p-value
Avg. still render FPS	JPEG	0.04305 for 16 processors and 16 VTU files

5.3 Significant observations

It can be observed from the results obtained above that the average still render frame rate for 8 CPUs and 8 VTU files was significantly better than those for 16 CPUs and 16 VTU files by a factor greater than 8. This is also shown by the t-tests. The interactive render frame rate for 8 CPUs and 8 VTU files was also significantly better than those for 16 CPUs and 16 VTU files by a factor greater than 1.6. Based on the ANOVA analysis, the most significant factor deciding the frame rates was the number of CPUs and VTU files and if JPEG compression was enabled. Quality and JPEG chrominance sub-sampling did not have any significant impact on the dependent variables.

The average data received by the client for 8 CPUs and 8 VTU files was significantly more than those for 16 CPUs and 16 VTU files by a factor almost equal to 2. The peak data received by the client for 8 CPUs and 8 VTU files was almost the same as for 16 CPUs and 16 VTU files.

The average memory consumed by the client for 8 CPUs and 8 VTU files was slightly less than that consumed for 16 CPUs and 16 VTU files by a factor almost equal to 1.11. The peak memory consumed by the client for 8 CPUs and 8 VTU files was almost equal to that consumed for 16 CPUs and 16 VTU files. The peak memory consumed by the server for 8 CPUs and 8 VTU files was significantly less than that consumed for 16 CPUs and 16 VTU files by a factor almost equal to 2.

5.4 Effect of network and stereo

All the results that we got used the VT-RNET 10 Gbps network, which meant that if we tried to upload or download a single file, we could get speeds almost equal to 256 MB/sec (we would get higher speeds with parallel downloads or uploads). However, had we been using a 1 Gbps connection, we would have got upload and download speeds in the range of 5 MB/sec (we would get higher speeds with parallel downloads or uploads). Generally speaking, the network speeds often have a bottleneck with hard disk throughput, bus contention, frame size, packet size distribution among other reasons. The download and upload speeds that we are reporting in this paper is based on experimental data collected in the lab. In our experiments run using ParaView, the peak data transfer speed achieved was 52.7 MB/sec, which means that if we were using the 1 Gbps network to run our experiments, our results would have been affected significantly.

All our experiments were run in a monoscopic mode using 8 processors or 16 processors. As we observed in our results, the frame rates achieved with 8 processors was almost double the frame rates achieved with 16 processors and this is mainly because of how ParaView duplicates the data on each node (see section 6.1 for details). Therefore, we believe that if we were to do stereoscopic data visualization in the CAVE that would mean defining 2x number of DISPLAYs in the pvx file (one for the left eye and one for the right eye) and requesting 2x number of processors for rendering, and because of how ParaView duplicates data on each node, adding more processors should adversely affect performance.

6 LIMITATIONS

6.1 Big data in CAVE

Our initial goal was to visualize all the five billion points in the CAVE and test the different compression parameters. But we encountered an issue with ParaView and the MPI implementation when we tried to load big data sets. More research into this issue showed that there are two MPI routines that are used for sending and receiving data used in ParaView - MPI_Send and MPI_Recv. The return type for both of these routines is an integer meaning that the return value can not be greater than 2^31 - 1, which is equal to 2 GB. This essentially meant that whenever MPI will try to send or receive data greater than 2 GB, it will crash. ParaView in a CAVE mode duplicates data on each node and there is no compositing. This duplication of data does not happen in a non-CAVE mode and hence, bigger datasets can be visualized. The reason given by the ParaView developers for the duplication of data in the CAVE mode is to increase its efficiency. There are at least two solutions that can potentially be applied in the ParaView source code to fix this issue. If an application wants to send, say, 4 x 2 GB to a peer, instead of sending one big single 8 GB message, the application can send 4 messages each of 2 GB in size. Another workaround is to use “MPI_TYPE_CONTIGUOUS" to create a datatype comprised of multiple elements, and then send multiple of these creating a multiplicative effect (Squyres 2014)

6.2 Stereo rendering in CAVE

With Barco F50 projectors that we used in the lab at Virginia Tech, the active stereo happens at the projector level. It sends two separate video channels, one for the left eye and one for the right eye. ParaView in a CAVE mode does not have an option to define a left eye and a right eye and as of writing this paper, ParaView does not have this functionality in their latest version v5.4. It assumes that there is just one channel coming from the projector that has information for both the eyes. The DISPLAYs set in the ParaView’s pvx file are not meant to be different for different eyes and defines a single eye separation value for the whole configuration. This problem can be solved if we enable the stereo option in the Nvidia configuration file in which case ParaView would have given us a left eye on one display and the right eye on the other and this would have been fed to the projector that would again turn it into active stereo. But we kept the stereo option disabled in the X configuration file since when stereo was enabled it interfered with the Nvidia driver-level blending. This is a known issue with Nvidia drivers for Linux (as of this writing, Nvidia has put it on their to-do list to fix it). The other option to solve this problem as suggested by the ParaView community is to define a left eye and a right eye in the ParaView source code, which would lead to passive stereo (this feature does not exist in the latest ParaView version v5.4).

7 CONCLUSION AND FUTURE WORK

In this work, we remotely rendered a point cloud dataset consisting of more than 40 million points in a CAVE-style system. Our pipeline consisted of ParaView, VirtualGL, and TurboVNC. We varied 6 parameters including the number of processors, number of VTU files, JPEG compression, Quality, JPEG chrominance sub-sampling, and compression level. We measured the data flowing to and from the client to the servers, the memory foot print on the client and the server, and the frame rates received at the client side. We used NewRiver HPC machines at Virginia Tech for remote rendering. Our results show that beyond a point, adding more CPUs and dividing the data into more number of VTU files does not help in speeding the rendering process and visualizing the data in a CAVE-style system. The frame rates achieved with 8 processors and 8 VTU files was significantly better than the frame rates achieved with 16 processors and 16 VTU files.

The work we did in this paper leaves room for some future. It would be interesting to see how other combinations perform like 8 CPUs and 16 VTU files, 16 CPUs and 8 VTU files, etc. Currently, there is a limitation to how much data can be visualized using ParaView in the CAVE but if some more work is done on enhancing ParaView's capability to support bigger data sets by incorporating the changes suggested in section 6.1, it would be interesting to see how ParaView performs and how it affects the network and frame rates. In terms of network speed, we used a 10G network for all our tests, which could easily accommodate the maximum data flowing between the client and the server. This could be one of the reasons why compression parameters like quality and JPEG chrominance sub-sampling did not have any significant impact on the frame rates achieved. It would be interesting to see how the same experiments will behave while using a 1G network. Adding active stereo support to ParaView in addition to the mono mode that it currently supports is also future work. We believe that using stereo will adversely affect frame rates compared to the frame rates achieved in a monoscopic mode

REFERENCES

Ahrens, J., K. Brislawn, K. Martin, B. Geveci, C. C. Law, and M. Papka. 2001. "Large-scale data visualization using parallel data streaming". *IEEE Computer graphics and Applications* vol. 21 (4), pp. 34–41.

Ahrens, J and Geveci, B and Law, C 2005. "ParaView: An End-User Tool for Large Data Visualization. Number ISBN-13: 978-0123875822. Visualization Handbook".

Ahrens, J. P., N. Desai, P. S. McCormick, K. Martin, and J. Woodring. 2007. "A modular extensible visualization system architecture for culled prioritized data streaming". In *Visualization and Data Analysis 2007*, Volume 6495, pp. 64950I. International Society for Optics and Photonics.

Advanced Research Computing 2017. "Newriver Cluster for HPC". https://secure.hosting.vt.edu/www.arc.vt.edu/computing/newriver/. [Online; accessed 11-April-2017].

Cruz-Neira, C., D. J. Sandin, T. A. DeFanti, R. V. Kenyon, and J. C. Hart. 1992. "The CAVE: audio visual experience automatic virtual environment". *Communications of the ACM* vol. 35 (6), pp. 64–72.

Eilemann, S., M. Makhinya, and R. Pajarola. 2009. "Equalizer: A scalable parallel rendering framework". *IEEE transactions on visualization and computer graphics* vol. 15 (3), pp. 436–452.

Elmer FEM 2016. "VTU file format and ParaView for visualization". http://www.elmerfem.org/blog/uncategorized/vtu-file-format-and-paraview-for-visualization. [Online; accessed 17-April-2017].

Humphreys, G., M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. 2002. "Chromium: a stream-processing framework for interactive rendering on clusters". *ACM transactions on graphics (TOG)* vol. 21 (3), pp. 693–702.

Kitware 2017. "ParaView - Parallel Visualization Application". http://www.paraview.org/. [Online; accessed 12-April-2017].

Molnar, S., M. Cox, D. Ellsworth, and H. Fuchs. 1994. "A sorting classification of parallel rendering". *IEEE computer graphics and applications* vol. 14 (4), pp. 23–32.

Moreland, K., and D. Thompson. 2003. "From cluster to wall with VTK". In *Parallel and Large-Data Visualization and Graphics, 2003. PVG 2003. IEEE Symposium on*, pp. 25–31. IEEE.

Moreland, K., B. Wylie, and C. Pavlakos. 2001. "Sort-last parallel rendering for viewing extremely large data sets on tile displays". In *Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, pp. 85–92. IEEE Press.

Open MPI 2017. "Open MPI: Open Source High Performance Computing". https://www.open-mpi.org. [Online; accessed 10-April-2017].

MPICH 2017. "MPICH". https://www.mpich.org. [Online; accessed 10-April-2017].

Moses Palmér 2014. "pynput Package Documentation". https://pynput.readthedocs.io/en/latest/. [Online; accessed 17-April-2017].

Reutebuch, S. E., H.-E. Andersen, and R. J. McGaughey. 2005. "Light detection and ranging (LIDAR): an emerging tool for multiple resource inventory". *Journal of Forestry* vol. 103 (6), pp. 286–292.

Richardson, T., Q. Stafford-Fraser, K. R. Wood, and A. Hopper. 1998. "Virtual network computing". *IEEE Internet Computing* vol. 2 (1), pp. 33–38.

Samanta, R., T. Funkhouser, K. Li, and J. P. Singh. 2000. "Hybrid sort-first and sort-last parallel rendering with a cluster of PCs". In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pp. 97–108. ACM.

Schroeder, W. J., K. M. Martin, and W. E. Lorensen. 1996. "The design and implementation of an object-oriented toolkit for 3D graphics and visualization". In *Proceedings of the 7th conference on Visualization'96*, pp. 93–ff. IEEE Computer Society Press.

Sourceforge 2016. "What is Ganglia?". http://ganglia.sourceforge.net. [Online; accessed 14-April-2017].

Jeff Squyres 2014. "Can I MPI_SEND (and MPI_RECV) with a count larger than 2 billion?". http://blogs.cisco.com/performance/can-i-mpi_send-and-mpi_recv-with-a-count-larger-than-2-billion. [Online; accessed 17-April-2017].

TurboVNC 2017. "A Brief Introduction to TurboVNC". http://www.turbovnc.org/About/Introduction. [Online; accessed 10-April-2017].

VirtualGL 2017. "A Brief Introduction to VirtualGL". http://www.virtualgl.org/About/Introduction. [Online; accessed 12-April-2017].

AUTHOR BIOGRAPHIES

FAIZ ABIDI graduated with a Master's in Computer Science from Virginia Tech in 2017.

NICHOLAS POLYS is the Director of Visual Computing at Virginia Tech Advanced Research Computing and an Affiliate Professor in the School of Engineering and the Department of Computer Science.

SRIJITH RAJAMOHAN is a Computational Scientist at ARC at Virginia Tech.

LANCE ARSENAULT is a visualization and virtual reality specialist at ARC at Virginia Tech.

AYAT MOHAMMED is a postdoctoral fellow at TACC's Scalable Visualization Technologies group.

TOWARD DESIGNING A DYNAMIC CPU CAP MANAGER FOR TIMELY DATAFLOW PLATFORMS

M. Reza HoseinyFarahabady

The University of Sydney, School of IT
Center for Distributed
& High Performance Computing
Australia
reza.hoseiny@sydney.edu.au

Saeed Bastani

Karlstad University
Department of Computer Science
Sweden
saeed.bastani@kau.se

Javid Taheri

Karlstad University
Department of Computer Science
Sweden
javid.taheri@kau.se

Albert Y. Zomaya

The University of Sydney
Center for Distributed
& High Performance Computing
School of IT, Australia
albert.zomaya@sydney.edu.au

Zahir Tari

RMIT University
School of Science
Australia
zahir.tari@rmit.edu.au

Samee U. Khan

North Dakota State University
Dept. of Electrical & Computer Engineering
Fargo, ND, USA
samee.khan@ndsu.edu

ABSTRACT

In this work, we propose a control-based solution for the problem of CPU resource allocation in data-flow platform that considers the degradation of performance caused by running concurrent data-flow processes. Our aim is to cut the QoS violation incidents for applications belonging to the highest QoS class. The performance of the proposed solution is bench-marked with the famous round robin algorithm. The experimental results confirms that the proposed algorithm can decrease the latency of processing data records for applications by 48% compared to the round robin policy.

Keywords: Distributed data-flow processing engines, Scheduling and resource allocation algorithms, Quality of Service

1 INTRODUCTION

Timely data-flow (TDF) is a recent programming model to support application developers with a rich set of primitives for developing distributed and parallel applications that may consist of iterative modules over streaming data sets (Murray et al. 2013). The demand for fast analysis over a large volume of such data led to develop several batch/streaming data engines such as MapReduce (Dean and Ghemawat 2008), Apache Spark (Zaharia et al. 2010), CStream (Şahin 2015), Apache Storm (Apache Software Foundation 2016), IBM System S (Zou et al. 2010), Yahoo S4 (Neumeyer et al. 2010), SCOPE (Chaiken et al. 2008), and Microsoft Sonora (Yang et al. 2012). However, an important missing point of almost all above-mentioned systems is their intrinsic deficiency to brace *iterative* modules over incoming data (Wang 2013), (Clauss and Gustedt 2010), (Dudoladov et al. 2015). Supporting such a feature is almost inevitable for developing most of emerging applications in different domains such as learning algorithms and large-scale graph-based data engines.

The main advantage of timely data-flow model is that it offers all three main benefits of batch and streaming computational models within the same system. Achieving such an ambitious goal is realized by supporting a mechanism that provides application developers for designing independent yet stateful iterative and incremental computational modules. In TDF model, an application can be expressed as a set of several directed (possibly *cyclic*) graphs (DCG) in which the set of graph vertex actually represents the parallel tasks of the application. Such tasks can exchange data elements along the directed edges of the DCG graph. Furthermore, the model label each application's message with a non-conventional logical time-stamp (see (Murray et al. 2013)) for more details) to keep the state of computations.

On the other side, one of the most critical requirements for a data-flow processing engine that runs over a server farm of tens or even hundreds of server nodes is finding an effective way to fulfil the quality of service (QoS) requirements enforced by the application owners. In almost all existing practical solutions, a conventional resource allocation algorithm often picks a new server node only if the overall computing capacity of the current resources cannot cope with unexpected spikes in the incoming workload. Nevertheless, in practical scenarios where the amount of computing capacity is limited, it is almost impossible to avoid violating of QoS requirements for all applications over the course of their execution. A conventional resource allocation strategy, such as round robin or best effort, which is not aware of QoS enforcement level might severely damage the performance level of those applications that belong to the highest QoS level when the capacity of the available resources cannot cope with the incoming demand from all applications. In such cases, a *QoS-aware* resource allocation algorithm is needed to truly comply with the service-level objectives declared in SLA (service level agreement) while maximizing the overall performance of the system.

In this paper, we present an approach for the design as well as a prototype development of a close-loop controller as an elastic solution for allocating CPU shares in a TDF platform. The main key features of the proposed solution is that it considers QoS enforcement level when making resource allocation decisions. We use an ARIMA based prediction model to estimates the future rates for incoming data records per application. Then we apply a simple queue based system model proposed by Kleinrock in (Kleinrock 1976) to estimate the latency of each computational module in the next controlling interval using a history of past observations. In addition, the loop counter of each iteration module is estimated by employing a Monte Carlo sampling method that uses the least possible number of measurements to estimate the unknown value of each loop counter. The experimental evaluation section presents the result of a series of tests that we have conducted as our evaluation study to test the performance of the proposed solution against the round robin policy with respect to both the average response time and the QoS violation rates. Particularly, the proposed solution outperforms the round-robin algorithm by an average improvement of 48% in the overall response time of processing data records, while it reduces the QoS violation incidents by a factor of 3.6 on scenarios that the available computing capacity of server nodes in the local cluster cannot cope with the total demand from all applications.

The rest of the article is organized as follows. In Section 2, we give the background knowledge and related work associated with the timely data-flow platform. Section 3 formally presents the design principle of the proposed resource allocation algorithm. In Section 4, we evaluate the performance of our solution through experiments on real systems. Section 5 briefly discusses the existing work. We then draw our conclusion in Section 6.

2 BACKGROUND

Timely data-flow can be considered as rather new yet evolutionary programming model for designing scalable and fault-tolerant distributed systems first introduced by Microsoft researches (Murray et al. 2013). The programming model can be employed by application developers for deploying parallel programs that runs continuously over a set of streaming data for further processing. Such a streaming processing has been recently receiving a lot of attention, particularly for dealing with the upcoming issues in processing in near real-time fashion.

The new cyclic data-flow model offers all three advantages of prevalent giant models for big-data processing in one system. Particularly, it offers (1) *high throughput* (for batch processing systems), (2) *low latency* (for stream processing engines), and (3) the ability to do *iterative*, *stateful*, and *incremental* computations over incoming streaming data records. In fact, the complexities of combining such features (like nesting loops inside streaming contexts or keeping the state of the computations) in one system can be easily resolved by using timely data-flow model (Murray et al. 2013).

In this model, each message of any computational module is being attached with a logical time-stamp as a lightweight mechanism for coordinating among iterative and incremental processing modules. For the purpose of tracking the progress of the computations, the underlying platform keeps the set of point-stamps of all messages which are still in their execution progress. This helps each parallel worker to realize the set of pending messages that still needs to be delivered to each vertex node at any given time. It also can be used by every working threads to find out if more data records will be arrived in the future epochs (Chandu Thekkath 2017).

The directed graph of might consist of a a data-flow computation consists of a set of input, output, ingress, egress, feedback, and normal computational vertices. The input (output) vertex indicates a stream of data coming (emitting) from the an external producer (to an external consumer). Ingress, egress and feedback vertices identify a loop context (which can be nested within other loop contexts). Such vertices are used to deliver the message in a correct order.

3 THE PROPOSED RESOURCE ALLOCATION SCHEMA

We consider end-to-end response time as the main performance concern of the end-users. We use a model based on queuing theory to allocate/dismiss computing resources to each sub-component of the a timely data-flow application. We also use “control theory” principles to design a robust feedback controller for allocating CPU resources in a dynamic fashion. Our target platform is a cluster that hosts several data-flow applications belong to users with different QoS enforcements.

The key idea of the proposed solution is to leave the decision of allocating CPU resources to the execution time, when it can effectually prevent the issue of capacity bottleneck. The controller measures the following metrics to determine the online state of the system. (1) the incoming traffic rate of each computational module from external source, (2) the available CPU capacity of each host, and (3) the amount of QoS violation incidents occurred so far per each user. Whereas finding an exact relationship among different parts of the system under study might result in an accurate solution, employing a simpler linear difference equation that approximates such a relationship, like the one we use in this paper, is more common in practice. Employing a similar predictive mechanism to control the computing resources in a variety of evnet- and stream-based data processing platforms is not new and has been effectively used by several researchers in the past, such as (Casalicchio et al. 2013), (De Matteis and Mencagli 2016), (DeMatteis and Mencagli 2017), (HoseinyF. et al. 2017a).

We use one buffer per stateful computational module (shown by C_i) to keep track of outstanding messages. By adjusting the percentage of the amount of CPU core cap to be assigned to the corresponding working thread of every computational module, we can respond to the temporal changes in their arrival rates while taking QoS enforcement level into account. Particularly, we approximate the end-to-end delays of each data-flow message by measuring the number of outstanding messages in each buffer at any given time. To this end, the controller uses a prediction tool to estimate the future rate of incoming data elements to each application over a finite-time horizon. Then, based on the current measurements and the predicated future states, the proposed controller takes the near-optimal CPU cap collocation decisions. We will show (in Section 4) that the given solution is robust despite the modelling errors.

We use auto regressive integrated moving average equation (ARIMA) model to estimate the input traffic *rate* of data records coming to each data-flow application within the next controlling interval. According to the ARIMA formula, the future values of a target random variable, λ_k, where k denote the interval index, is estimated based on previous observations of such a random variable as follows.

$$\hat{\lambda}_k = \varepsilon_k + \sum_{\ell=1\cdots h} \beta_\ell \lambda_{k-\ell} + \theta_\ell \varepsilon_{k-\ell} \tag{1}$$

, where ε's are independent and identically distributed error values taken from a normal distribution with mean zero and a finite variance (*e.g.*, a white noise). The coefficients of β and θ are updated using the least-squares regression method right after realizing a new observation.

QoS Guarantee. For the sake of designing a QoS-aware resource allocation, we use the following equations to identify by how extent a resource allocation policy can satisfy the desirable performance metric from the user's perspective Let us assume that there are exactly Q different level of QoS classes from which an end-user can choose its desirable performance level. Each class $1 \leq q \leq Q$ is associated with two values of ω^*_q and $\mathcal{V}_q$ as the maximum acceptable average processing delay and an acceptable upper bound for the percentage of QoS violation incidents can occur for that class, respectively.

For example, let us assume that there are 3 different QoS classes, denoted by $q = 1\cdots 3$. Let us also assume that the associated upper bounds of each class is taken from $\mathcal{V}_{q=1\cdots 3} \in \{0.50, 0.80, 0.99\}$, where the third class (*i.e.*, $q = 3$) has the highest priority. So, the end-to-end response time of processing each data records belong to application from the third class can be just higher than ω^*_3 only for 1% of the whole set of data records during any given period.

Controlling Algorithm. There are at least two key concepts in designing a feedback control for a computing system: (1) the measured output vector, as the system characteristic to be controlled as close as possible as to a desired value (it typically depends on the nature of workload being served); and (2) the control input vector, as the system variables that influence the system output. The input vector is manipulated either by the system administrator or an automatic controlling mechanism to modify the system output. The first step to designing a good feedback control is understanding how the input vector modifies the the measured system output. Further, it is important to realize that the system output value also depends on the system workload, that are normally unknown *a priori* or can change over the execution time (Hellerstein et al. 2004).

Besides to the control input and output vectors, there are other essential elements of a feedback control system as (i) the set-point trajectory, as the desired value of the control output vector, and (ii) the error vector, that is the difference between the set-point trajectory and the measured system output. The main duty of the automatic controller is to find a setting for the control inputs such that the system output follows the set-point trajectory as the reference point. The main advantage of using such a feedback control is that the system administrator just needs to specify the desired output value as the right set-point trajectory instead of direct steering of the input variables which requires substantial expertise set.

The aforementioned queuing model can be successfully applied to handle the issue of service-level guarantees. We regulate the end-to-end response time of each data-flow application by introducing a feedback controller for manipulating the service rate (*i.e.*, the CPU cap) of each application. The implicit objective of such a system is to maximize the number of processed data-flow requests subject to constraints on QoS enforcements (*i.e.*, response times). In our target system, the reference *set-point trajectory* is the desired response time according to QoS enforcements, and the control *input vector* is the CPU cap of all data-flow applications at each controlling interval. The *control error* needs to be calculated by subtracting the average response times from the reference trajectory.

At the beginning of each interval $k = 1, 2, \cdots$, the controlling algorithm accomplishes the following actions.

- It collects the number of pending messages in the buffer of each input module as well as the number of data records that is processed by each computational module during the previous interval.
- The controller predicts the arrival rate of messages per input module for the future step. Using such information, the controller can find out the response time of each application for the next step.
- The controller solve an optimization problem (described later) to find out the best possible CPU cap configuration per application, and then apply such an supposedly optimal solution.
- At time $k+1$, the controller observes the performance of the underlying system as the feedback loop and the above cycle is repeated.

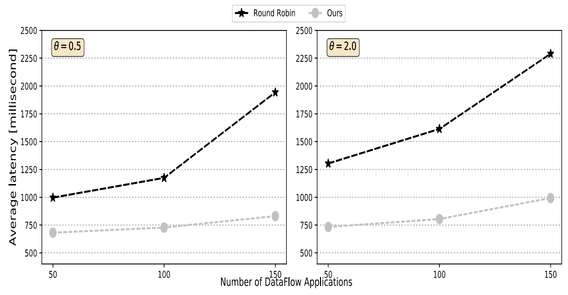
We compute the CPU cap of each application based on the following model (which is obtained from queuing analysis) for the dynamic of a TDF platform. For the data records buffered in each computational unit, we assume a first-come, first-served scheduling policy. So, the data elements depart in the same order in which they arrive to the buffer of each module. We do not assume any further restriction on the inter-arrival time distributions. Let r_k denote the response time (experienced delay) of the k^{-th} data element, τ_k denote the time between the arrival of two consequent data elements at interval k, and s_k denote the service time needed to process such data element. Such variables are related based on the following equation as stated in (Kleinrock 1976).

$$r_{k+1} = (r_k - \tau_{k+1})^+ + s_{k+1} \tag{2}$$

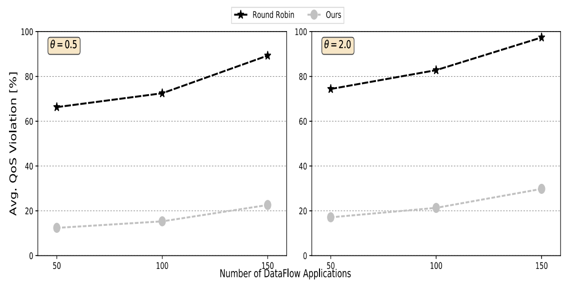
, where $(x)^+$ denote the $\max\{0, x\}$.

If a module C_i resides within a loop context, then we should multiply the average delay obtained by Equation (2) with the mean value of the times that the loop context is executed, shown by $\bar{\eta}_{C_i}$. We use an estimation method based on Monte Carlo sampling algorithm to estimate the mean value of the loop variable for every computational modules. This algorithm, knowns as **AA** Algorithm, is a fully polynomial randomized approximation scheme (FPRAS) that estimates the value of $\bar{\eta}$ by using the *minimum* possible number of measurements (Dagum et al. 2000).

After determining the desirable service time of each application for the CPU credit in the next controlling interval (*i.e.*, s_{k+1}), a central optimization module computes the CPU cap that must be allocated to each data-flow application . The optimization module formulates it as a *capital budgeting problem*, where the *finite budget* is the total amount of CPU share that is available on the server farm, and the *reward* function associated with each data-flow application is the multiplication of the QoS class that the application belongs



(a)



(b)

Figure 1: Improvement in (a) average response time (latency), (b) the percentage of the QoS violation incidents achieved by the proposed controller compared to the static round robin allocation schema as seen by applications belong to the highest priority QoS class.

to and its requested CPU demand. The solution can be found using a dynamic programming approach (for more details see (Powell 2007)).

4 EXPERIMENTAL EVALUATION

The effectiveness of the proposed approach is carried out with respect to QoS violation rate of data-flow applications. We assessed the proposed approach against round robin greedy algorithm that tries to assign CPU share of all accessible physical machines uniformly among data-flow applications.

We use a local cluster consisting of three machines with a total number of 12 logical cores. Each machine is installed with 8 GB of main memory and equipped with a 3.40 GHz Intel i3 CPU. The proposed controller is developed in C++ and Python 2.7 on the top of a modular open-source implementation of timely data-flow in Rust. The controller runs over a dedicated machine equipped with an Intel i7 2.3 GHz CPU with 16GB of RAM and a SSD drive. It can be equally applied to other implementation of timely data-flow, such as Microsoft .NET Naiad system (Microsoft Naiad 2017).

We created $n = \{50, 100, 150\}$ data-flow applications. Each application has four computational modules as a simple loop. Each module runs a dummy script that its running time varies from 0.1 second to 5.0 second with an average of 2.2 second. We also fixed the number of QoS classes to three ($|Q| = 3$) and randomly assign each application to one of the QoS classes. We bind the first computational module of each application to an external emitter which its generation rate is taken from a Poisson process with $\lambda \in [0.5, 2]$, where λ represents the average number of data records generated per 0.1 seconds. The controlling interval length is chosen to be 1 second.

Figure 1 (a) depicts the average response time of the highest QoS class achieved by the proposed controller compared to the one from RR policy when the number of applications increases from 50 to 150 (x axis). The proposed algorithm can reduce the QoS violation incidents, Figure 1 (b), by a factor of 3.6 compared to the RR heuristic on average, too. The experiment also confirms the need for a controlling mechanism in a environment where applications with different QoS levels share the CPU capacity. The main reason is that the static schema equally distributes the CPU shares among applications. Such a fair decision causes a severe QoS violations for applications of higher QoS classes particularity if a burst period exists in which some applications from lowest QoS class consume as much CPU capacity as they need without any restrictions.

The running time of the proposed algorithm to find an optimal solution is negligible (less than 0.01 second) for a scenario with 150 data-flow applications and 3 machines. The reason is that we exploit a dynamic programming schema which can quickly find the optimal solution for the capital budget problem.

We also performed sensitivity analysis of the proposed controller against occurrence of errors in parameter prediction phase. We deliberately injected a prediction error to the generation rate parameter ranging from 10% to 90%, and then collected the relative influence of such an error on the output of the system. To this end, we use the sensitivity coefficient parameter (ψ) as introduced in (HoseinyF. et al. 2017b) to examine how much an output vector varies if there is an estimation error of $\varepsilon_{\mathbf{x}}$ in the input vector $\mathbf{x}$. More precisely,

$$\psi_{\varepsilon,\mathbf{z}} = \frac{|\mathbf{z}(\mathbf{x}) - \mathbf{z}(\mathbf{x} \pm \varepsilon)|}{|\mathbf{z}(\mathbf{x})|}$$

The experimental results confirmed that even an error of 90% in the prediction model has a negligible influence in the solution quality (up to 30% for delay and 13% for the QoS violation avoidance rate). The result showed that the proposed controller is robust enough against the accuracy of the prediction model to be used in real circumstances.

5 RELATED WORK

Several data-flow platforms have been used in different big data applications where their algorithms show iterative nature. Naiad (Murray et al. 2013) has been introduced as one of the earliest attempt to design a distributed system for running parallel and iterative operations over either batch or streaming data records. C-Stream (Şahin 2015) is another elastic streaming processing engine implemented in C++11. Unlike Naiad, operator in C-Stream needs to explicitly request incoming data-tuples from the corresponding ports.

Apache Spark (Zaharia et al. 2010) is a fast, in-memory data processing engine to execute iterative algorithms over the streaming data-sets. It probably has the most similarity with timely data-flow paradigm when compared to other existing frameworks. On the other hand, Cilk (Blumofe et al. 1996) is the most famous work-stealing schedulers for multi-threaded parallelism, where each task produces additional workers by spawning more tasks. In the context of timely data-flow, such a work stealing from other threads is not a straightforward concept to be implemented and may increase interference among threads.

Many existing resource allocation strategies, such as (Li et al. 2014), (Huang et al. 2016), (Padala et al. 2009), (Xiaoyong et al. 2011), (Ballani et al. 2011), (Shen et al. 2011), (Li et al. 2014), (Huang et al. 2016), manage resources based on OS level metrics, such as per core utilization, I/O capacities, and energy usage of resources while ignore the negative performance caused by interference at the shared resources, like LLC or memory bandwidth. Using a control mechanism is not new in computing systems, such as the work in (Mencagli et al. 2014), (Mencagli 2016), (Abdelwahed et al. 2009), (Mencagli et al. 2014), (Mencagli 2016), and (Padala et al. 2009). Most of these works proposed a multi-input, multi-output (MIMO) resource controller that automatically adapts to dynamic changes in a shared infrastructure. Such models try to estimate the complex relationship between the application performance and the resource allocation, and adjusts the embedded model by measuring the clients' response time. While there are similarities between the proposed solution with previous controllers, our solution responds to the degraded performance level by measuring the number of waiting messages and then applying a more accurate queuing based formula to estimate the response time of each application.

6 CONCLUSIONS

We presented a resource allocation strategy for timely data-flow in a shared platform. Timely data-flow is a powerful and general-purpose programming abstraction for creating iterative and streaming computational components. Our aim is to design an elastic resource allocation strategy that continually monitors the important performance metrics of the underlying system and assigns the amount of CPU to DFT applications that belong to different QoS level. The effectiveness of the proposed controller has demonstrated an average improvement in latency of processing data records for all applications by 48% in average compared to the round robin policy.

Future work. We do not consider neither auto-scaling nor migration issues in this project. We realized that the proposed controller has a certain upper bound on achieving its performance when running on a local cluster, particularly when a majority of computing modules suddenly receives a vast amount of incoming data elements. In such cases, the proposed controller needs to be equipped with a migration technique so that it launches more working threads and migrates some computing modules to newly launched threads to avoid QoS violation. Furthermore, we have only compared the controller with round robin algorithm as a classic simple algorithm. The experimental result of this report was quite simple, too, as it only includes a single set of experiments where all examined applications had the same structure, consisting of a single loop, So, the next step can be a comprehensive report for comparing the effectiveness and efficiency of the proposed method with some advanced sophisticated scheduling algorithms, such as (Lai, Fan, Zhang, and Liu 2015), to be validated more extensively under complex experimental settings.

ACKNOWLEDGMENT

We would like to acknowledge the support by Australian Research Council (ARC) for the work carried out in this paper, under Linkage project scheme (LP160100406): *Cloud Resource Allocation under Bursty Conditions for Heavy-Tailed Job*. Samee U. Khan's work is supported by (while serving at) the National Science Foundation. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. We also gratefully acknowledge the anonymous reviewers for their careful reading of our manuscript and their constructive comments which helped us improve and clarify this manuscript.

REFERENCES

Abdelwahed, S. et al. 2009. "On the application of MPC techniques for adaptive performance management of computing systems". *IEEE Trans. on Network & Service Management* vol. 6 (4).

Apache Software Foundation 2016. "Storm, an open source distributed real-time computation system".

Ballani, H., P. Costa, T. Karagiannis, and A. Rowstron. 2011. "Towards predictable datacenter networks". In *SIGCOMM Computer Communication Review*, Volume 41, pp. 242–253. ACM.

Blumofe, R. D., C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. 1996. "Cilk: An efficient multithreaded runtime system". *Journal of parallel and distributed computing* vol. 37 (1), pp. 55–69.

Casalicchio, E., D. A. Menascé, and A. Aldhalaan. 2013. "Autonomic resource provisioning in cloud systems with availability goals". In *Cloud & Autonomic Computing Conf.*, pp. 1. ACM.

Chaiken, R., B. Jenkins et al. 2008. "SCOPE: easy & efficient parallel processing of massive data sets". *Proc. of VLDB Endowment* vol. 1 (2), pp. 1265–1276.

Chandu Thekkath 2017. "Naiad project".

Clauss, P.-N., and J. Gustedt. 2010. "Iterative computations with ordered read–write locks". *J. of Parallel & Distr. Computing* vol. 70 (5), pp. 496–504.

Dagum, P., R. Karp, M. Luby, and S. Ross. 2000. "An optimal algorithm for Monte Carlo estimation". *SIAM Journal on computing* vol. 29 (5), pp. 1484–1496.

De Matteis, T., and G. Mencagli. 2016. “Keep calm & react with foresight: Strategies for low-latency & energy-eff. elastic data stream proc.”. In *Principles & Practice of Parallel Programming*, pp. 13.

Dean, J., and S. Ghemawat. 2008. “MapReduce: simplified data processing on large clusters”. *Communications of the ACM* vol. 51 (1), pp. 107–113.

DeMatteis, T., and G. Mencagli. 2017. “Proactive elasticity and energy awareness in data stream processing”. *J. of Systems & Software* vol. 127, pp. 302–319.

Dudoladov, S., C. Xu, S. Schelter, A. Katsifodimos, S. Ewen, K. Tzoumas, and V. Markl. 2015. “Optimistic recovery for iterative dataflows in action”. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 1439–1443. ACM.

Hellerstein, J. L., Y. Diao, S. Parekh, and D. M. Tilbury. 2004. *Feedback control of computing systems*. John Wiley & Sons.

HoseinyF., M., A. Y. Zomaya, and Z. Tari. 2017a. “A Model Predictive Controller for Managing QoS Enforcements and Microarchitecture-Level Interferences in a Lambda Platform”. *IEEE Transactions on Parallel and Distributed Systems* vol. -.

HoseinyF., M., A. Y. Zomaya, and Z. Tari. 2017b. “QoS- & Contention-Aware Resource Provisioning in a Stream Proc. Engine”. In *Cluster Computing (CLUSTER)*, pp. 137–146. IEEE.

Huang, X., G. Xue, R. Yu, and S. Leng. 2016. “Joint scheduling and beamforming coordination in cloud radio access networks with QoS guarantees”. *IEEE Transactions on Vehicular Technology* vol. 65 (7), pp. 5449–5460.

Kleinrock, L. 1976. *Queueing systems, volume 2: Computer applications*, Volume 66. Wiley-Interscience New York.

Lai, P., R. Fan, W. Zhang, and F. Liu. 2015. “Utility Maximizing Thread Assignment and Resource Allocation”. *arXiv preprint arXiv:1507.01101*.

Li, K., C. Liu, and K. Li. 2014. “An approximation algorithm based on game theory for scheduling simple linear deteriorating jobs”. *Theoretical Computer Science* vol. 543, pp. 46–51.

Mencagli, G. 2016. “Adaptive model predictive control of autonomic distributed parallel computations with variable horizons and switching costs”. vol. 28 (7), pp. 2187–2212.

Mencagli, G., M. Vanneschi, and E. Vespa. 2014. “A cooperative predictive control approach to improve the reconfiguration stability of adaptive distributed parallel applications”. *ACM Trans. on Autonomous & Adaptive Systems (TAAS)* vol. 9 (1), pp. 2.

Microsoft Naiad 2017. “Microsoft Naiad: A Timely Dataflow System”. https://github.com/MicrosoftResearch/Naiad. Accessed: 2017-11-6.

Murray, D. G., F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. 2013. “Naiad: a timely dataflow system”. In *Symposium on Operating Systems Principles*, pp. 439–455. ACM.

Neumeyer, L. et al. 2010. “S4: Distributed stream computing platform”. In *Data Mining Workshops*, pp. 170–177. IEEE.

Padala, P. et al. 2009. “Automated Control of Multiple Virt. Resources”. In *European Conf. on Computer Systems (EuroSys)*, pp. 13–26. NY, ACM.

Powell, W. B. 2007. *Approximate Dynamic Programming: Solving the curses of dimensionality*, Volume 703. John Wiley & Sons.

Şahin, S. 2015. *C-Stream: A coroutune-based elastic stream processing engine*. Ph. D. thesis, bilkent university.

Shen, Z., S. Subbiah et al. 2011. "Cloudscale: elastic resource scaling for multi-tenant cloud systems". In *Symposium on Cloud Computing*, pp. 5. ACM.

Wang, G. 2013. *Automatic Scaling Iterative Computations*. Ph. D. thesis, NY, USA.

Xiaoyong, T., K. Li, Z. Zeng, and B. Veeravalli. 2011. "A novel security-driven scheduling algorithm for precedence-constrained tasks in heterogeneous distributed systems". *IEEE Transactions on Computers* vol. 60 (7), pp. 1017–1029.

Yang, F., Z. Qian, X. Chen, I. Beschastnikh, L. Zhuang, L. Zhou, and J. Shen. 2012. "Sonora: A platform for continuous mobile-cloud computing". *Technical Report. Microsoft Research Asia, Tech. Rep.*

Zaharia, M., M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. 2010. "Spark: Cluster computing with working sets.". *HotCloud* vol. 10 (10-10), pp. 95.

Zou, Q., H. Wang et al. 2010. "From a stream of relational queries to distr. stream processing". *Proc. of the VLDB Endowment* vol. 3 (1-2), pp. 1394–1405.

AUTHOR BIOGRAPHIES

M.REZA HOSEINYFARAHABADY received the BSc degree in computer engineering and the MSc degree in information technology and network engineering both from Sharif University of Technology, Tehran, Iran, in 2005 and 2007, respectively. He received the PhD degree from the School of Information Technologies at the University of Sydney, in 2015. He is currently a research associate in the centre for Distributed and High Performance Computing, School of Information Technologies at the University of Sydney. His research interests include scheduling and resource allocation for parallel and distributed computing systems including Cloud computing and stream-lined dataflow applications, and control systems engineering. His email address is reza.hoseiny@sydney.edu.au.

SAEED BASTANI received his BSc degree (1998) in software engineering from Isfahan University of Technology, Isfahan, Iran. He received a MSc degree (2002) in artificial intelligence and robotics from Iran University of Science and Technology, Tehran, Iran, and his Ph.D. degree (2014) in computer science from University of Sydney, Australia. Prior to his Ph.D. studies, he worked for seven years as a researcher and developer in telecommunication sector. After completing his Ph.D. studies, he worked for six months as an associate researcher, focused on vehicular communication networks, at University of New South Wales, Australia. Since March 2014, he has been a researcher at Lund and Karlstad Universities in Sweden, where he has been involved in European projects in different roles, from research to leadership. His current research interests are performance modeling and simulation of networked systems including wireless networks, datacenter and cloud, Software Defined Networking (SDN), and Network Function Virtualization (NFV). His email address is saeed.bastani@kau.se.

JAVID TAHERI received his BSc and MSc of Electrical Engineering from Sharif University of Technology, Tehran, Iran, in 1998 and 2000, respectively. He received his Ph.D. in the field of Mobile Computing from the School of Information Technologies at the University of Sydney, Australia. Since 2006, he has been actively working in several fields, including: networking, optimization, parallel/distributed computing, and cloud computing. He also holds several cloud/networking related industrial certification from VMware (VCP-DCV, VCP-DT, and VCP-Cloud), Cisco (CCNA), Microsoft, etc. He is currently working as an Associate Professor and a Docent at the Department of Computer Science in Karlstad University, Sweden. His major areas of interest are profiling, modelling and optimization techniques for cloud infrastructures, software defined networking and network function virtualization. His email address is javid.taheri@kau.se.

ALBERT Y. ZOMAYA is currently the Chair Professor of High Performance Computing and Networking and Australian Research Council Professorial Fellow in the School of Information Technologies, The University of Sydney. He is also the Director of the Centre for Distributed and High Performance Computing which was established in late 2009. He is the author/co-author of seven books, more than 500 publications in technical journals and conferences, and the editor of 14 books and 17 conference volumes. He is currently the Editor in Chief of the IEEE Transactions on Computers and serves as an associate editor for 20 journals including some of the leading journals in the field. Prof. Zomaya was the Chair the IEEE Technical Committee on Parallel Processing (1999-2003) and currently serves on its executive committee. He also serves on the advisory board of the IEEE Technical Committee on Scalable Computing, the advisory board of the Machine Intelligence Research Labs. Prof. Zomaya served as General and Program Chair for morethan60eventsand servedonthe committeesofmore than 600 ACM and IEEE conferences. He delivered more than 130 keynote addresses, invited seminars and media briefings. Prof. Zomaya is a Fellow of the IEEE, AAAS, the Institution of Engineering and Technology (U.K.), a Distinguished Engineer of the ACM and a Chartered Engineer (CEng). He received the 1997 Edgeworth David Medal from the Royal Society of New South Wales for outstanding contributions to Australian Science. He is also the recipient of the IEEE Computer Societys Meritorious Service Award and Golden Core Recognition in 2000 and 2006, respectively. Also, he received the IEEE TCPP Outstanding Service Award and the IEEE TCSC Medal for Excellence in Scalable Computing, both in 2011. His email address is albert.zomaya@sydney.edu.au.

ZAHIR TARI is a full professor in Distributed Systems at RMIT University (Australia). He received a bachelor degree in Mathematics from University of Algiers (USTHB, Algeria) in 1984, MSc in Operational Research from University of Greno- ble (France) in 1985 and PhD degree in Com- puter Science from University of Grenoble (France) in 1989. From 1990-1992, Zahir worked at the Database Laboratory at EPFL (Swiss Federal Institute of Technology) as a senior researcher, where he looked at various aspects of distributed database systems. In 1996, he moved to RMIT University as a senior lecturer and currently Professor, where he led the DSN (Distributed Systems and Networking) discipline. Zahir's expertise is in the areas of system performance (e.g., Web servers, P2P, Cloud) and system security (e.g., SCADA, Smart Grid, Cloud). He is the co-author of six books and he has edited over 25 conference proceedings. Zahir is a recipient of over 8M$ in funding from ARC (Australian Research Council) and lately part of a successful 7th Framework AU2EU (Australia to European) bid on Authorisation and Authentication for Entrusted Unions. Zahir was an associate editor of the IEEE Transactions on Computers (TC), IEEE Transactions on Parallel and Distributed Systems (TPDS) and IEEE Magazine on Cloud Computing. His email address is zahir.tari@rmit.edu.au.

SAMEE U. KHAN received a PhD in 2007 from the University of Texas, Arlington, TX, USA. Currently, he is a Program Director at the National Science Foundation, where he is responsible for the Smart & Autonomous Systems program, Critical Resilient Interdependent Infrastructure Systems and Processes program, and Computer Systems Research cluster. He also is a faculty at the North Dakota State University, Fargo, ND, USA. His research interests include optimization, robustness, and security of computer systems. His work has appeared in over 375 publications. He is on the editorial boards of leading journals, such as IEEE Access, IEEE Communications Surveys and Tutorials, IET Wireless Sensor Systems, Scalable Computing, IET Cyber-Physical Systems, and IEEE IT Pro. He is an ACM Distinguished Speaker, an IEEE Distinguished Lecturer, a Fellow of the Institution of Engineering and Technology (IET, formerly IEE), and a Fellow of the British Computer Society (BCS). His email address is samee.khan@ndsu.edu.

CONVERGENCE AND RESILIENCE OF THE FINE-GRAINED PARALLEL INCOMPLETE LU FACTORIZATION FOR NON-SYMMETRIC PROBLEMS

Evan Coleman

Naval Surface Warfare Center
Dahlgren Division
17320 Dahlgren Rd
Dahlgren, VA, USA
ecole028@odu.edu

Masha Sosonkina

Department of Modeling, Simulation
and Visualization Engineering
Old Dominion University
5115 Hampton Blvd, Norfolk, VA, USA
msosonki@odu.edu

ABSTRACT

This paper presents an investigation into the convergence of the fine-grained parallel algorithm for computing an incomplete LU factorization for non-symmetric and indefinite matrices. The fine-grained parallel incomplete LU factorization is a nonlinear fixed point iteration and convergence has not been extensively studied for problems that are not symmetric positive definite. This work investigates the convergence of the algorithm for these more difficult problems and additionally investigates how the occurrence of a computing fault may impact the convergence of the algorithm for the problems being studied. The results obtained suggest that this class of problems presents challenges for the fine-grained parallel incomplete LU factorization (less than 30% of configurations converge naturally), and that while the occurrence of a fault can cause significant negative effects, the simple algorithmic change advocated here can completely ameliorate the effects of a fault.

Keywords: incomplete LU factorizations, preconditioning, fine-grained parallelism, fault tolerance

1 INTRODUCTION

Fine-grained methods are increasing in popularity recently due to their ability to be parallelized naturally on modern co-processors such as GPUs and Intel Xeon Phis. Many examples of recent work using fine-grained parallel methods are available (Chow, Anzt, and Dongarra 2015, Chow and Patel 2015, Anzt, Chow, and Dongarra 2015). A specific area of interest is on techniques that utilize fixed point iteration, i.e., $x = G(x)$ for some vector x and map G. These methods can be computed in either a synchronous or asynchronous manner which helps tolerate latency in high-performance computing (HPC) environments. Looking forward to the future of HPC, it is important to keep in mind the need for developing algorithms that are resilient to faults. On future platforms, the rate at which faults occur is expected to decrease dramatically (Cappello, Geist, Gropp, Kale, Kramer, and Snir 2009, Cappello, Geist, Gropp, Kale, Kramer, and Snir 2014, Asanovic, Bodik, Catanzaro, Gebis, Husbands, Keutzer, Patterson, Plishker, Shalf, Williams, et al. 2006, Geist and Lucas 2009). The fine-grained parallel incomplete LU (FGPILU) factorization (Chow and Patel 2015), which is the focus of this work, is a popular preconditioning method that can be used as a building block for iterative linear-system solvers geared towards novel computing platforms. Typically when working with difficult problems, preconditioning techniques move beyond the fill-in level-based ILU factorizations, of which FGPILU is representative, to threshold-based ones such as ILUT (Saad 2003), or to even to algebraic multilevel preconditioning such as ARMS (Saad and Suchomel 2002).

The FGPILU algorithm is a nonlinear fixed point iteration, and while convergence of the algorithm is guaranteed for some neighborhood around the solution, questions remain regarding the practical performance of the algorithm with respect to difficult problems. For the purposes of this work, "difficult" problems are defined to be those that satisfy any one of the following three criteria: (i) non-symmetric, (ii) not diagonally dominant, or (iii) ill-conditioned (including indefinite matrices). The majority of the work on the algorithm so far has focused on matrices that are symmetric and positive-definite (SPD) (Chow and Patel 2015, Chow, Anzt, and Dongarra 2015, Coleman, Sosonkina, and Chow 2017), and the performance of the algorithm on non-symmetric and indefinite matrices has not been firmly established. Moreover, if the convergence of the algorithm for these classes of problems is less than desirable, they may be more prone to suffer divergence when faced with a fault.

Developing algorithms that are resilient to faults is of paramount importance and fine-grained parallel (fixed point) methods are no exception. Faults can be divided into two categories: hard faults and soft faults (Bridges, Ferreira, Heroux, and Hoemmen 2012). Hard faults cause immediate program interruption and typically come from negative effects on the physical hardware components of the system or on the operating system itself. Soft faults represent all faults that do not cause the executing program to stop and are the focus of this work. Most often, these faults refer to some form of data corruption that is occurring either directly inside of, or as a result of, the algorithm that is being executed. This paper examines the potential impact of soft faults on the fine-grained parallel incomplete LU factorization, and also investigates the use of fine-grained parallel incomplete LU algorithm generated preconditioners on Krylov subspace solvers. The main contributions of this work are analyzing the ability of the FGPILU fixed point iteration to complete successfully when attempting to solve difficult problems under a myriad of different configurations, investigating how the convergence is affected by the occurrence of a soft fault, and demonstrating that the effects of a fault can be mitigated by the simple checkpointing scheme proposed in Coleman, Sosonkina, and Chow 2017. The structure of this paper is organized as follows: In Section 2, a brief summary of some related studies is provided. In Section 3, an overview of fixed-point iterations specific to their use in high performance computing is given. In Section 4, background information is provided for the fine-grained parallel incomplete LU algorithm. In Section 5, a theoretical underpinning of the fine-grained parallel incomplete LU algorithm with respect to its convergence for nonsymmetric or indefinite problems is explored. In Section 6, a series of numerical results are provided, while Section 7 concludes.

2 RELATED WORK

Other work on using (conventional) incomplete LU factorizations for solving difficult problems from various disciplines has been conducted previously, including the more general studies found in Chow and Saad 1997 and Benzi, Haws, and Tuma 2000. The increase in faults for future HPC systems is detailed in several different places (Asanovic, Bodik, Catanzaro, Gebis, Husbands, Keutzer, Patterson, Plishker, Shalf, Williams, et al. 2006, Cappello, Geist, Gropp, Kale, Kramer, and Snir 2009, Cappello, Geist, Gropp, Kale, Kramer, and Snir 2014, Snir, Wisniewski, Abraham, Adve, Bagchi, Balaji, Belak, Bose, Cappello, Carlson, et al. 2014, Geist and Lucas 2009). An initial look into fault tolerance for the FGPILU algorithm is provided in Coleman, Sosonkina, and Chow 2017; the variants of the FGPILU algorithm discussed therein build on ideas from various methods for fault tolerance (Sao and Vuduc 2013, Bridges, Ferreira, Heroux, and Hoemmen 2012, Hoemmen and Heroux 2011). A more general approach for simulating the occurrence of faults is taken in this study. Several recent studies have adopted similar techniques (Coleman and Sosonkina 2016, Coleman, Jamal, Baboulin, Khabou, and Sosonkina 2017, Elliott, Hoemmen, and Mueller 2015, Elliott, Hoemmen, and Mueller 2014, Stoyanov and Webster 2015). The fault model used in this paper is a combination of a modified version of the one initially developed in Coleman and Sosonkina 2016 that was used in Coleman, Sosonkina, and Chow 2017. Details on the fault model used here are provided in Section 6.1. Fine-grained parallel methods, specifically parallel fixed point methods, are an area of increased research activity due to the practical use of these methods on HPC resources. An initial exploration of fault tolerance for the FGPILU factorization studied here is provided in Coleman, Sosonkina, and Chow 2017 and **?**, and an exploration of resilience for stationary iterative linear solvers (i.e., Jacobi) is given in Anzt, Dongarra, and Quintana-Ortí 2015. Fault tolerance for fixed point algorithms has been investigated in Stoyanov and Webster 2015, and a more general exploration of fault tolerance for fine-grained methods is provided in Coleman and Sosonkina 2017. The general convergence of parallel fixed point methods has been explored extensively (Addou and Benahmed 2005, Frommer and Szyld 2000, Bertsekas and Tsitsiklis 1989, Ortega and Rheinboldt 2000, Baudet 1978, Benahmed 2007).

3 FIXED POINT ITERATION

Fixed point iterations are concerned with finding solutions to the iteration $x^{k+1} = G(x^k)$ where $G : \mathbb{R}^n \to \mathbb{R}^n$ is composed of component-wise functionals g_i such that

$$x_1 = g_1(\vec{x}) \ , \ x_2 = g_2(\vec{x}) \ , \ \ldots \ x_n = g_n(\vec{x}) \ ,$$

where the subscript represents the *component*, the iteration superscripts have been removed, and the vector notation is added to emphasize that each individual functional used to update a specific component can (potentially) rely on all other components. In a parallel computing environment, the task of finding the update for an individual (or set of) component(s) can be assigned to an individual processing element. In a system that relies on synchronous updates, the functionals all utilize the same components of $\vec{x}$. In particular, $x_i^{k+1} = g_i(\vec{x}^k)$ for all components $i \in \{1, 2, \ldots, n\}$. On the other hand, in the asynchronous case processors will use the latest information available to them. This will lead to different update patterns for each of the individual functionals, each of which will be utilizing components that are updated a different number of times. The convergence of parallel fixed point iterations is discussed in the literature for both the synchronous (Addou and Benahmed 2005) and asynchronous (Frommer and Szyld 2000) cases among many other sources (Bertsekas and Tsitsiklis 1989, Ortega and Rheinboldt 2000, Baudet 1978, Benahmed 2007).

4 FINE-GRAINED PARALLEL INCOMPLETE LU FACTORIZATION

The fine-grained parallel incomplete LU (FGPILU) factorization approximates the true LU factorization and writes a matrix A as the product of two factors L and U where, $A \approx LU$. Normally, the individual components of both L and U are computed in a manner that does not allow easy use of parallelization. The recent FGPILU algorithm proposed in Chow and Patel 2015 allows each element of both the L and U factors to be computed independently. The algorithm progresses towards the incomplete LU factors that would be found by a traditional algorithm in an iterative manner. To do this, the FGPILU algorithm uses the property $(LU)_{ij} = a_{ij}$ for all (i, j) in the sparsity pattern S of the matrix A, where $(LU)_{ij}$ represents the (i, j) entry of the product of the current iterate of the factors L and U. This leads to the observation that the FGPILU algorithm (given in Algorithm 1) is defined by the following two nonlinear equations:

$$l_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right) , \ u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \ . \tag{1}$$

Following the analysis presented in (Chow and Patel 2015), it is possible to collect all of the unknowns l_{ij} and u_{ij} into a single vector x, then express these equations as a fixed-point iteration, $x^{(p+1)} = G\left(x^{(p)}\right)$, where the function G implements the two nonlinear equations described above. The FGPILU algorithm is given in Algorithm 1. Keeping with the terminology used in Chow and Patel 2015 and Chow, Anzt, and

Algorithm 1: FGPILU algorithm as given in (Chow and Patel 2015).

```
Input: Initial guesses for $l_{ij} \in L$ and $u_{ij} \in U$
Output: Factors $L$ and $U$ such that $A \approx LU$
for $sweep = 1, 2, \ldots, m$ do
    for $(i, j) \in S$ do in parallel
        if $i > j$ then
            $l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}) / u_{jj}$
        else
            $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}$
```

Dongarra 2015, each pass the algorithm makes in updating all of the l_{ij} and u_{ij} elements (alternatively: each element of the vector x) is referred to as a "sweep". After each sweep of the algorithm, the L and U factors progress towards convergence. At the beginning of the algorithm, the factors L and U are set with an initial guess. In this study, the initial L factor will be taken to be the lower triangular part of A and the initial U will be taken to be the upper triangular portion of A (as in Chow and Patel 2015, Coleman, Sosonkina, and Chow 2017, Anzt, Chow, Saak, and Dongarra 2016). Adopting the technique used in Chow and Patel 2015, Chow, Anzt, and Dongarra 2015, Coleman, Sosonkina, and Chow 2017, a scaling of the input matrix A is first performed such that the diagonal elements of A are equal to one. As pointed out in Chow and Patel 2015, this diagonal scaling is able to help maintain reasonable convergence rates for the algorithm, and the working assumption in this paper is that all matrices have been scaled in this manner.

5 CONVERGENCE OF THE FGPILU FACTORIZATION

The analysis to show convergence of the FGPILU algorithm relies on properties of the Jacobian associated with the nonlinear mapping defined by $G : \mathbb{R}^m \to \mathbb{R}^m$ where m represents the number of non-zero terms in the matrix A. To define the Jacobian, an order of the elements in both the L and U factors (which together constitute all of the elements in the vector $\vec{x}$ from Section 3) needs to be defined. In particular, an ordering g will map a pair of (i, j) coordinates specifying the location of a non-zero term in A to an index of the vector x, or to the set $\{1, 2, 3, \ldots, m\}$ where $m = nnz(L) + nnz(U)$. That is,

$$x_{g(i,j)} = \begin{cases} l_{ij} & i > j \\ u_{ij} & i \leq j \ . \end{cases}$$

Given this, the two nonlinear equations that define FGPILU, i.e., Eq. (1), can be rewritten such that,

$$G_{g(i,j)} = \begin{cases} \frac{1}{x_{g(j,j)}} \left(a_{ij} - \sum\limits_{1 \leq k \leq j-1} x_{g(i,k)} x_{g(k,j)} \right) & i > j \\ a_{ij} - \sum\limits_{1 \leq k \leq i-1} x_{g(i,k)} x_{g(k,j)} & i \leq j \ , \end{cases} \tag{2}$$

where both sums are taken over all pairs, (i, k) and $(k, j) \in S(A)$. The Jacobian itself can then be written $G'(x) = J(G(x))$ and is defined by the following equations (Chow and Patel 2015):

$$\begin{aligned}\frac{\partial G_{g(i,j)}}{\partial x_{g(k,j)}} &= -\frac{x_{g(i,k)}}{x_{g(j,j)}}, k < j \\ \frac{\partial G_{g(i,j)}}{\partial x_{g(i,k)}} &= -\frac{x_{g(k,j)}}{x_{g(j,j)}}, k < j \\ \frac{\partial G_{g(i,j)}}{\partial x_{g(j,j)}} &= -\frac{1}{x^2_{g(j,j)}}\left(a_{ij} - \sum_{k=1}^{j-1} x_{g(i,k)}x_{g(k,j)}\right)\end{aligned} \tag{3}$$

for a row in the Jacobian where $i > j$ (i.e., corresponding to an unknown $l_{ij} \in L$). Conversely, for a row $i \leq j$ (i.e., corresponding to an unknown $u_{ij} \in U$), the partial derivatives are given by:

$$\frac{\partial G_{g(i,j)}}{\partial x_{g(i,k)}} = -x_{g(i,k)}, k < i, \qquad \frac{\partial G_{g(i,j)}}{\partial x_{g(k,j)}} = -x_{g(i,k)}, k < i\ . \tag{4}$$

Under the assumption that there is a single fixed point solution x^* of the nonlinear iteration defined by $G(x)$, the following result given in Theorem 1 provides convergence for the nominal FGPILU algorithm:

Theorem 1 (Frommer and Szyld 2000)**.** *Assume that x^* lies in the interior of the domain of G and that G is F-differentiable at x^*. If $\rho(G'(x^*)) < 1$, then there exists some local neighborhood of x^* such that the asynchronous iteration defined by G converges to x^* given that the initial guess is inside of this neighborhood.*

Details of this analysis are provided in Chow and Patel 2015. As pointed out in that paper, one consequence of Theorem 1 is that the algorithm will be successful when the norm of the Jacobian is small. Examining the equations that define the partial derivatives inside of the Jacobian, this implies that the FGPILU algorithm will be effective when the terms on the diagonal are large and the off diagonal terms are small; indicating that the FGPILU algorithm will perform well for matrices that are diagonally dominant.

5.1 Improving the Convergence of the FGPILU Algorithm

In this work, an investigation is made into the performance of the FGPILU algorithm with respect to more difficult problems. For a given problem, the FGPILU algorithm may fail to converge; i.e., the nonlinear residual norm (see Eq. (2)) fails to decrease below a desired threshold or diverges entirely. Additionally, the structure of the input matrix may preclude unmodified use of the FGPILU algorithm (e.g., due to zeros on the diagonal). If the progression of the algorithm reaches a point where the norm of the Jacobian is greater than one, the fixed point iteration no longer represents a (local) contraction and further sweeps will not help the algorithm make progress towards the desired preconditioning factors. Even if the FGPILU algorithm converges to a set of preconditioning factors, it is possible that, if the system was changed too much to ensure convergence, the preconditioning factors will not aid in the convergence of the associated Krylov solver. In fact, it is possible for the resulting L and U factors to actually slow convergence or prevent convergence entirely (see both Table 2 and (**?**)). In an effort to improve the convergence of the FGPILU algorithm, this study focuses on employing two techniques; both aim to increase the diagonal dominance of the original matrix, which will in turn reduce the norm of the Jacobian and help ensure that the fixed point iteration continues to make progress.

The first technique involves reordering the matrix in order to aid the convergence of the algorithm. Three reorderings are considered here. The first is the `MC64` reordering that attempts to permute the largest entries of the matrix to the diagonal (**?**), the second is the approximate minimum degree (`AMD`) as implemented in `MATLAB` since it has previously been observed to help convergence of the FGPILU algorithm on non-symmetric problems (Chow and Patel 2015, Benzi, Haws, and Tuma 2000), the other is the Reverse Cuthill-Mckee ordering (`RCM`) which attempts to reduce the bandwidth of the matrix which can potentially aid in the convergence of the FGPILU algorithm and has shown to be effective in the case of symmetric, positive-definite (SPD) matrices (Chow, Anzt, and Dongarra 2015, Chow and Patel 2015, Coleman, Sosonkina, and Chow 2017). After the ordering is applied, the second technique consists of an α-shift that is performed in the manner originally suggested in **?**. Specifically, the original input matrix A can be written, $A = D - B$, where D holds only the diagonal elements of A, and B contains all other elements. Instead of performing the incomplete LU factorization on the original matrix A, the factorization is instead applied to a matrix that is close to A but has an increased level of diagonal dominance. In particular, the incomplete LU factorization can be applied to $\hat{A} = (1+\alpha)D - B$, where $\hat{A} \approx A$ but the size of the diagonal has been increased. This α-shift technique has been used historically for improving the stability of the preconditioning factors generated by conventional incomplete LU factorizations, but given the discussion above in Section 5 concerning the fine-grained incomplete LU factorization that is the subject of this work, it is reasonable to expect this shift to improve the convergence of the FGPILU algorithm. Moreover, since incomplete LU factorizations are by nature, approximate, using the preconditioning factors obtained from applying the FGPILU algorithm to $\hat{A}$ before a Krylov solve of the original matrix A can be expected to accelerate the overall convergence for reasonable values of α. These claims will be explored numerically through the remainder of the paper.

To track the progression of the FGPILU algorithm, a common metric to monitor the progression of the FGPILU algorithm is the so-called nonlinear residual norm (Chow and Patel 2015, Chow, Anzt, and Dongarra 2015, Coleman, Sosonkina, and Chow 2017). This is a value

$$\tau = \sum_{(i,j)\in S} \left| a_{ij} - \sum_{k=1}^{\min(i,j)} l_{ik}u_{kj} \right| , \tag{5}$$

where the set S contains all of the non-zero elements in both L and U, i.e., the non-zero pattern on which the incomplete LU factorization is sought. The nonlinear residual norm decreases as the number of sweeps progresses and the factors produced by the algorithm become closer to the conventional L and U factors that would be computed by a traditional incomplete LU factorization. Because of this it is possible to detect convergence of the algorithm if the nonlinear residual norm is reduced by some pre-specified order of magnitude. However, it should be noted that L and U factors are capable of being used successfully as preconditioning factors (Chow and Patel 2015), and that in practice it may not be necessary to perform very many sweeps of the FGPILU algorithm.

For the purposes of this study, a single fault tolerant variant of the FGPILU algorithm from Coleman, Sosonkina, and Chow 2017 is tested. The variant of the algorithm that was selected monitors the progression of the nonlinear residual norm τ and rejects updates if the update causes the nonlinear residual norm to increase. In particular, the following condition is checked: $\tau^{(sweep)} > \gamma \cdot \tau^{(sweep+r)}$, where the parameter γ controls how monotonic the convergence of the nonlinear residual norm is expected to be, and the parameter r controls how long to delay (in terms of the number of sweeps) in between performing the check.

6 NUMERICAL RESULTS

The test problems that were used in this study are intended to form a representative but not complete set of matrices that are harder to solve than the simpler SPD problems that have been utilized previously. The convergence of the fixed point iteration associated with the FGPILU algorithm displays good convergence with this type of problem (Coleman, Sosonkina, and Chow 2017, Chow, Anzt, and Dongarra 2015, Chow and Patel 2015). However, solving fixed point iterations that feature nonlinear functionals (i.e., in Algorithm 1) is often difficult, and developing the associated convergence theory is also typically hard to accomplish (see for example: Bertsekas and Tsitsiklis 1989, Ortega and Rheinboldt 2000).

The experimental setup for this study is an NVIDIA Tesla K80 GPU on the Turing high performance cluster at Old Dominion University. The nominal, fault-free iterative incomplete factorization algorithms and iterative solvers were taken from the MAGMA open-source software library (Innovative Computing Lab 2015), and minimal modifications were made to the existing MAGMA source code in order to implement the modifications to the FGPILU algorithm, add the α-shift, and to inject faults into the algorithm. All of the results provided in this study reflect double precision, real arithmetic. The test matrices used here come from a variety of sources. The first comes from the seminal work on the performance of incomplete LU factorization for indefinite matrices (Chow and Saad 1997), `fs_760_3`. The next matrix comes from the domain of circuit simulation, `ecl32`, and has been studied previously (**?**, **?**). The last matrix comes from the set of 8 SPD matrices that were studied in previous works on the FGPILU algorithm (Chow, Anzt, and Dongarra 2015, Coleman, Sosonkina, and Chow 2017, Chow and Patel 2015), and is the matrix among those eight with the largest condition number (as estimated by `MATLAB`'s `CONDEST` function); 'offshore'. Condition numbers for the 8 previously studied SPD problems range from 1.11e+03 to 2.24e+13. A brief summary of all three matrices is provided in Table 1. The matrices that are presented here attempt to give

Table 1: Characteristics of the matrices used: Column `Sym?` reflects the symmetry, `PD?` provides positive-definiteness, `Dim`—number of rows, and `Non-zeros`–number of non-zeros in each matrix.

Matrix Name	Abbr.	Sym?	PD?	`CONDEST`	Dim.	Non-zeros	Description
fs_760_3	FS	N	N	9.93E+19	760	5,816	chemical engineering
ecl32	ECL	N	N	9.41E+15	51,993	380,415	circuit simulation
OFFSHORE	OFF	Y	Y	2.24E+13	259,789	4,242,673	electric field diffusion

some indication as to the performance of the nonlinear fixed point iteration associated with the FGPILU with respect to matrices that are more challenging computationally than the problems that are featured in the majority of the previous work on the algorithm (i.e. Chow and Patel 2015, Chow, Anzt, and Dongarra 2015, Coleman, Sosonkina, and Chow 2017).

6.1 Fault Model

In this study, faults are modeled as perturbations similar to several recent studies (Coleman and Sosonkina 2016, Coleman, Sosonkina, and Chow 2017, Stoyanov and Webster 2015); the goal being producing fault tolerant algorithms for future computing platforms that are not too dependent on the precise mechanism of a fault (e.g. bit flip). Modifying the perturbation-based fault model described in Coleman, Sosonkina, and Chow 2017, a single data structure is targeted and a small random perturbation is injected into each component transiently. For example, if the targeted data structure is a vector x and the maximum size of the perturbation-based fault is ϵ, then proceed as follows: generate a random number $r_i \in (-\epsilon, \epsilon)$, and then set $\hat{x}_i = x_i + r_i$ for all values of i. The resultant vector $\hat{x}$ is, thus, perturbed away from the original vector x. In this study, faults are injected into the FGPILU algorithm following the perturbation based methodology described above. Due to the relatively short execution time of the FGPILU algorithm on the given test problems, a fault is induced only once during each run, and the fault is designated to occur at a random sweep number before convergence. Three ranges for faults injected by the perturbation-based model were considered: $r_i \in (-0.01, 0.01), r_i \in (-1, 1)$, and $r_i \in (-100, 100)$. Due to the non-deterministic nature of the fault model, multiple runs are conducted for each value of r_i and the data is averaged in order to obtain representative results.

6.2 Convergence of the FGPILU fixed point iteration and associated Krylov solver

In these fault-free experiments, the convergence of the FGPILU algorithm is examined for three different levels (0,1, and 2) of the incomplete LU factorization (see Benzi 2002 or Saad 2003 for a clear description of levels of incomplete LU factorizations), and three different values of α in the α-shift. Note that regardless of the ordering being utilized, all runs start with a symmetrically scaled matrix such that the entries on the diagonal are less than or equal to 1. As such, appropriate values for α range from 0 to 1 and in this study three discrete values were selected: 0, 0.5, 1.0. More extreme values for α can help improve the convergence of the FGPILU algorithm by increasing the diagonal dominance of the matrix that the FGPILU algorithm is applied to, but this comes at the expense of preparing the preconditioner for a problem increasingly less related to the original problem. As an example, for the `OFFSHORE` problem with `AMD` ordering and symmetric scaling, the FGPILU algorithm converges in a smaller number of sweeps for increasing values of α, but the overall performance of the Krylov subspace solver deteriorates. Details are provided in Table 2.

Table 2: Effects of increasing α for the `OFFSHORE` problem.

α	FGPILU Sweeps	Krylov solver iterations	Krylov solver time
0	24	30	24.8067
1	9	56	46.4995
10	5	144	130.0958

For each of the three matrices that were tested: four orderings were tested (`MC64, AMD, RCM`, and the natural ordering), 3 level of ILU fill-in were tested (levels 0, 1, and 2), and 3 factors for α were used (0, 0.5, and 1.0). This leads to a total of 108 permutations to test. Of these 108 combinations, 84 (77.78%) led to a case were the FGPILU algorithm converged, but only 29 (26.85%) resulted in a successful GMRES solve of the entire linear system using a restart parameter of 50 and a tolerance of 1e-10. Details for those 29 cases are provided below in Table 3.

Table 3: Successful runs with their parameter combinations.

Matrix	**Ordering**	α	**ILU Level**	**Sweeps**	**Krylov Its.**	**Time (s)**
offshore	AMD	0	0	19	30	18
offshore	AMD	0.5	0,1,2	10,11,11	40,34,34	24,55,144
offshore	AMD	1	0,1,2	8,9,9	56,54,54	34,96,229
offshore	RCM	0	0	19	19	35
offshore	RCM	0.5	0,1,2	10,11,11	37,34,34	68,306,771
offshore	RCM	1	0,1,2	9,9,9	54,54,54	101,484,1226
offshore	Natural	0	0	22	22	84
offshore	Natural	0.5	0,1,2	11,12,12	38,34,34	146,312,695
offshore	Natural	1	0,1,2	9,10,10	54,54,54	210,491,1104
ecl32	AMD	0	2	15	127	104
ecl32	RCM	0	2	24	9	39
ecl32	Natural	0	2	18	11	16
fs_760_3	AMD	0	2	55	3	0.4
fs_760_3	RCM	0	1,2	52,63	2,2	0.4,0.4
fs_760_3	MC64	0	1	16	3	0.3
fs_760_3	Natural	0	1	16	3	0.3

In general, higher levels of fill are capable of producing better preconditioning factors (Benzi, Haws, and Tuma 2000, Chow and Saad 1997), but come at the cost of increased storage and computational costs. There is an inherent trade-off in using higher fill levels to produce incomplete factors that are closer to the full L

and U factors that must be evaluated. A few other general observations: the two non-symmetric problems tend to perform better with smaller values of α and higher levels of fill-in allowed, and the level of ILU fill-in tends to not have as much of an impact on whether or not the problem can be solved when compared to the ordering or value for α, but affects the performance. In the results found here, the benefit of having more complete L and U factors from going to a higher fill-in level tends to be outweighed by the increased computational cost of the fixed-point iteration associated with the FGPILU algorithm for a drastically larger number of elements. As an example of the drastic increase in the number of non-zero elements for each of the matrices, consider the data in Table 4.

Table 4: Increase in non-zeros for different levels of ILU fill-in.

Matrix	**nnz(ILU-0)**	**nnz(ILU-1)**	**nnz(ILU-2)**
offshore	4.502E+06	9.974E+06	2.172E+07
ecl32	4.324E+05	9.473E+05	1.954E+06
fs_760_3	6.576E+03	1.757E+04	3.231E+04

6.3 Resilience of the FGPILU fixed point iteration

The experiments conducted in this section reflect the resilience of the FGPILU algorithm with respect to transient soft faults. Resilience is provided by checkpointing and restoring prior data based on the progression of the nonlinear residual norm. To illustrate the resilience of the FGPILU algorithm, only combinations of ordering, ILU-level, and α from Section 6.2 that were successful in the fault-free scenario have been selected for experimentation. A single set of parameters for the checkpointing check, $\tau^{(sweep)} > \gamma \cdot \tau^{(sweep+r)}$, is used. Both γ and r were set to one so that a strict check on the monotonicity of the nonlinear residual norm is performed after every sweep. For SPD problems, this level of check may be unnecessary (Coleman, Sosonkina, and Chow 2017), but this provides the maximum level of protection for the FGPILU algorithm and provides a measure of how effective this check can be for the more difficult problems under investigation in this study. A summary of the data found in these experiments is provided in Table 5, which depicts the

Table 5: Solver performance using FGPILU with no fault tolerance (`NoFT`) and checkpointing (`CP`) .

Scenario	Success Rate (NoFT)	Success Rate (CP)	Timing Ratio	Sweeps Ratio	Its. Ratio
Total	46.65%	100.00%	1.02	0.63	1.01
Small fault	88.59%	100.00%	1.03	0.69	1.03
Medium fault	42.94%	100.00%	1.01	0.48	1.00
Large fault	14.71%	100.00%	1.00	0.73	0.99

percentage of runs that succeeded—resulted in a successful linear system solve—subject to faults (column `Scenario`), when no fault tolerance (column `NoFT`) and the checkpointing FGPILU variant (column `CP`) were employed, respectively. Three ratios of the results with `CP` and `NoFT` are shown in Table 5 as `Timing`, `Sweeps`, and `Its`, defining the timing increase, reduction in the total number of sweeps needed, and the change in the GMRES iterations, respectively. The checkpointing algorithm mitigates well the potential impact of a fault. Note that the largest benefit comes from correcting the impact of a large fault. Smaller faults—which cause effects similar to those produced by bit flips in a less significant bit of the mantissa—tend to be corrected naturally by the iterative nature of the fixed-point iteration. Another important factor in comparing any fault tolerance methods is quantifying how much overhead they introduce. Due to the non-deterministic block-asynchronous nature of the GPU implementation of the FGPILU algorithm in the absence of faults and the inherent randomness involved in the fault model utilized in this study, it is difficult to compare individual cases. However, comparing runs utilizing the same parameters over all cases where both the fault-free variants and the checkpointing variant solved the linear system successfully, there is about a 2% increase in the time required to reach a solution in order to provide fault tolerance to the

FGPILU algorithm using this methodology. There is more of an impact on cases with small faults since it is often possible for the iterative nature of the algorithm to correct the impact of a sufficiently small fault. Note that varying the parameters γ and r that determine the frequency and strictness of the check could change both the efficiency and efficacy of the checkpointing variant of the FGPILU.

7 CONCLUSION & FUTURE WORK

This study has presented some experiments and analysis concerning the convergence of the fine-grained parallel incomplete LU factorization proposed in Chow and Patel 2015 with respect to more difficult problems than have previously been studied. Additionally, initial work concerning the resilience of the FGPILU algorithm with respect to transient soft faults has been explored for this same difficult problem set. Moving forward, further adaptations to the FGPILU algorithm are possible. This series of experiments has worked with various levels of ILU factorization, while a variant of ILUT that takes advantage of fine-grained parallelism called ParILUT (**?**) has been proposed recently, and previous work on the performance of incomplete factorizations for the solution of indefinite and non-symmetric problems (Chow and Saad 1997, Benzi, Haws, and Tuma 2000) has indicated that different styles of incomplete factorization may be more effective for these classes of problems. This suggests that future work on fine-grained preconditioners should include recent developments in fine-grained factorizations when possible. It would be also helpful to compare the performance of various solvers, such as Bi-CGSTAB and TFQMR, with the preconditioning factors that are generated by the FGPILU algorithm. Another avenue for future exploration includes expanding the experimentation conducted with respect to various fault tolerance techniques. In particular, the checkpointing scheme proposed here could be further analyzed in an attempt to optimize the amount of computational overhead induced as opposed to the increase in convergence rate.

ACKNOWLEDGMENTS

This work was supported in part by the Air Force Office of Scientific Research under the AFOSR award FA9550-12-1-0476, by the U.S. Department of Energy (DOE), Office of Advanced Scientific Computing Research Exascale Computing Project, under the grant DE-SC-0016564 through the Ames Laboratory, operated by Iowa State University under contract No. DE-AC00-07CH11358, by the U.S. Department of Defense High Performance Computing Modernization Program, through a HASI grant, and through the ILIR/IAR program at the Naval Surface Warfare Center - Dahlgren Division. This work was also supported by the Turing high performance computing cluster at Old Dominion University.

REFERENCES

Addou, A., and A. Benahmed. 2005. “Parallel synchronous algorithm for nonlinear fixed point problems”. *International Journal of Mathematics and Mathematical Sciences* vol. 2005 (19), pp. 3175–3183.

Anzt, H., E. Chow, and J. Dongarra. 2015. “Iterative sparse triangular solves for preconditioning”. In *European Conference on Parallel Processing*, pp. 650–661. Springer.

Anzt, H., E. Chow, J. Saak, and J. Dongarra. 2016. “Updating incomplete factorization preconditioners for model order reduction.”. *Numerical Algorithms* vol. 73 (3), pp. 611–630.

Anzt, H., J. Dongarra, and E. S. Quintana-Ortí. 2015. “Tuning stationary iterative solvers for fault resilience”. In *Proceedings of the 6th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pp. 1. ACM.

Asanovic, K., R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams et al. 2006. “The landscape of parallel computing research: A view from Berkeley”. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

Baudet, G. M. 1978. “Asynchronous iterative methods for multiprocessors”. *Journal of the ACM (JACM)* vol. 25 (2), pp. 226–244.

Benahmed, A. 2007. “A convergence result for asynchronous algorithms and applications”. *Proyecciones (Antofagasta)* vol. 26 (2), pp. 219–236.

Benzi, M. 2002. “Preconditioning techniques for large linear systems: a survey”. *Journal of computational Physics* vol. 182 (2), pp. 418–477.

Benzi, M., J. C. Haws, and M. Tuma. 2000. “Preconditioning highly indefinite and nonsymmetric matrices”. *SIAM Journal on Scientific Computing* vol. 22 (4), pp. 1333–1353.

Bertsekas, D. P., and J. N. Tsitsiklis. 1989. *Parallel and distributed computation: numerical methods*, Volume 23. Prentice hall Englewood Cliffs, NJ.

Bridges, P., K. Ferreira, M. Heroux, and M. Hoemmen. 2012. “Fault-tolerant linear solvers via selective reliability”. *arXiv preprint arXiv:1206.1390*.

Cappello, F., A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. 2009. “Toward exascale resilience”. *The International Journal of High Performance Computing Applications* vol. 23 (4), pp. 374–388.

Cappello, F., A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. 2014. “Toward exascale resilience: 2014 update”. *Supercomputing frontiers and innovations* vol. 1 (1).

Chow, E., H. Anzt, and J. Dongarra. 2015. “Asynchronous iterative algorithm for computing incomplete factorizations on GPUs”. In *International Conference on High Performance Computing*, pp. 1–16. Springer.

Chow, E., and A. Patel. 2015. “Fine-grained parallel incomplete LU factorization”. *SIAM journal on Scientific Computing* vol. 37 (2), pp. C169–C193.

Chow, E., and Y. Saad. 1997. “Experimental study of ILU preconditioners for indefinite matrices”. *Journal of Computational and Applied Mathematics* vol. 86 (2), pp. 387–414.

Coleman, E., A. Jamal, M. Baboulin, A. Khabou, and M. Sosonkina. 2017. “A Comparison and Analysis of Soft-Fault Error Models using FGMRES and ARMS RBT”. In *Proceedings of the 12th International Conference on Parallel Processing and Applied Mathematics*. ACM.

Coleman, E., and M. Sosonkina. 2016. “Evaluating a Persistent Soft Fault Model on Preconditioned Iterative Methods”. In *Proceedings of the 22nd annual International Conference on Parallel and Distributed Processing Techniques and Applications*.

Coleman, E., and M. Sosonkina. 2017. “Fault Tolerance for Fine-Grained Iterative Methods”. In *Proceedings of the 7th annual Virginia Modeling, Simulation, and Analysis Center Capstone Conference*. Virginia Modeling, Simulation, and Analysis Center.

Coleman, E., M. Sosonkina, and E. Chow. 2017. “Fault Tolerant Variants of the Fine-Grained Parallel Incomplete LU Factorization”. In *Proceedings of the 2017 Spring Simulation Multiconference*. Society for Computer Simulation International.

Elliott, J., M. Hoemmen, and F. Mueller. 2014. “Evaluating the impact of SDC on the GMRES iterative solver”. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 1193–1202. IEEE.

Elliott, J., M. Hoemmen, and F. Mueller. 2015. “A Numerical Soft Fault Model for Iterative Linear Solvers”. In *Proceedings of the 24nd International Symposium on High-Performance Parallel and Distributed Computing*.

Frommer, A., and D. B. Szyld. 2000. “On asynchronous iterations”. *Journal of computational and applied mathematics* vol. 123 (1), pp. 201–216.

Geist, A., and R. Lucas. 2009. “Major computer science challenges at exascale”. *International Journal of High Performance Computing Applications*.

Hoemmen, M., and M. Heroux. 2011. “Fault-tolerant iterative methods via selective reliability”. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC). IEEE Computer Society*, Volume 3, pp. 9. Citeseer.

Innovative Computing Lab 2015. “Software distribution of MAGMA”. http://icl.cs.utk.edu/magma/.

Ortega, J. M., and W. C. Rheinboldt. 2000. *Iterative solution of nonlinear equations in several variables*. SIAM.

Saad, Y. 2003. *Iterative methods for sparse linear systems*. SIAM.

Saad, Y., and B. Suchomel. 2002. “ARMS: An algebraic recursive multilevel solver for general sparse linear systems”. *Numerical linear algebra with applications* vol. 9 (5), pp. 359–378.

Sao, P., and R. Vuduc. 2013. “Self-stabilizing iterative solvers”. In *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pp. 4. ACM.

Snir, M., R. Wisniewski, J. Abraham, S. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson et al. 2014. “Addressing failures in exascale computing”. *International Journal of High Performance Computing Applications*.

Stoyanov, M., and C. Webster. 2015. “Numerical analysis of fixed point algorithms in the presence of hardware faults”. *SIAM Journal on Scientific Computing* vol. 37 (5), pp. C532–C553.

AUTHOR BIOGRAPHIES

EVAN COLEMAN is a lead scientist with the Naval Surface Warfare Center Dahlgren Division. He holds an MS in Mathematics from Syracuse University and is working on a PhD in Modeling and Simulation from Old Dominion University. His email address is ecole028@odu.edu.

MASHA SOSONKINA is a Professor of Modeling, Simulation and Visualization Engineering at Old Dominion University. Her research interests include high-performance computing, large-scale simulations, parallel numerical algorithms, and performance analysis. Her email address is msosonki@odu.edu.

PREDICTIVE MODELING OF I/O CHARACTERISTICS IN HIGH PERFORMANCE COMPUTING SYSTEMS

Thomas C. H. Lux

Dept. of Computer Science
Virginia Polytechnic Institute
& State University (VPI & SU)
Blacksburg, VA 24061
tchlux@vt.edu

Layne T. Watson

Dept. of Computer Science
Dept. of Mathematics
Dept. of Aerospace & Ocean Eng.
VPI & SU

Tyler H. Chang

Dept. of Computer Science
VPI & SU, Blacksburg, VA 24061

Jon Bernard

Dept. of Computer Science
VPI & SU, Blacksburg, VA 24061

Bo Li

Dept. of Computer Science
VPI & SU, Blacksburg, VA 24061

Li Xu

Dept. of Statistics
VPI & SU, Blacksburg, VA 24061

Godmar Back

Dept. of Computer Science
VPI & SU, Blacksburg, VA 24061

Ali R. Butt

Dept. of Computer Science
VPI & SU, Blacksburg, VA 24061

Kirk W. Cameron

Dept. of Computer Science
VPI & SU, Blacksburg, VA 24061

Yili Hong

Dept. of Statistics
VPI & SU, Blacksburg, VA 24061

ABSTRACT

Each of high performance computing, cloud computing, and computer security have their own interests in modeling and predicting the performance of computers with respect to how they are configured. An effective model might infer internal mechanics, minimize power consumption, or maximize computational throughput of a given system. This paper analyzes a four-dimensional dataset measuring the input/output (I/O) characteristics of a cluster of identical computers using the benchmark IOzone. The I/O performance characteristics are modeled with respect to system configuration using multivariate interpolation and approximation techniques. The analysis reveals that accurate models of I/O characteristics for a computer system may be created from a small fraction of possible configurations, and that some modeling techniques will continue to perform well as the number of system parameters being modeled increases. These results have strong implications for future predictive analyses based on more comprehensive sets of system parameters.

Keywords: Regression, approximation, interpolation, performance modeling

1 INTRODUCTION AND RELATED WORK

Performance tuning is often an experimentally complex and time intensive chore necessary for configuring HPC systems. The procedures for this tuning vary largely from system to system and are often subjectively guided by the system engineer(s). Once a desired level of performance is achieved, an HPC system may only be incrementally reconfigured as required by updates or specific jobs. In the case that a system has changing workloads or nonstationary performance objectives that range from maximizing computational throughput to minimizing power consumption and system variability, it becomes clear that a more effective and automated tool is needed for configuring systems. This scenario presents a challenging and important application of multivariate approximation and interpolation techniques.

Predicting the performance of an HPC system is a challenging problem that is primarily attempted in one of two ways: (1) build a statistical model of the performance by running experiments on the system at select settings, or (2) run artificial experiments using a simplified simulation of the target system to estimate architecture and application bottlenecks. In this paper the proposed multivariate modeling techniques rest in the first category, and they represent a notable increase in the ability to model complex interactions between system parameters.

Many previous works attempting to model system performance have used simulated environments to estimate the performance of a system (Grobelny et al. 2007, Wang et al. 2009, Wang et al. 2013). Some of these works refer to statistical models as being oversimplified and not capable of capturing the true complexity of the underlying system. This claim is partially correct, noting that a large portion of predictive statistical models rely on simplifying the model to one or two parameters (Snavely et al. 2002, Bailey and Snavely 2005, Barker et al. 2009, Ye et al. 2010). These limited statistical models have provided satisfactory performance in very narrow application settings. Many of the aforementioned statistical modeling techniques claim to generalize, while simultaneously requiring additional code annotations, hardware abstractions, or additional application level understandings in order to generate models. The approach presented here requires no modifications of the application, no architectural abstractions, nor any structural descriptions of the input data being modeled. The techniques used are purely mathematical and only need performance data as input.

Among the statistical models presented in prior works, Bailey and Snavely (2005) specifically mention that it is difficult for the simplified models to capture variability introduced by I/O. System variability in general has become a problem of increasing interest to the HPC and systems communities, however most of the work has focused on operating system (OS) induced variability (Beckman et al. 2008, De et al. 2007). The work that has focused on managing I/O variability does not use any sophisticated modeling techniques (Lofstead et al. 2010). Hence, this paper presents a case study applying advanced mathematical and statistical modeling techniques to the domain of HPC I/O characteristics. The models are used to predict the mean throughput of a system and the variance in throughput of a system. The discussion section outlines how the techniques presented can be applied to any performance metric and any system.

In general, this paper compares five multivariate approximation techniques that operate on inputs in $\mathbb{R}^d$ (d-tuples of real numbers) and produce predicted responses in $\mathbb{R}$. In order to provide coverage of the varied mathematical strategies that can be employed to solve the continuous modeling problem, three of the techniques are regression based and the remaining two are interpolants. The sections below outline the mathematical formulation of each technique and provide computational complexity bounds with respect to the size (number of points and dimension) of input data. Throughout, d will refer to the dimension of the input data, n is the number of points in the input data, $x^{(i)} \in \mathbb{R}^d$ is the i-th input data point, $x_j^{(i)}$ is the j-th component of $x^{(i)}$, and $f(x^{(i)})$ is the response value of the i-th input data point.

The remainder of the paper is broken up into five major parts. Section 2 provides an overview of the multivariate modeling techniques, Section 3 outlines the methodology for comparing and evaluating the performance of the models, Section 4 presents the IOzone predictions, Section 5 discusses the obvious and subtle implications of the models' performance, and Section 6 concludes and offers directions for future work.

2 MULTIVARIATE MODELS

2.1 Regression

Multivariate approximations are capable of accurately modeling a complex dependence of a response (in $\mathbb{R}$) on multiple variables (represented as a points in $\mathbb{R}^d$). The approximations to some (unknown) underlying function $f : \mathbb{R}^d \to \mathbb{R}$ are chosen to minimize some error measure related to data samples $f(x^{(i)})$. For example, least squares regression uses the sum of squared differences between modeled response values and true response values as an error measure.

2.1.1 Multivariate Adaptive Regression Splines

This approximation was introduced in Friedman (1991) and subsequently improved to its current version in Friedman and the Computational Statistics Laboratory of Stanford University (1993), called fast multivariate adaptive regression splines (Fast MARS). In Fast MARS, a least squares fit model is iteratively built by beginning with a single constant valued function and adding two new basis functions at each iteration of the form

$$B_{2s-1}(x) = B_l(x)[c(x_i - v)]_+,$$
$$B_{2s}(x) = B_k(x)[c(x_i - v)]_-,$$

where s is the iteration number, $B_l(x)$ and $B_k(x)$ are basis functions from the previous iteration, $c, v \in \mathbb{R}$,

$$w_+ = \begin{cases} w, & w \geq 0 \\ 0, & w < 0 \end{cases},$$

and $w_- = (-w)_+$. After iteratively constructing a model, MARS then iteratively removes basis functions that do not contribute to goodness of fit. In effect, MARS creates a locally component-wise tensor product approximation of the data. The overall computational complexity of Fast MARS is $\mathcal{O}(ndm^3)$ where m is the maximum number of underlying basis functions. This paper uses an implementation of Fast MARS (Rudy and Cherti 2017) with $m = 200$.

2.1.2 Multilayer Perceptron Regressor

The neural network is a well studied and widely used method for both regression and classification tasks (Hornik et al. 1989). When using the rectified linear unit (ReLU) activation function (Dahl et al. 2013) and training with the BFGS minimization technique (Møller 1993), the model built by a multilayer perceptron uses layers $l : \mathbb{R}^i \to \mathbb{R}^j$ defined by

$$l(u) = \left(u^t W_l\right)_+,$$

where W_l is the i by j weight matrix for layer l. In this form, the multilayer perceptron (MLP) produces a piecewise linear model of the input data. The computational complexity of training a multilayer perceptron is $\mathcal{O}(ndm)$, where m is determined by the sizes of the layers of the network and the stopping criterion of the BFGS minimization used for finding weights. This paper uses the scikit-learn MLP regressor (Pedregosa et al. 2011), a single hidden layer with 100 nodes, ReLU activation, and BFGS for training.

2.1.3 Support Vector Regressor

Support vector machines are a common method used in machine learning classification tasks that can be adapted for the purpose of regression (Basak et al. 2007). How to build a support vector regressor (SVR) is beyond the scope of this summary, but the resulting functional fit $p : \mathbb{R}^d \to \mathbb{R}$ has the form

$$p(x) = \sum_{i=1}^{n} a_i K(x, x^{(i)}) + b,$$

where K is the selected kernel function, $a \in \mathbb{R}^n$, $b \in \mathbb{R}$ are coefficients to be solved for simultaneously. The computational complexity of the SVR is $\mathcal{O}(n^2 dm)$, with m being determined by the minimization convergence criterion. This paper uses the scikit-learn SVR (Pedregosa et al. 2011) with a polynomial kernel function.

2.2 Interpolation

In some cases it is desirable to have a model that can recreate the input data exactly. This is especially the case when the confidence in the response values for known data is high. Both interpolation models analyzed in this paper are based on linear functions.

2.2.1 Delaunay

The Delaunay method of interpolation is a well studied geometric technique for producing an interpolant (Lee and Schachter 1980). The Delaunay triangulation of a set of data points into simplices is such that the sphere defined by the vertices of each simplex contains no data points in the sphere's interior. For a d-simplex S with vertices $v^{(0)}$, $v^{(1)}$, …, $v^{(d)}$, $x \in S$, and data values $f(v^{(i)})$, $i = 0, \ldots, d$, x is a unique convex combination of the vertices,

$$x = \sum_{i=0}^{d} w_i v^{(i)}, \quad \sum_{i=0}^{d} w_i = 1, \quad w_i \geq 0, \quad i = 0, \ldots, d,$$

and the Delaunay interpolant to f at x is given by

$$p(x) = \sum_{i=0}^{d} w_i f(v^{(i)}).$$

The computational complexity of the Delaunay triangulation (for the implementation used here) is $\mathcal{O}(n^{\lceil d/2 \rceil})$, which is not scalable to $d > 10$ (Sartipizadeh and Vincent 2016). The scipy interface (Jones

System Parameter	**Values**
File Size	64, 256, 1024
Record Size	32, 128, 512
Thread Count	1, 2, 4, 8, 16, 32, 64, 128, 256
Frequency	$\{12, 14, 15, 16, 18, 19, 20, 21, 23, 24, 25, 27, 28, 29, 30, 30.01\} \times 10^5$
Response Values	
Throughput Mean	$[2.6 \times 10^5, 5.9 \times 10^8]$
Throughput Variance	$[5.9 \times 10^{10}, 4.7 \times 10^{16}]$

Table 1: A description of the system parameters being considered in the IOzone tests. Record size must not be greater than file size and hence there are only six valid combinations of the two. In total there are $6 \times 9 \times 16 = 864$ unique system configurations.

et al. 2017) to the QuickHull implementation (Barber et al. 1996) of the Delaunay triangulation is used here.

2.2.2 Linear Shepard

The linear Shepard method (LSHEP) is a blending function using local linear interpolants, a special case of the general Shepard algorithm (Thacker et al. 2010). The interpolant has the form

$$p(x) = \frac{\sum_{k=1}^{n} W_k(x) P_k(x)}{\sum_{k=1}^{n} W_k(x)},$$

where $W_k(x)$ is a locally supported weighting function and $P_k(x)$ is a local linear approximation to the data satisfying $P_k\big(x^{(k)}\big) = f\big(x^{(x)}\big)$. The computational complexity of LSHEP is $\mathscr{O}(n^2 d^3)$. This paper uses the Fortran#95 implementation of LSHEP in SHEPPACK (Thacker et al. 2010).

3 METHODOLOGY

3.1 Data

In order to evaluate the viability of multivariate models for predicting system performance, this paper presents a case study of a four-dimensional dataset produced by executing the IOzone benchmark from Norcott (2017) on a homogeneous cluster of computers. All experiments were performed on parallel shared-memory nodes common to HPC systems. Each system had a lone guest Linux operating system (Ubuntu 14.04 LTS//XEN 4.0) on a dedicated 2TB HDD on a 2 socket, 4 core (2 hyperthreads per core) Intel Xeon E5-2623 v3 (Haswell) platform with 32 GB DDR4. The system performance data was collected by executing IOzone 40 times for each of a select set of system configurations. A single IOzone execution reports the max I/O throughput seen for the selected test in kilobytes per second. The 40 executions for each system configuration are converted into the mean and variance, both values in $\mathbb{R}$ capable of being modeled individually by the multivariate approximation techniques presented in Section 2. The summary of data used in the experiments for this paper can be seen in Table 1. Distributions of raw throughput values being modeled can be seen in Figure 1.

3.2 Dimensional Analysis

This work utilizes an extension to standard k-fold cross validation that allows for a more thorough investigation of the expected model performance in a variety of real-world situations. Alongside randomized splits, two extra components are considered: the amount of training data provided, and the dimension of

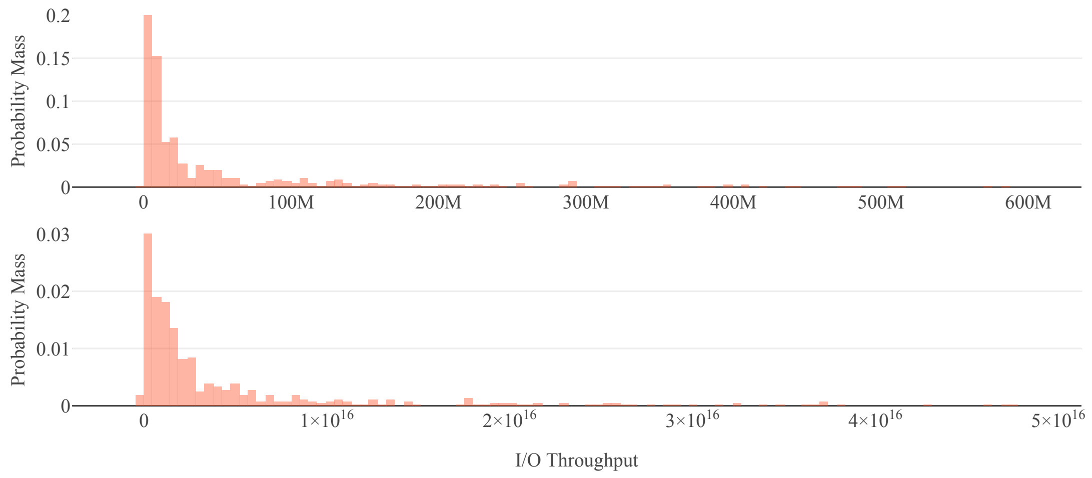


Figure 1: Histograms of 100-bin reductions of the PMF of I/O throughput mean (top) and I/O throughput variance (bottom). In the mean plot, the first 1% bin (truncated in plot) has a probability mass of .45. In the variance plot, the second 1% bin has a probability mass of .58. It can be seen that the distributions of throughputs are primarily of lower magnitude with occasional extreme outliers.

the input data. It is important to consider that algorithms that perform well with less training input also require less experimentation. Although, the amount of training data required may change as a function of the dimension of the input and this needs to be studied as well. The framework used here will be referred to as a multidimensional analysis (MDA) of the IOzone data.

3.2.1 Multidimensional Analysis

This procedure combines random selection of training and testing splits with changes in the input dimension and the ratio of training size to testing size. Given an input data matrix with n rows (points) and d columns (components), MDA proceeds as follows:

1. For all $k = 1, \ldots, d$ and for all nonempty subsets $F \subset \{1, 2, \ldots, d\}$, reduce the input data to points $(z, f_F(z))$ with $z \in \mathbb{R}^k$ and $f_F(z) = E\left[\left\{f\left(x^{(i)}\right) \mid \left(x_F^{(i)} = z\right)\right\}\right]$, where $E[\cdot]$ denotes the mean and $x_F^{(i)}$ is the subvector of $x^{(i)}$ indexed by F.
2. For all r in $\{5, 10, \ldots, 95\}$, generate N random splits $(train, test)$ of the reduced data with r percentage for training and $100 - r$ percentage for testing.
3. When generating each of N random $(train, test)$ splits, ensure that all points from $test$ are in the convex hull of points in $train$ (to prevent extrapolation); also ensure that the points in $train$ are well spaced.

In order to ensure that the testing points are in the convex hull of the training points, the convex hull vertices of each set of (reduced dimension) points are forcibly placed in the training set. In order to ensure that training points are well spaced, a statistical method for picking points from Amos et al. (2014) is used:

1. Generate a sequence of all pairs of points $\left(z^{(i_1)}, z^{(j_1)}\right), \left(z^{(i_2)}, z^{(j_2)}\right), \ldots$ sorted by ascending pairwise Euclidean distance between points, so that $\left|\left|z^{(i_k)} - z^{(j_k)}\right|\right|_2 \leq \left|\left|z^{(i_{k+1})} - z^{(j_{k+1})}\right|\right|_2$.

2. Sequentially remove points from candidacy until only $|train|$ remain by randomly selecting one point from the pair $\left(z^{(i_m)}, z^{(j_m)}\right)$ for $m = 1, \ldots$ if both $z^{(i_m)}$ and $z^{(j_m)}$ are still candidates for removal.

Given the large number of constraints, level of reduction, and use of randomness in the MDA procedure, occasionally N unique training/testing splits may not be created or may not exist. In these cases, if there are fewer than N possible splits, then deterministically generated splits are used. Otherwise after $3N$ attempts, only the unique splits are kept for analysis. The MDA procedure has been implemented in Python#3 while most regression and interpolation methods are Fortran wrapped with Python. All randomness has been seeded for repeatability.

For any index subset F (of size k) and selected value of r, MDA will generate up to N multivariate models $f_F(z)$ and predictions $\hat{f}_F\left(z^{(i)}\right)$ for a point $z^{(i)} \in \mathbb{R}^k$. There may be fewer than N predictions made for any given point. Extreme points of the convex hull for the selected index subset will always be used for training, never for testing. Points that do not have any close neighbors will often be used for training in order to ensure well-spacing. Finally, as mentioned before, some index subsets do not readily generate N unique training and testing splits. The summary results presented in this work use the median of the (N or fewer) values $\hat{f}_F(z)$ at each point as the model estimate for error analysis.

4 RESULTS

A naïve multivariate prediction technique such as nearest neighbor could experience relative errors in the range $[0, \left(\max_x f(x) - \min_x f(x)\right) / \min_x f(x)]$ when modeling a nonnegative function $f(x)$ from data. The IOzone mean data response values span three orders of magnitude (as can be seen in Table 1) while variance data response values span six orders of magnitude. It is expected therefore, that all studied multivariate models perform better than a naïve approach, achieving relative errors strictly less than 10^3 for throughput mean and 10^6 for throughput variance. Ideally, models will yield relative errors significantly smaller than 1. The time required to compute thousands of models involved in processing the IOzone data through MDA was approximately five hours on a CentOS workstation with an Intel i7-3770 CPU at 3.40GHz. In four dimensions for example, each of the models could be constructed and evaluated over hundreds of points in less than a few seconds.

4.1 I/O Throughput Mean

Almost all multivariate models analyzed make predictions with a relative error less than 1 for most system configurations when predicting the mean I/O throughput of a system given varying amounts of training data. The overall best of the multivariate models, Delaunay, consistently makes predictions with relative error less than .05 (5% error). In Figure 3 it can also be seen that the Delaunay model consistently makes good predictions even with as low as 5% training data (43 of the 864 system configurations) regardless of the dimension of the data.

4.2 I/O Throughput Variance

The prediction results for variance resemble those for predicting mean. Delaunay remains the best overall predictor (aggregated across training percentages and dimensions) with median relative error of .47 and LSHEP closely competes with Delaunay having a median signed relative error of -.92. Outliers in prediction error are much larger for all models. Delaunay produces relative errors as large as 78 and other models achieve relative errors around 10^3. The relative errors for many models scaled proportional to the increased orders of magnitude spanned by the variance response compared with mean response. As can be seen in Figure 4, all models are more sensitive to the amount of training data provided than their counterparts for predicting mean.

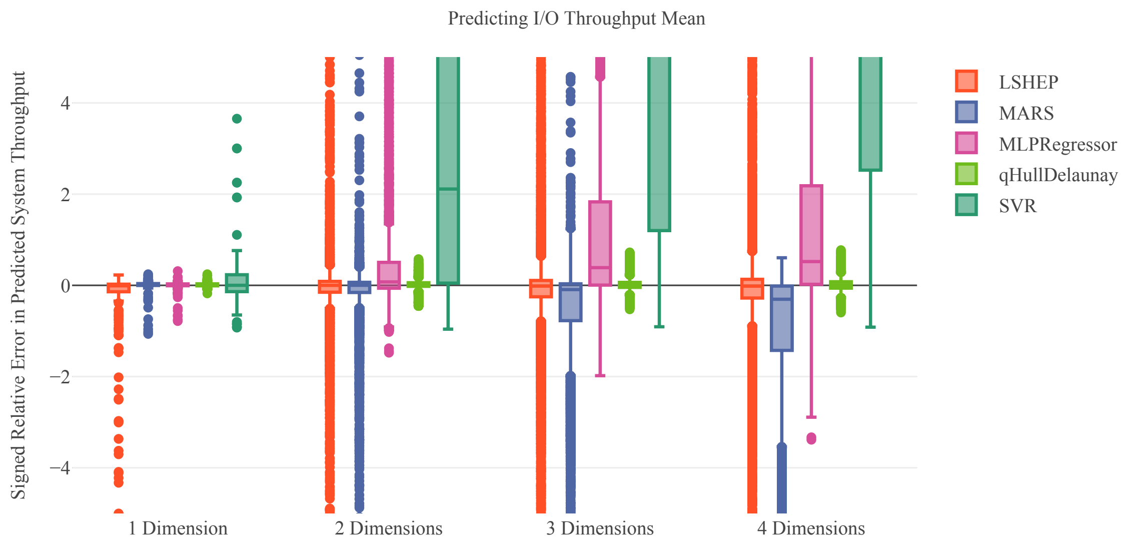


Figure 2: These box plots show the prediction error of mean with increasing dimension. The top box whisker for SVR is 40, 80, 90 for dimensions 2, 3, and 4, respectively. Notice that each model consistently experiences greater magnitude error with increasing dimension. Results for all training percentages are aggregated.

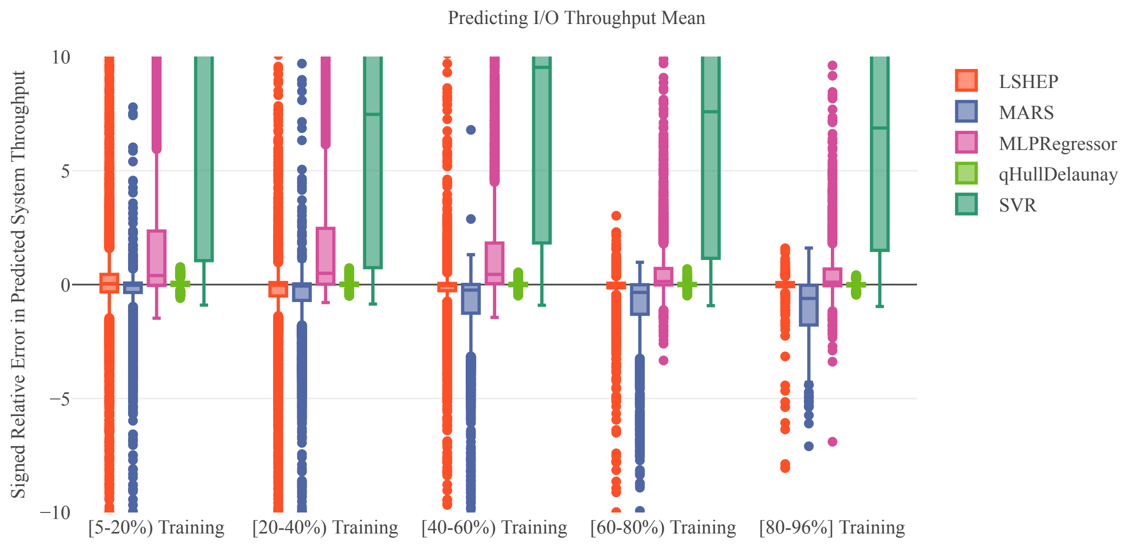


Figure 3: These box plots show the prediction error of mean with increasing amounts of training data provided to the models. Notice that MARS is the only model whose primary spread of performance increases with more training data. Recall that the response values being predicted span three orders of magnitude and hence relative errors should certainly remain within that range. For SVR the top box whisker goes from around 100 to 50 from left to right and is truncated in order to maintain focus on better models. Results for all dimensions are aggregated. Max training percentage is 96% due to rounding training set size.

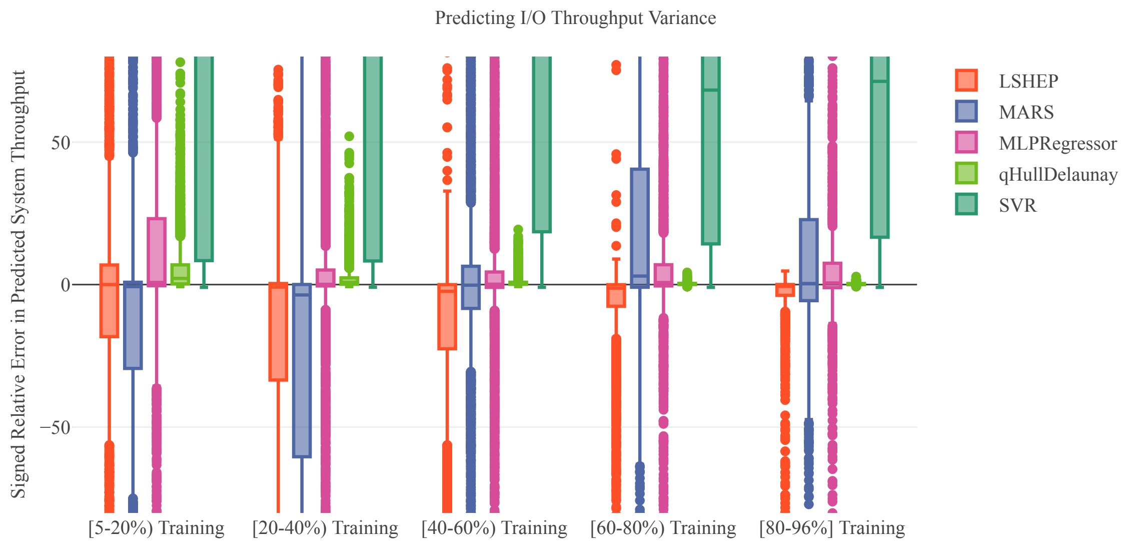


Figure 4: These box plots show the prediction error of variance with increasing amounts of training data provided to the models. The response values being predicted span six orders of magnitude. For SVR the top box whisker goes from around 6000 to 400 (decreasing by factors of 2) from left to right and is truncated in order to maintain focus on better models. Results for all dimensions are aggregated. Max training percentage is 96% due to rounding training set size.

4.3 Increasing Dimension and Decreasing Training Data

As can be seen in Figure 2, all of the models suffer increasing error rates in higher dimension. This is expected, because the number of possible interactions to model grows exponentially. However, LSHEP and Delaunay maintain the slowest increase in relative error. The increase in error seen for Delaunay suggests that it is capable of making predictions with a range of relative errors that grows approximately linearly with increasing dimension input. This trend suggests that Delaunay would remain a viable technique for accurately modeling systems with 10's of parameters given only small amounts of training data. All models, with the exception of MARS, produce smaller errors given more training data. Increasing the amount of training data most notably reduces the number of prediction error outliers.

5 DISCUSSION

The present results demonstrate that a straightforward application of multivariate modeling techniques can be used to effectively predict HPC system performance. Some modeling effort on the part of a systems engineer combined with a significant amount of experimentation (days of CPU time for the IOzone data used here) can yield a model capable of accurately tuning an HPC system to the desired performance specification, although qualitatively correct predictions can be achieved with much less (10%, say) effort.

5.1 Modeling the System

The modeling techniques generated estimates of drastically different quality when predicting I/O throughput mean and variance. A few observations: SVR has the largest range of errors for all selections of dimension and amounts of training data; MARS and LSHEP produce similar magnitude errors while the former consistently underestimates and the latter consistently overestimates; Delaunay has considerably fewer outliers than all other methods. SVR likely produces the poorest quality predictions because the underlying parametric representation is global and oversimplified (a single polynomial), making it unable to capture the complex local behaviors of system I/O. It is still unclear, however, what causes the behaviors of LSHEP, MARS, and Delaunay. An exploration of this topic is left to future work.

While the Delaunay method appears to be the best predictor in the present IOzone case study, it is important to note that the Delaunay computational complexity increases with the dimension of the input more rapidly than other techniques. The implementation of Delaunay (QuickHull) used would experience unacceptably large training times beyond ten-dimensional input. This leaves much room for other techniques to perform best in higher dimension unless a more efficient implementation of Delaunay can be used.

Finally, the ability of the models to predict variance was significantly worse than for the I/O mean. The larger scale in variance responses alone do not account for the increase in relative errors witnessed. This suggests that system variability has a greater underlying functional complexity than the system mean and that latent factors are reducing prediction performance.

5.2 Extending the Analysis

System I/O throughput mean and variance are simple and useful system characteristics to model. The process presented in this work is equally applicable to predicting other useful performance characteristics of HPC systems such as: computational throughput, power consumption, processor idle time, context switches, RAM usage, or any other ordinal performance metric. For each of these there is the potential to model system variability as well. This work has chosen variance as a measure of variability, but the techniques used in this paper could be applied to more precise measures of variability such as the percentiles of the performance distribution or the entire distribution itself. A thorough exploration of HPC systems applications of multivariate modeling constitutes future work.

6 CONCLUSION

Multivariate models of HPC system performance can effectively predict I/O throughput mean and variance. These multivariate techniques significantly expand the scope and portability of statistical models for predicting computer system performance over previous work. In the IOzone case study presented, the Delaunay method produces the best overall results making predictions for 821 system configurations with less than 5% error when trained on only 43 configurations. Analysis also suggests that the error in the Delaunay method will remain acceptable as the number of system parameters being modeled increases. These multivariate techniques can and should be applied to HPC systems with more than four tunable parameters in order to identify optimal system configurations that may not be discoverable via previous methods nor by manual performance tuning.

6.1 Future Work

The most severe limitation to the present work is the restriction to modeling strictly ordinal (not categorical) system parameters. Existing statistical approaches for including categorical variables are inadequate for nonlinear interactions in high dimensions. Future work could attempt to identify the viability of different techniques for making predictions including categorical system parameters.

There remain many other multivariate modeling techniques not included in this work that should be evaluated and applied to HPC performance prediction. For I/O alone, there are far more than the four tunable parameters studied in this work. Alongside experimentation with more models, there is room for a theoretical characterization of the combined model and data properties that allow for the greatest predictive power.

REFERENCES

Amos, B. D., D. R. Easterling, L. T. Watson, W. I. Thacker, B. S. Castle, and M. W. Trosset. 2014. “Algorithm XXX: QNSTOP—quasi-Newton algorithm for stochastic optimization”. *Technical Report 14-02, Dept. of Computer Science, VPI&SU, Blacksburg, VA*.

Bailey, D. H., and A. Snavely. 2005. “Performance modeling: Understanding the past and predicting the future”. In *European Conference on Parallel Processing*, pp. 185–195. Springer.

Barber, C. B., D. P. Dobkin, and H. Huhdanpaa. 1996, December. “The Quickhull Algorithm for Convex Hulls”. *ACM Trans. Math. Softw.* vol. 22 (4), pp. 469–483.

Barker, K. J., K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. 2009. “Using performance modeling to design large-scale systems”. *Computer* vol. 42 (11).

Basak, D., S. Pal, and D. C. Patranabis. 2007. “Support vector regression”. *Neural Information Processing-Letters and Reviews* vol. 11 (10), pp. 203–224.

Beckman, P., K. Iskra, K. Yoshii, S. Coghlan, and A. Nataraj. 2008. “Benchmarking the effects of operating system interference on extreme-scale parallel machines”. *Cluster Computing* vol. 11 (1), pp. 3–16.

Dahl, G. E., T. N. Sainath, and G. E. Hinton. 2013. “Improving deep neural networks for LVCSR using rectified linear units and dropout”. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2013*, pp. 8609–8613. IEEE.

De, P., R. Kothari, and V. Mann. 2007. “Identifying sources of operating system jitter through fine-grained kernel instrumentation”. In *IEEE International Conference on Cluster Computing*, pp. 331–340. IEEE.

Friedman, J. H. 1991. “Multivariate adaptive regression splines”. *The Annals of Statistics*, pp. 1–67.

Friedman, J. H., and the Computational Statistics Laboratory of Stanford University. 1993. *Fast MARS*.

Grobelny, E., D. Bueno, I. Troxel, A. D. George, and J. S. Vetter. 2007. “FASE: A framework for scalable performance prediction of HPC systems and applications”. *Simulation* vol. 83 (10), pp. 721–745.

Hornik, K., M. Stinchcombe, and H. White. 1989. “Multilayer feedforward networks are universal approximators”. *Neural networks* vol. 2 (5), pp. 359–366.

Jones, E. and Oliphant, T. and Peterson, P. 2017. “SciPy: Open source scientific tools for Python”. http://www.scipy.org/ [Online; accessed 2017-06-23].

Lee, D.-T., and B. J. Schachter. 1980. “Two algorithms for constructing a Delaunay triangulation”. *International Journal of Computer & Information Sciences* vol. 9 (3), pp. 219–242.

Lofstead, J., F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf. 2010. “Managing variability in the IO performance of petascale storage systems”. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC), 2010*, pp. 1–12. IEEE.

Møller, M. F. 1993. “A scaled conjugate gradient algorithm for fast supervised learning”. *Neural networks* vol. 6 (4), pp. 525–533.

Norcott, W. D. 2017. “IOzone Filesystem Benchmark”. http://www.iozone.org [Online; accessed 2017-11-12].

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. “Scikit-learn: Machine Learning in Python”. *Journal of Machine Learning Research* vol. 12, pp. 2825–2830.

Rudy, J. and Cherti, M. 2017. “Py-Earth: A Python Implementation of Multivariate Adaptive Regression Splines”. https://github.com/scikit-learn-contrib/py-earth [Online; accessed 2017-07-09].

Sartipizadeh, H., and T. L. Vincent. 2016. “Computing the Approximate Convex Hull in High Dimensions”. *CoRR* vol. abs/1603.04422.

Snavely, A., L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. 2002. “A framework for performance modeling and prediction”. In *Supercomputing, ACM/IEEE Conference*, pp. 21–21. IEEE.

Thacker, W. I., J. Zhang, L. T. Watson, J. B. Birch, M. A. Iyer, and M. W. Berry. 2010. “Algorithm 905: SHEPPACK: Modified Shepard algorithm for interpolation of scattered multivariate data”. *ACM Transactions on Mathematical Software (TOMS)* vol. 37 (3), pp. 34.

Wang, G., A. R. Butt, P. Pandey, and K. Gupta. 2009. “A simulation approach to evaluating design decisions in mapreduce setups”. In *IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS’09)*, pp. 1–11. IEEE.

Wang, G., A. Khasymski, K. R. Krish, and A. R. Butt. 2013. “Towards improving mapreduce task scheduling using online simulation based predictions”. In *International Conference on Parallel and Distributed Systems (ICPADS), 2013*, pp. 299–306. IEEE.

Ye, K., X. Jiang, S. Chen, D. Huang, and B. Wang. 2010. “Analyzing and modeling the performance in xen-based virtual cluster environment”. In *12th IEEE International Conference on High Performance Computing and Communications (HPCC)*, pp. 273–280. IEEE.

AUTHOR BIOGRAPHIES

THOMAS C. H. LUX is a Ph.D. student at Virginia Tech studying computer science under Dr. Layne Watson.

LAYNE T. WATSON (Ph.D., Michigan, 1974) has interests in numerical analysis, mathematical programming, bioinformatics, and data science. He has been involved with the organization of HPCS since 2000.

TYLER H. CHANG is a Ph.D. student at Virginia Tech studying computer science under Dr. Layne Watson.

JON BERNARD is a Ph.D. student at Virginia Tech studying computer science under Dr. Kirk Cameron.

BO LI is a senior Ph.D. student at Virginia Tech studying computer science under Dr. Kirk Cameron.

LI XU is a Ph.D. student at Virginia Tech studying statistics under Dr. Yili Hong.

GODMAR BACK (Ph.D., University of Utah, 2002) has broad interests in computer systems, with a focus on performance and reliability aspects of operating systems and virtual machines.

ALI R. BUTT (Ph.D., Purdue, 2006) has interests in cloud computing, distributed computing, and operating system induced variability.

KIRK W. CAMERON (Ph.D., Louisiana State, 2000) has interests in computer systems design, performance analysis, and power and energy efficiency.

YILI HONG (Ph.D., Iowa State, 2009) has interests in engineering statistics, statistical modeling, and data analysis.

PARALLEL IN TIME ALGORITHMS FOR MULTISCALE DYNAMICAL SYSTEMS USING INTERPOLATION AND NEURAL NETWORKS

Gopal R. Yalla

Institute for Computational Engineering and Sciences
University of Texas at Austin
201 East 24th Street
Austin, TX, USA
gopal@ices.utexas.edu

Bjorn Engquist

Institute for Computational Engineering and Sciences
University of Texas at Austin
201 East 24th Street
Austin, TX, USA
engquist@ices.utexas.edu

ABSTRACT

The parareal algorithm allows for efficient parallel in time computation of dynamical systems. We present a novel coarse scale solver to be used in the parareal framework. The coarse scale solver can be defined through interpolation or as the output of a neural network, and accounts for slow scale motion in the system. Through a parareal scheme, we pair this coarse solver with a fine scale solver that corrects for fast scale motion. By doing so we are able to achieve the accuracy of the fine solver at the efficiency of the coarse solver. Successful tests for smaller but challenging problems are presented, which cover both highly oscillatory solutions and problems with strong forces localized in time. The results suggest significant speed up can be gained for multiscale problems when using a parareal scheme with this new coarse solver as opposed to the traditional parareal setup.

Keywords: parareal, multiscale, dynamical systems, neural networks, interpolation

1 INTRODUCTION

Emerging multicore and many-core architectures will lead to a continued increase in computing power in the coming years. These complex architectures necessitate highly parallel and distributed computing for scientific simulations. While the capabilities of these machines benefit most scientific and engineering disciplines, they can also present challenges such as those for systems governed by time dependent dynamical systems. The effect of causality in time is more naturally handled sequentially. In (Lions, Maday, et al. 2001), the parareal algorithm for parallel in time computations is introduced, which is essential for highly parallel systems when distributed computation in space saturates. See also (Gander 2015) for a presentation of early parallel in time techniques including parareal algorithms. Initially the technique was applied to dissipative systems where the memory effect is more limited, but there are very important applications without dissipations as, for example, hyperbolic partial differential equations (PDE) and systems of ordinary differential equations (ODE) for molecular dynamics and celestial mechanics.

The parareal method is based on two solvers, one coarse scale solver, which is less accurate but fast enough to be applied sequentially, and one fine scale solver with full accuracy that is applied in parallel in time. At each iteration, the coarse solver is run sequentially over each subdomain similar to a shooting method, and then the fine solver is run in parallel as a correction. See section 2 for how the coarse and fine solvers interact. This coupling of the solvers is able to achieve the accuracy of the fine scale solver at the efficiency of the coarse scale solver. The overall technique can be seen as a domain decomposition method for the fine scale method; the coarse scale method and the iterative correction provide the coupling between the domains.

The coarse solver is a usually based on one or a few steps of a regular numerical solver. For instance, the coarse scale solver is typically taken to be an ODE solver with significantly larger time step compared to the fine solver or an approximation where the ODE itself has been modified to be less stiff or oscillatory. In opposition, the fine scale solver typically involves a very large number of local time steps for high accuracy. Gander and Hairer (2014) show that for Hamiltonian systems, very high accuracy is required already for the coarse solver for the process to work. This is not helpful in, for example, molecular dynamics where the fine step size is chosen to be at the limit of what gives a meaningful result. Since the computational efficiency of the parareal algorithm comes from the fact that the costly fine scale solver can be implemented in parallel in time, forcing the sequential coarse solver to be nearly as accurate as the fine solver succumbs the algorithm to Amdahl's law and ruins any speed up gained by parallelization.

The main innovation in this work is the new coarse solver, which relies on a pre-computed phase plane map, similar to the work in (Nievergelt 1964) and (Ying and Candès 2006), but applied in a parareal setting and optimized for highly oscillatory, autonomous dynamical systems. See (Barker 2014) for additional work in extending Nievergelt's method. The phase-plane map can be built with interpolation or the output of a neural network. A given dynamical system is accurately approximated with the fine solver for a single time interval and for a number of different initial values. The pre-computing for different initial values is embarrassingly parallel and the phase plane map does not suffer from the difficulties of highly oscillatory behavior, which is a main problem with standard parareal computations. Even a linear harmonic oscillator is a severe challenge for standard parareal simulations (Gander and Hairer 2014). For the interpolation based phase plane map linear ODEs require only linear interpolation for getting the accuracy of the fine scale solver at the efficiency of the coarse solver.

2 DESCRIPTION OF THE ALGORITHM

2.1 The Parareal Algorithm

Consider the dynamical system,

$$\begin{aligned} &u'(t) = f(u(t)), \qquad t \in (t_0, t_f) \\ &u(t_0) = u_0 \end{aligned}, \tag{1}$$

where $f : \mathbb{R}^d \to \mathbb{R}^d$ and $u : \mathbb{R} \to \mathbb{R}^d$. To obtain the parareal algorithm as in (Gander, Hairer, et al. 2008), first divide the time domain $\Omega \equiv (t_0, t_f)$ into N equal subdomains $\Omega_n = (T_n, T_{n+1})$, $n = 0, 1, \ldots, N-1$, with $t_0 = T_0 < T_1 < \cdots < T_{N-1} < T_N = t_f$, and $\Delta T \equiv T_{n+1} - T_n$. Consider now the problem on each time subdomain:

$$\begin{aligned} u_n'(t) &= f(u_n(t)), \qquad t \in (T_n, T_{n+1}) \\ u(T_n) &= U_n \end{aligned}, \tag{2}$$

where the initial conditions U_n are the solution of (1) at time T_n. If we let $\phi_{\Delta T_n}(U_n)$ denote the solution of (1) with initial condition U_n after time ΔT_n, equation (2) is simply a shooting method in time and is equivalent to solving:

$$\begin{pmatrix} U_0 - u_0 \\ U_1 - \phi_{\Delta T_0}(U_0) \\ \vdots \\ U_{N-1} - \phi_{\Delta T_{N-2}}(U_{N-2})) \end{pmatrix} = 0 \tag{3}$$

Applying Newton's method to (3) gives,

$$\begin{aligned} U_0^{k+1} &= u_0 \\ U_n^{k+1} &= \phi_{\Delta T_{n-1}}(U_{n-1}^k) + \phi'_{\Delta T_{n-1}}(U_{n-1}^k)(U_{n-1}^{k+1} - U_{n-1}^k) \end{aligned}. \tag{4}$$

Now, introduce a fine solver $\mathcal{F}$ and coarse solver $\mathcal{G}$ as follows. Let $\mathcal{F}(U_n)$ be an accurate approximation to the solution $\phi_{\Delta T_n}(U_n)$ and $\mathcal{G}(U_n)$ be a less accurate approximation to $\phi_{\Delta T_n}(U_n)$. For example, $\mathcal{G}$ may be defined on a coarser grid than $\mathcal{F}$ or be a lower order method than $\mathcal{F}$. If we use the fine solver to approximate $\phi_{\Delta T_{n-1}}(U_{n-1}^k)$ in (4) and the coarse solver to approximate the Jacobian term in (4), the parareal algorithm to solve (1) is given by, (Gander and Hairer 2014),:

$$\begin{aligned} U_0^{k+1} &= u_0 \\ U_n^{k+1} &= \mathcal{G}(U_{n-1}^{k+1}) + \left[\mathcal{F}(U_{n-1}^k) - \mathcal{G}(U_{n-1}^k)\right] \end{aligned} \tag{5}$$

so that $U_n^k \approx u(T_n)$, where k represents the k^{th} iteration of the algorithm. See (Gander, Hairer, et al. 2008) for a general convergence theory. The term in brackets in (5) can also be seen as a correction term and highlights how the fine solver can be used in parallel at each iteration to correct the approximation made by the coarse solver.

2.2 Phase Plane Map

Suppose now that the time domain has been divided into N equal subdomains so that $\Delta T = (T_N - T_0)/N$. This will be the step size over which our new coarse solver operates. Consider a set of M initial conditions of (1) at time $t = t_0$, denoted by $\{u_0^i\}_{i=1}^M$, and a set of M corresponding target points $\{v^i\}_{i=1}^M$ defined by,

$$v^i = \phi_{\Delta T_0}(u_0^i) \qquad i = 1, \ldots, M\,. \tag{6}$$

As in the derivation of the parareal algorithm, we let $\mathcal{F}(u_0^i)$ approximate $\phi_{\Delta T_0}(u_0^i)$ so that, computationally,

$$v^i = \mathcal{F}(u_0^i) \qquad i = 1,\ldots,M \,. \tag{7}$$

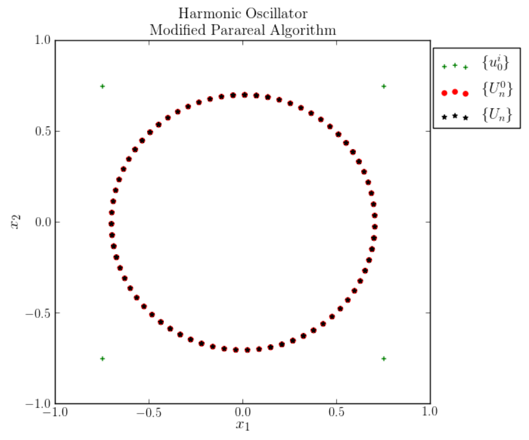
Together, the set $\left\{u_0^i \to v^i\right\}_{i=1}^M$ acts as a type of look-up table for future initial conditions; given a new initial condition and the mapping $\left\{u_0^i \to v^i\right\}_{i=1}^M$, it is possible to estimate a solution using, e.g., interpolation. Formally, we define a phase plane map $\mathcal{G}^{map}(\left\{u_0^i \to v^i\right\}_{i=1}^M, U_n)$ to be a coarse approximation of $\phi_{\Delta T}(U_n)$ that uses the information contained in $\left\{u_0^i \to v^i\right\}_{i=1}^M$ to determine the solution U_{n+1} at time T_{n+1}, given a point U_n at time T_n. $\mathcal{G}^{map}$ may be defined through interpolation on the set $\left\{u_0^i \to v^i\right\}_{i=1}^M$, or through a neural network where the network is trained using $\left\{u_0^i\right\}_{i=1}^M$ as the inputs and $\left\{v^i\right\}_{i=1}^M$ as the outputs. For autonomous dynamical systems like those commonly found in molecular dynamics or celestial mechanics, $\mathcal{G}^{map}$ can be applied over each subdomain Ω_n. For non-autonomous dynamical systems, one can easily extend this idea by generating a set $\left\{u_j^i \to v^i\right\}_{i=1}^M$ for each subdomain Ω_j, $j = 0,\ldots,N-1$, after some initial coarse approximation of the solution is obtained, as in (Nievergelt 1964). Moreover, the computation of $\left\{u_0^i \to v^i\right\}_{i=1}^M$ is embarrassingly parallel and can be done as a preprocessing step to the parareal algorithm, i.e., each v^i is computed independently using the fine solver over Ω_0.

There are several trade offs and design considerations that need to be addressed when building $\mathcal{G}^{map}$. The numerical examples presented in section 3 highlight initial findings for most of these considerations; however, many of them require further investigation. The first problem that needs to addressed is how to select the number of necessary initial conditions M used to build the set $\left\{u_0^i \to v^i\right\}_{i=1}^M$. Clearly, the higher M the more accurate $\mathcal{G}^{map}$ can be; however, it may be computationally expensive to generate a suitable mapping (even though the computation is highly parallel). Nievergelt (1964) suggests that a reasonable choice for M would be one such that the interpolation error is of the same order of magnitude as the truncation error of the fine scale method, $\mathcal{F}$. In a parareal setup, one might expect that M should be chosen so that the interpolation error is comparable to the truncation error of a suitable coarse solver in order for the scheme to provide meaningful results. Furthermore, one must discuss the bounds within which to place these initial conditions $\left\{u_0^i\right\}_{i=1}^M$. In practice, we have used two methods for determining appropriate constraints on the initial conditions. First, domain specific knowledge can often times provide meaningful bounds on the physical phenomenon be simulated. For example, when simulating two atoms bound by a molecular potential, the bounds can be determined from the actual initial condition of interest, U_0, which specifies the positions of the atoms and the total energy in the system. Secondly, for other problems where such information may be difficult to obtain, a coarse approximation to the solution's trajectory can be obtained. For example, if one is interested in simulating one hundred revolutions of a harmonic oscillator, a coarse solver may be used to estimate just one revolution of the oscillator, and appropriate bounds on $\left\{u_0^i\right\}_{i=1}^M$ can be determined from there. Lastly, the efficiency of the algorithm relies on choosing a suitable method for $\mathcal{G}^{map}$. For instance, interpolation may work well for problems in low dimensions, whereas a neural network approach may be better for problems in high dimensions where interpolation is more costly.

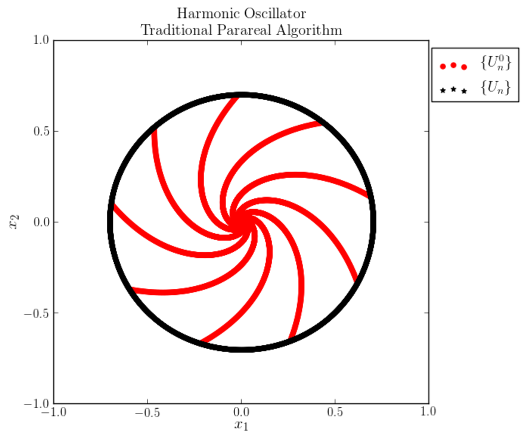
3 NUMERICAL EXAMPLES

3.1 Nomenclature

Throughout the numerical examples we will let T be the total simulation time, N be the number of time subdomains, and M be number of grid points in each dimension so that $\mathcal{G}^{map}$ is defined with $\{u_0^i \to v^i\}_{i=1}^{M^d}$. The fine solver used in each computation is SciPy's adaptive RK45 scheme (Jones, Oliphant, et al. 2001). We refer to the *traditional* parareal algorithm as a parareal scheme where RK4 is used as the coarse solver



(a)



(b)

Figure 1: (a) Results of the modified parareal scheme applied to (8). (b) Results of the traditional parareal scheme applied to (8). Recall that $\left\{u_0^i\right\}$ are the points used to define $\mathcal{G}^{map}$, $\left\{U_n^0\right\}$ is the initial approximation made by the coarse solver, and $\{U_n\}$ is the final converged solution.

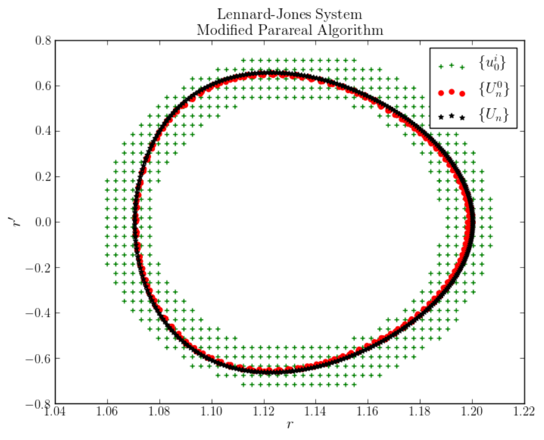
with time step T/N, and the *modified* parareal algorithm as that when the coarse solver is defined through $\mathcal{G}^{map}$.

3.2 Harmonic Oscillator

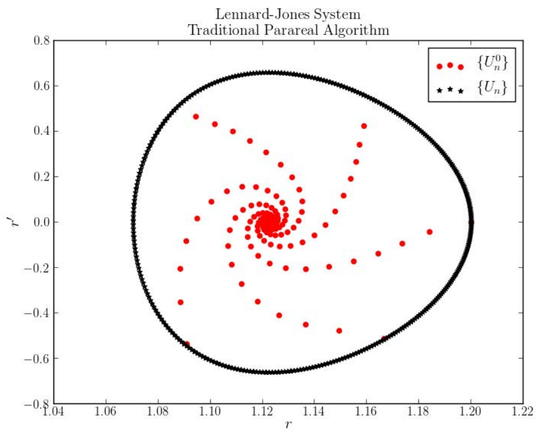
Following similar steps to those in (Nievergelt 1964), it is possible to prove the modified parareal algorithm with $\mathcal{G}^{map}$ defined through linear interpolation will converge in just one iteration for any linear dynamical system. Problems such as a harmonic oscillator that typically present a challenge for parareal schemes, can now be solved in only one iteration. To demonstrate this speedup, consider the following system:

$$\begin{aligned} x_1 &= \frac{1}{\varepsilon}x_2 \\ x_2 &= \frac{-1}{\varepsilon}x_1 \end{aligned} \tag{8}$$

with $\varepsilon = 0.01$ and the initial value $\mathbf{x}(0) = [-2/9, -2/3]^T$. We are interested in the long time solution of this problem, so we set $T = 70$ which corresponds to 1000 revolutions of the harmonic oscillator. Setting $M = 2$ and $N = 100$ and applying the modified parareal algorithm to (8) produces the results show in Figure 1a, and converges in just one iteration. In Figure 1a, the converged solution $\{U_n\}$ and the initial approximation made by the coarse solver, $\left\{U_n^0\right\}$, lie on top of each other, because the initial approximation made by $\mathcal{G}^{map}$ is exact. In fact, the results would be the same for any value of N. Recall that N sets the step size of the coarse solve, $\Delta T = T/N$. This is not the case if the traditional parareal scheme is applied to (8). For N smaller than around 10000, the algorithm diverges. For $N = 10000$, the algorithm converges in 42 iterations; however, the coarse step size is almost as small as the fine step size, so any speedup from the parareal algorithm is lost. The results of the traditional parareal algorithm are shown in Figure 1b; the initial approximation made by the coarse solver is far away from the true solution so many more parareal iterations are needed for convergence.



(a)



(b)

Figure 2: (a) Results of the modified parareal scheme applied to (10). (b) Results of the traditional parareal scheme applied to (10).

These results for a simple harmonic oscillator motivate the study of physical systems that behave in a similar manner, such as those found in molecular dynamics or celestial mechanics.

3.3 Colinear Lennard-Jones System

Consider two colinear atoms bound by the Lennard-Jones Potential:

$$v(r) = 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] , \tag{9}$$

where r is the separation of the atoms, ε is the 'well depth' and σ is the distance at which the potential is zero. We set $\varepsilon = 1$ and $\sigma = 1$, which gives a minimum potential energy at $r^* = 1.12$. The Lennard-Jones potential is shown in Figure 3. The equation governing this system is given by:

$$r'' = \frac{2}{m} F(r) \tag{10}$$

where $F(r) = -\nabla v(r)$. Near the minimum potential energy, equation (10) behaves very similar to a harmonic oscillator. Further away, the system behaves nonlinearly.

First we set the initial condition to $\mathbf{r}_0 = [1.2, 0]^T$, $T = 50$, and $N = 400$. This corresponds to around 100 oscillations of the atoms. Applying the traditional parareal method to (10) gives the results shown in Figure 2b. As in the case of the harmonic oscillator, using a standard RK4 scheme as the coarse solver makes a poor initial approximation of the true solution, which results in a larger number of iterations needed for convergence; in this case, 53 iterations. The modified parareal algorithm with $\mathcal{G}^{map}$ defined through linear interpolation with $M = 50$ converges in just 10 iterations when applied to (10). Figure 2a shows the results. The phase plane map makes a very accurate initial approximation of the solution, so the fine solver only needs a few iterations of corrections to converge to the true solution.

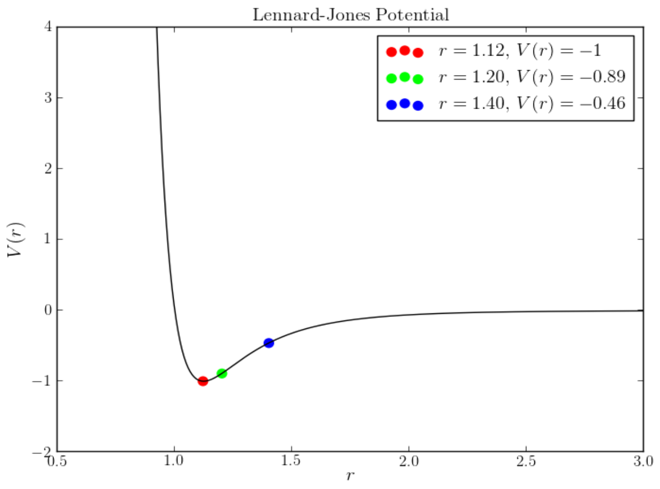


Figure 3: Lennard-Jones potential given by (9).

Unlike the linear harmonic oscillator, the efficiency of the modified parareal algorithm can be affected by defining the phase plane map with more or less training and target points $\{u_0^i \to v^i\}_{i=1}^{M^d}$. Figure 4 indicates how the number of parareal iterations needed for convergence decreases to one as the number of grid points gets arbitrarily large. Since the computation of $\{u_0^i \to v^i\}_{i=1}^{M^d}$ is embarrassingly parallel and done in a pre-processing step, the only adverse cost in increasing M comes from how one defines $\mathcal{G}^{map}$, e.g., interpolation will be more expensive as M increases.

The nonlinearity of the Lennard-Jones system also affects the performance of the modified parareal algorithm. Consider the three separation distances shown in Figure 3, $r = 1.12, r = 1.2$ and $r = 1.4$. As r increases, (10) becomes increasingly nonlinear. Table 1 shows how this nonlinearity affects the efficiency of the modified parareal algorithm. As r increases, so must M in order to maintain roughly the same number of parareal iterations for convergence.

Table 1: Tables showing how the nonlinearity of the Lennard-Jones system affects the efficiency of the modified parareal algorithm. Selected r values correspond to those found in Figure 3.

(●) $r = 1.12$

M	Iterations
3	7
5	3
10	2
50	1

(●) $r = 1.2$

M	Iterations
10	19
50	10
100	8
500	4

(●) $r = 1.4$

M	Iterations
50	207
100	39
500	14
1000	9

As in the simple harmonic oscillator case, one can improve the results of the traditional parareal algorithm by making the step size for the coarse solver smaller and smaller; however, as the coarse step size approaches the fine step size, any speed up of the parareal algorithm is lost. With the phase plane map used as the coarse solver, the coarse step size can stay significantly larger than the fine step size, while still achieving good results. For example, suppose we increase the total time in the Lennard-Jones simulation shown in Figure 2 to $T = 500$ and take $M = 1000$. If we use the phase plane map as the coarse solver and take $N = 400$, the parareal algorithm converges in 14 iterations. If we were to use the traditional parareal algorithm and also wanted convergence in 14 iterations, we must take $N \approx 50{,}000$. At this point, the coarse solver is almost as expensive as the fine solver. Furthermore, N is typically chosen to be the number of cores used for the parareal algorithm, which implies less cores are needed for the modified parareal algorithm than the traditional parareal algorithm to achieve the same efficiency.

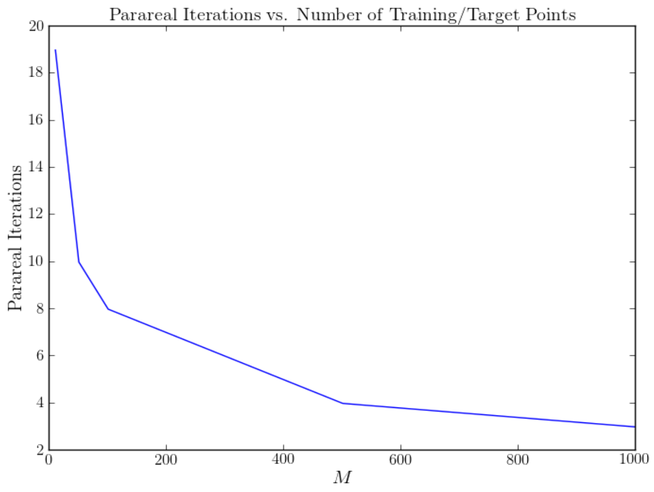


Figure 4: The number of the parareal iterations needed for convergence vs. the number of points used to define $\mathcal{G}^{map}$.

3.4 High Dimensional Harmonic Oscillator

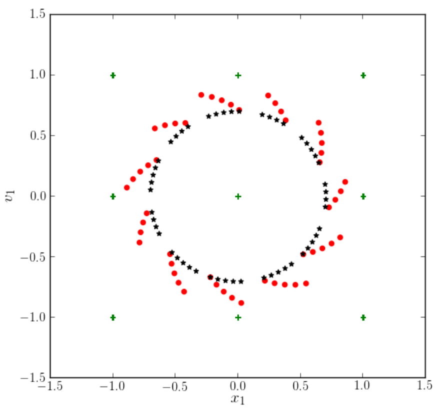
In high dimensions defining $\mathcal{G}^{map}$ through interpolation may be infeasible. This presents an interesting opportunity to make use of neural networks. As a testing case, consider a high dimensional system of harmonic oscillators:

$$\dot{\mathbf{x}} = \frac{1}{\varepsilon} A\mathbf{x} \tag{11}$$

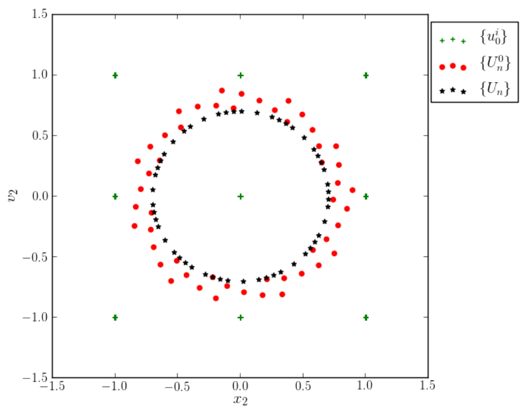
where $x = [x_1, v_1, x_2, v_2, x_3, v_3, x_4, v_4]^T$, $\varepsilon = 0.01$, and $A \in \mathbb{R}^{8\times 8}$ is defined as,

$$A = \begin{bmatrix} 0 & a & 0 & 0 & 0 & 0 & 0 & 0 \\ -a & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & b & 0 & 0 & 0 & 0 \\ 0 & 0 & -b & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & c & 0 & 0 \\ 0 & 0 & 0 & 0 & -c & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & d \\ 0 & 0 & 0 & 0 & 0 & 0 & -d & 0 \end{bmatrix} \tag{12}$$

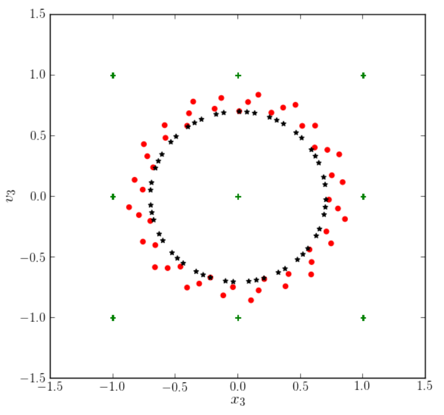
for parameters a, b, c, d. Since this system is linear, the modified parareal algorithm with $\mathcal{G}^{map}$ defined through interpolation will converge in just one iteration for any N and $M \geq 2$. However, high dimensional interpolation is quite expensive, especially if we were to consider a nonlinear problem.



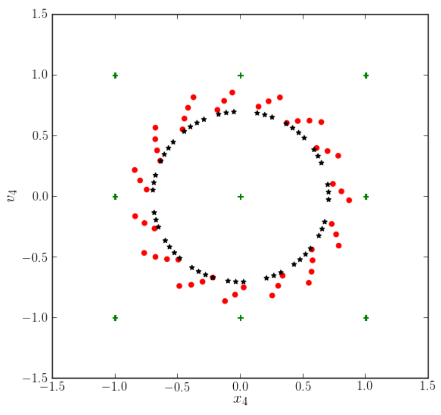
(a)



(b)



(c)



(d)

Figure 5: (a) Results of the modified parareal scheme applied to (11) with $\mathcal{G}^{map}$ defined with a neural network.

An alternative approach is to define $\mathcal{G}^{map}$ through a neural network. Below we test this approach on (11) setting $T = 10$, $N = 50$, $M = 3$, and take $a = 1, b = 2, c = 3$, and $d = 4$. The neural network is defined with a logistic activation function and one hidden layer of size 1000; a limited memory BFGS solver is used for optimization (Pedregosa, Varoquaux, et al. 2011). The results of this scheme applied to (11) are shown in Figure 5. The algorithm converges in 6 iterations, as opposed to one iteration when linear interpolation was used, however, the neural network approach converges in roughly a third of the time. The initial approximation of the neural network, while not exact, is still very close to the true solution, so only a few number of parareal iterations are needed for correction. Moreover, the neural network can be improved by increasing M, as in section 3.3, and has the additional advantage that the cost of applying the neural network does not increase with M, as opposed to the case when interpolation is used. This makes defining $\mathcal{G}^{map}$ through a neural network particularly advantageous for high dimensional problems, as all additional cost is made in the preprocessing step, which is embarrassingly parallel. Using the output of a neural network as a macroscopic model and coupling the result with a fine scale microscopic model for corrections is an interesting approach for solving physical systems in general, and may help bridge the gap between data science and computational science in the future.

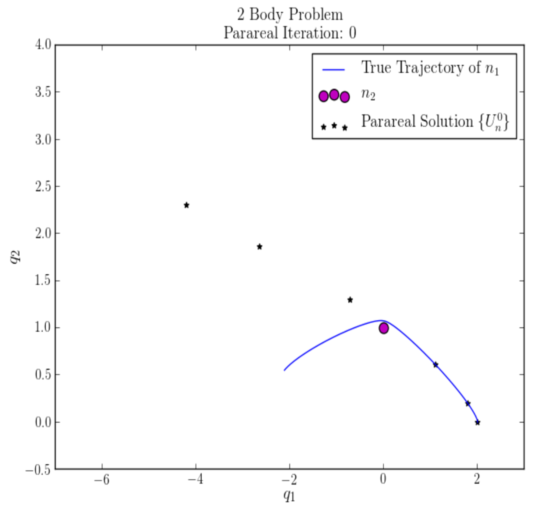
3.5 Localized Multiscale Problems

E, Engquist, et al. (2003) and E (2011) describe two types of multiscale problems. The first type are problems that contain local defects or singularities so that a macroscale model is sufficient for most of the physical domain, and a microscale model is needed only near the defects. These type of problems are known as *type A* multiscale problems. The second type of multiscale problems are those for which a microscale model is needed everywhere. These are known as *type B* multiscale problems. In the previous sections we have only consider type B multiscale problems, and in fact, this is the situation in which most parareal algorithms are applied, since the fine solver and coarse solver are coupled everywhere in the domain. The coarse solver acts as the macroscale solver and picks up on slow scale motion, while the fine solver acts as the microscale solver and corrects for fast scale motions. An important note can be made about the parareal algorithm in general applied to type A multiscale problems.

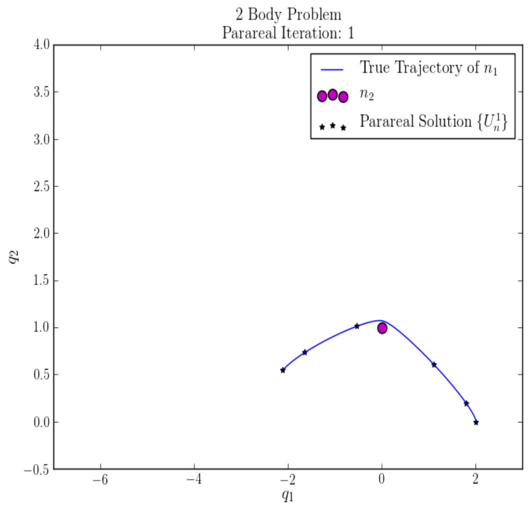
A good example of a type A multiscale problem can be found in celestial mechanics. In the context of an n-body problem, let m_i denote the mass of the i^{th} body, and $\mathbf{q}_i$ denote the position of the i^{th} body. The equation governing the motion of the i^{th} body is

$$m_i\ddot{\mathbf{q}}_i = \sum_{j\neq i}^{n} \frac{m_i m_j(\mathbf{q}_j - \mathbf{q}_i)}{\|\mathbf{q}_j - \mathbf{q}_i\|^3} \tag{13}$$

For illustrative purposes, let $n = 2$ in (13) and denote the two bodies as n_1 and n_2. Assume that $m_2 >> m_1$ so that $\mathbf{q}_2$ is essentially fixed. When the first body is far away from the second body, a macroscopic solver is accurate enough to predict its motion. However, as the two bodies get closer, a fine scale solver is needed to resolve their interactions. This situation is well-suited for the parareal algorithm. The coarse solver will be accurate enough in all regions far from n_2, so that the only corrections needed occur in regions where n_1 and n_2 are close. We apply a traditional parareal algorithm to (13) where the fine scale solver is an adaptive RK45 scheme and the coarse solver is a RK4 scheme with large step size. If we set $T = 0.5$ and $N = 5$, the parareal algorithm converges in just two iterations. The true solution and the parareal solution



(a)



(b)

Figure 6: (a) Initial coarse approximation given by the parareal algorithm applied to (13). (b) Results after one correction step of the parareal algorithm applied to (13).

for the trajectory of n_1 after each iteration is shown in Figure 6. The initial approximation made by the coarse solver (shown in Figure 6a) accurately predicts the motion of n_1 during the first two time intervals Ω_0 and Ω_1. During Ω_3, n_1 interacts with n_2 and alters its trajectory. The coarse solver does not detect this interaction due to the large step size, and incorrectly predicts the motion of n_1 for Ω_3, Ω_4 and Ω_5. After one iteration of the parareal solution, the fine solver corrects for the interaction between the two bodies during Ω_3, and when the coarse solver is run again, it is able to accurately predict the motion of n_1 far from n_2 as before. Therefore, the solution converges in just one iteration of the parareal algorithm. The fine solver is only needed to detect the localized interaction of n_1 and n_2.

4 CONCLUSION

We have presented the methodology behind a novel coarse scale solver for the parareal computation of highly oscillatory dynamical systems. This paper can be seen as a proof of concept. Further research is required to improve this approach into a fully developed robust and successful technique. Potential improvements are using modern sparse grid and adaptive methods for the interpolation, or optimizing the approach for neural networks. We plan on performing numerical tests of large systems with more realistic examples in the realm of molecular dynamics as well as determining scaling results on modern supercomputing platforms, even if the embarrassingly parallel nature of the algorithms should generate predictable scaling.

REFERENCES

Barker, A. T. 2014. “A minimal communication approach to parallel time integration”. *International Journal of Computer Mathematics* vol. 91 (3), pp. 601–615.

E, W. 2011. *Principles of multiscale modeling*. Cambridge University Press.

E, W., B. Engquist et al. 2003. “The heterognous multiscale methods”. *Communications in Mathematical Sciences* vol. 1 (1), pp. 87–132.

Gander, M. J. 2015. “50 years of time parallel time integration”. In *Multiple Shooting and Time Domain Decomposition Methods*, pp. 69–113. Springer.

Gander, M. J., and E. Hairer. 2014. “Analysis for parareal algorithms applied to Hamiltonian differential equations”. *Journal of Computational and Applied Mathematics* vol. 259, pp. 2–13.

Gander, M. J., E. Hairer et al. 2008. “Nonlinear convergence analysis for the parareal algorithm”. *Lecture Notes in Computational Science and Engineering* vol. 60, pp. 45.

Jones, E., T. Oliphant et al. 2001. “SciPy: Open source scientific tools for Python”.

Lions, J.-L., Y. Maday et al. 2001. “A parareal method in time discretization of pde’s”. *Comptes Rendus de l’Académie des Sciences-Series I-Mathematics* vol. 332 (7), pp. 661–668.

Nievergelt, J. 1964. “Parallel methods for integrating ordinary differential equations”. *Communications of the ACM* vol. 7 (12), pp. 731–733.

Pedregosa, F., G. Varoquaux et al. 2011. “Scikit-learn: Machine Learning in Python”. *Journal of Machine Learning Research* vol. 12, pp. 2825–2830.

Ying, L., and E. J. Candès. 2006. “The phase flow method”. *Journal of Computational Physics* vol. 220 (1), pp. 184–215.

AUTHOR BIOGRAPHIES

GOPAL R. YALLA is a Graduate Student at the Institute for Computational Engineering and Sciences at the University of Texas at Austin. His research interests lie in multiscale modeling and simulation, turbulence modeling, high performance computing, and parallel algorithms. gopal@ices.utexas.edu.

BJORN ENGQUIST holds the Computational and Applied Mathematics Chair I at the Institute for Computational Engineering and Sciences (ICES) at the University of Texas at Austin, and is director of the ICES Center for Numerical Analysis. Before coming to ICES, Engquist received his Ph.D. in numerical analysis from Uppsala University, was a professor of mathematics at UCLA, and the Michael Henry Stater University Professor of Mathematics and Applied and Computational Mathematics at Princeton University. He was also the director of the Research Institute for Industrial Applications of Scientific Computing and of the Centre for Parallel Computers at the Royal Institute of Technology, Stockholm. Engquist's research focuses on the development and analysis of numerical methods for differential equations, computational multiscale methods, and fast algorithms for wave propagation with applications in seismology. engquist@ices.utexas.edu.

MODELING A TASK-BASED MATRIX-MATRIX MULTIPLICATION APPLICATION FOR RESILIENCE DECISION MAKING

Erik Jensen and Masha Sosonkina

Department of Modeling, Simulation and Visualization Engineering,
Old Dominion University,
4700 Elkhorn Ave, Norfolk, VA 23529, USA
{ejens005, msosonki}@odu.edu

ABSTRACT

Faults are the manifestation of errors, which corrupt data or cause program termination. Currently operational computing clusters already experience the ill effects of faults. As component size decreases in future generations of hardware, faults are projected to increase accordingly. One strategy to minimize the threat of unreliable hardware is to make resilient software. A distributed-memory block matrix-matrix multiplication (MMM) application is used to investigate the development of a decision model that defines application behavior following cluster hardware failure. The MMM application employs a task-based model; each task consists of the multiplication of matrix sub-blocks. This MMM implementation is modeled with distribution functions for worker computation time generation and master-worker communication time generation. Decisions based on such a model are useful, e.g., when a fault results in the loss of a worker and the resilient application may determine to continue and redo certain tasks or to restart from the beginning. The task-based MMM application runtime is predicted by simulating task completion through the master-worker relationship. For a sample size of 42 different configurations, simulated application runtimes are compared to cluster runtimes: considering the errors for all the configurations, the mean error is 1.43%, and the standard deviation is 1.38%. This model may be extended to other task-based applications.

Keywords: resilience, fault tolerance, task-based application, performance model, matrix-matrix multiply

1 INTRODUCTION

In high-performance computing (HPC), faults threaten software applications in various ways: (1) soft faults corrupt data, the deleterious effects of which can manifest later with varying severity; (2) hard faults, also known as fail-stop failures, abruptly terminate the application following cluster error (Dongarra, Herault, and Robert 2015). Energy from external radiation can cause bits of data or instructions in memory to convert from 1 to 0, or vice-versa, potentially leading to soft and hard faults that cause data corruption and hardware failure (Geist 2016). For a typical computing cluster, the mean time between failures (MTBF) of any particular node is much greater than the execution time of a typical application. However, the susceptibility of an application to hard faults increases as the application scales. As computing clusters continue to scale in size, the likelihood of hard faults within the cluster increases the need for application resilience.

Task-based applications are those that feature task-based parallelism: a large problem is decomposed into a set of smaller tasks that are completed by independent processing units. The tasks, which may be similar to or different from one another, are completed in an order such that task-dependency relations are maintained. Task-based applications are an asset to resilience in HPC because computational work is decomposed and encapsulated into discrete tasks that can be repeated if necessary, independently from other tasks. Some tools, such as PaRSEC (Bosilca, Bouteiller, Danalis, Herault, Lemarinier, and Dongarra 2011) and StarPU (Augonnet, Thibault, Namyst, and Wacrenie 2011), provide frameworks for creating, scheduling and managing the runtime of task-based applications on heterogeneous architectures. Another tool for HPC resilience is User Level Fault Mitigation (ULFM) (Dongarra, Herault, and Robert 2015), which is the Message Passing Interface (MPI) extension that permits application recovery after a fail-stop fault, which are the variety of faults that interrupt application execution and cause premature termination. Functionality provided by ULFM can permit an application to continue to completion after a hard fault, under certain conditions. Neither PaRSEC nor StarPU are currently able to use ULFM: The interfaces of these tools do not permit the user to incorporate ULFM functionality, as MPI function calls are abstracted away.

This work uses an example of a parallel matrix-matrix multiplication (denoted here as MMM) to develop a model for determining appropriate application behavior following a fail-stop failure. MMM is chosen for its simplicity in implementation and ubiquity in scientific and engineering applications. The parallel MMM algorithm employs blocking to decompose the calculation into tasks that are completed by multiple processing units, e.g. nodes within the computing cluster. Using message passing, the master processing unit distributes tasks to worker processing units, which perform computations and return results to the master. For the sake of modeling clarity, this work does not use ULFM, although a resilient MMM application based on ULFM was also developed and successfully tested on smaller cluster sizes.

This paper considers an example task-based application regarding post-fault behavior: the application may continue with reduced resources or terminate and restart from the initial point with restored resources. When the resilient application encounters a fault, it may utilize the decision model to predict time to completion for the two competing behaviors, which are chosen depending on the amount of resources available. The model considered here is specialized for a distributed-memory block matrix-matrix multiplication. Model validation is determined through comparison of predicted execution times to empirical results, that is, observed real cluster application runtimes. The methods used to develop and employ the matrix-matrix multiplication model may be generalized to any task-based, distributed-memory application, given that tasks are homogeneous and independent. An extended, but similar, model could accommodate heterogeneous, dependent tasks. In summary, the contributions of the present work include (1) modeling worker task computation times, and master-worker task communication times, for a task-based master-worker application and (2) validating model capabilities to predict execution time for the same application.

Related Work High-level programming models allow users to develop applications for complex, heterogeneous hardware that is also portable between architectures. A modern task-based runtime system implements such a programming model and abstracts away the lower-level hardware details for the user (Agullo, Aumage, Faverge, Furmento, Pruvost, Sergent, and Samuel 2016). The user can focus on algorithmic concerns that are architecture-independent, while the application can achieve good performance on a variety of platforms. The directed acyclic graph (DAG) of tasks is a conceptual model that represents tasks and dependencies with vertices and edges. The parameterized task graph (PTG) model and the sequential task flow (STF) model are high-level programming models that are used to implement a DAG of tasks. Task dependency may be declared explicitly by the user, or the runtime engine may determine dependencies through implied relations between tasks. PaRSEC (Bosilca, Bouteiller, Danalis, Herault, Lemarinier, and Dongarra 2011), based on the PTG model, is the engine that powers the highly optimized DPLASMA dense linear StarPU (Augonnet, Thibault, Namyst, and Wacrenie 2011) is an STF-based implementation. StarPU and PaRSEC are sophisticated task-based application frameworks that offer convenience to the user and platform flexibility, but they do not incorporate resiliency and cannot survive MPI process failure.

The rest of the paper is organized as follows: Section 2 describes the example MMM application used here and notes on its ULFM implementation. Section 3 presents the development of the runtime model, while Section 4 provides simulation results, and Section 5 concludes.

2 BLOCK MMM AS AN EXAMPLE OF A TASK-BASED APPLICATION

The block MMM application developed for this work serves as an example task-based application to demonstrate proof of concept. In the master-worker MMM application, each MPI process decomposes the MMM calculation into an identical set of block-MMM tasks. Prior to the calculation, each process reads the input matrices directly from NetCDF-4 files, which are opened and read in parallel collectively (Unidata 2017). The NetCDF files are located on Turing's Lustre file system for increased read performance. By making the source matrices directly available to each worker process, the master process does not send matrix data to worker processes during the calculation, which significantly reduces communication overhead. The master distributes a task by sending the task number to the worker, at which point the worker identifies the required blocks from the input matrices, which are already stored in memory. The application features hybrid parallelism: for testing, each MPI process is mapped to one NUMA node, which gives each MPI process access to multiple cores for shared-memory parallelism. Workers perform MMM operations using the Intel Math Kernel Library (MKL) dgemm function across the NUMA node; the master uses the MKL vmdAdd function to add partial solutions to the solution matrix. Figure 1 demonstrates how the resilient application functions at the highest architectural level.

2.1 Tasks

In the MMM application, one task is defined as the multiplication of one block from matrix A times one block-row of matrix B, given the calculation $AB = C$. The completion of the task produces one partial block-row solution in matrix C. As block size is assumed to evenly divide the input matrix size, the total number of tasks to complete the calculation is defined by $(\frac{s}{b})^2$, where s is the size of one dimension of either square matrix A or B, and b is the size of one dimension of the square block. A worker allocates memory for the MKL dgemm operation solution during the first task, which increases computation time. Because the first-task computation time for a worker tends to be greater than computation times for following tasks, data in Fig. 2 are worker computation times that exclude first tasks.

2.2 Block Size

The block size b affects performance: smaller block sizes reduce performance by increasing data reads and master-worker communication. Further, given two square blocks, each size b, a worker can perform b^3 floating-point operations (FLOPs). Larger block sizes increase processor utilization while decreasing data reads and communication. Conversely, smaller block sizes offer potentially better load balancing and greater resilience. In task-based applications, generally, larger tasks tend to leave some workers idle while other workers complete the computations required to advance to the next step or finish to completion. In the event of a fault, lost work may be less significant and computation may recover faster if tasks are small. In this work, blocks of sizes $2,000$ and $4,000$ are considered, given input matrices of size 4×10^5.

2.3 User Level Fault Mitigation

The resilient MMM application that uses the migration model requires two functions specific to ULFM, which enable it to recover from faults.

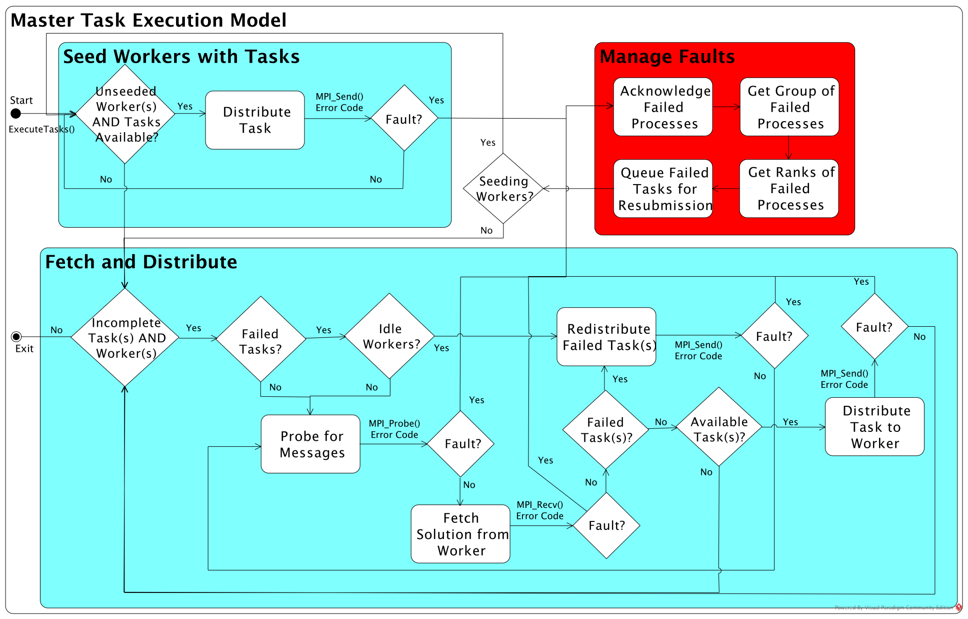


Figure 1: Resilient application activity diagram. The master process checks for faults with each send/receive. When a fault is detected, the master utilizes ULFM methods to continue the MPI application and determine which processes have failed.

1. `int MPI_Comm_failure_ack(MPI_Comm comm)` defines a reference point in time (Bland, Bouteiller, Herault, Hursey, Bosilica, and Dongarra 2012).
2. `int MPI_Comm_failure_get_acked(MPI_Comm comm, MPI _Group × failedgrp)` uses the reference point to retrieve the group of failed processes on `comm`, which can then be used to determine which MPI ranks have failed (Bland, Bouteiller, Herault, Hursey, Bosilica, and Dongarra 2012).

Figure 1 demonstrates how the resilient application functions with capabilities extended by ULFM. Master-worker applications are well suited for resilience, with the assumption that the master process is not vulnerable to faults.

3 MMM APPLICATION RUNTIME MODEL DEVELOPMENT

This work develops a runtime model, as a preliminary step to the decision model for the resilient application. The runtime model predicts the MMM calculation time, for a given block-worker configuration. Using data generated from several executions, empirical worker task computation times and master-worker task communication times are fitted to probability density functions with the SciPy statistics library. Computation time is defined as the time for the worker to perform the MKL dgemm computation for a single task. Communication time is defined as the time required for the master to retrieve a completed task from a worker, add the partial solution to the solution matrix, and then distribute the next task to the worker if applicable. The *generalized extreme value distribution* (GEV) is found to be an acceptable approximation for task computation and communication time modeling. GEV is used most commonly in climate science and hydrology, and has also been used in computational economics (Gellens 2002) (Lu and Stedinger 1992) (Gilli and këllezi 2006).

GEV distribution parameters for each block size are found from empirical data. Computation times depend on block size and processor type, and communication time depend on block size. Given that a resilient application (1) has the distribution parameters needed to sample computation and communication times, (2) can generate simulated task computation and communication times from the GEV distribution, and (3) can perform a simulation of MMM calculation by completing tasks, the resilient application may compare competing scenarios to determine whether it should continue to completion with reduced resources or terminate and restart with restored resources. Another alternative would be to checkpoint and restart with a set of completed tasks and restored resources. Investigation of the latter option is beyond the scope of this work.

Here, the efficacy of modeling computation and communications times, and using simulation to predict application runtime with simulated task times, is investigated and tested on the Turing computing platform, which is Old Dominion University's primary computer cluster. Turing contains 190 standard compute nodes, 10 GPU nodes, 10 Intel Xeon Phi Knight's Corner nodes, and 4 high memory nodes, connected with a Fourteen Data Rate (FDR) InfiniBand network (Old Dominion University 2017). Compute nodes contain between 16 and 32 cores and 128 Gb of RAM (Old Dominion University 2017). Some of the nodes utilized for this work contain two sockets of 10 Intel Xeon E5-2670 v2 2.50Ghz processors; the remaining nodes use the same socket configuration with Intel Xeon E5-2660 v2 2.20Ghz processors (Turing 2017).

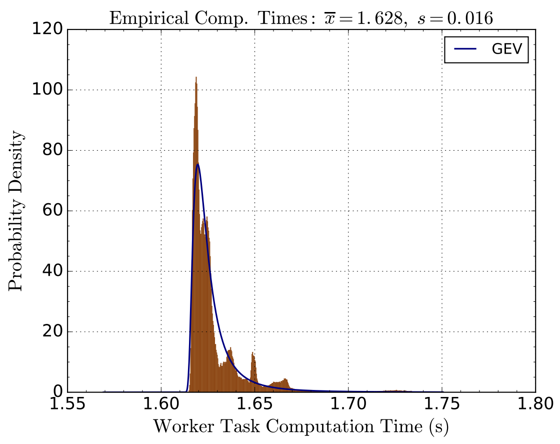
3.1 Distribution Fitting

Probability density functions are employed to generate simulated task computation and communication times that are representative of real system data. Empirical data is used to determine an appropriate distribution type suitable for all block-worker configurations. The GEV distribution function adequately models empirical task computation and communication times. Figures 2 and 3 depict the goodness of GEV fits to worker task computation times and task communication times.

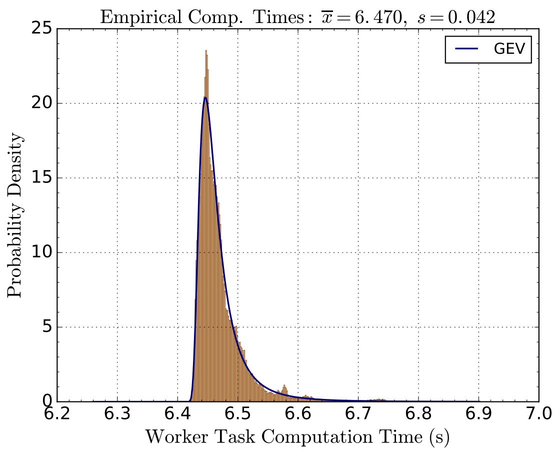
Other distribution functions are investigated for goodness of fit to empirical data and ease of random-variate generation with only mean and standard deviation data. Python 2.7 library SciPy is used to evaluate the efficacy of numerous continuous distributions for 22 datasets (11 worker quantities for each of two block sizes) compiled from multiple trials for each configuration (The Scipy community 2017). Maximum likelihood estimations (MLEs) are calculated for each distribution for each dataset and summed for each distribution across all datasets for general usefulness in all considered block-worker configurations. GEV performs comparably with many other distribution functions, considering MLE values, but compared to some alternatives, GEV more effectively fits empirical data and limits the generation of random variates that are not observed in the real system. Clustering behavior in communication time density is observed in Figure 3. While the communication data sets are not unimodal, that is, data points are aggregated in multiple clusters. This clustering likely results from Turing network topology. Though the model does not approach communication with such fine granularity as to consider network topology, it does not prohibit such an extension.

3.2 MMM Application Runtime Simulation

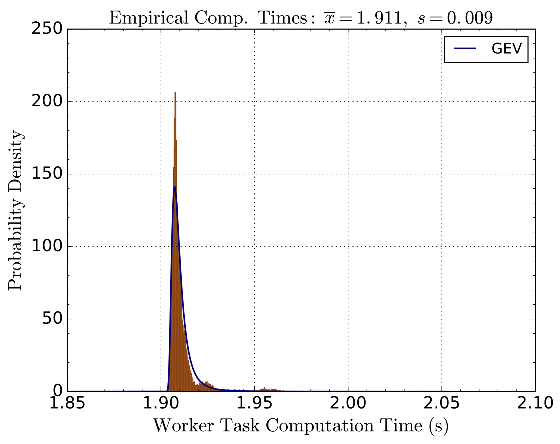
The simulation is implemented with an object-oriented Python script that attempts to mimic the behavior of the MPI application on Turing. The script generates computation and communication times from the generalized extreme value distribution (GEV) function, using parameters acquired from analysis of empirical data. For a given MMM application configuration, data acquired from several runs of the procedure described in subsections Sections 3.3 and 3.4 are compiled into a set of simulated MMM application runtimes that attempt to reflect real behavior on Turing. Real system MMM runtime variability results primarily from the set of nodes granted to a job by the SLURM scheduler. Therefore, the simulation model tests several node hardware configurations, to capture real system variability. Further simulation variability results from utilizing the task computation and communication GEV probability distribution functions, also reflective of real system behavior.



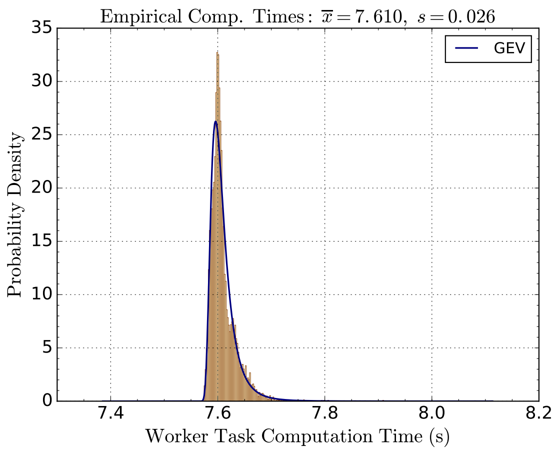
(a) block size 2000, coreV2-22-[001-024] nodes



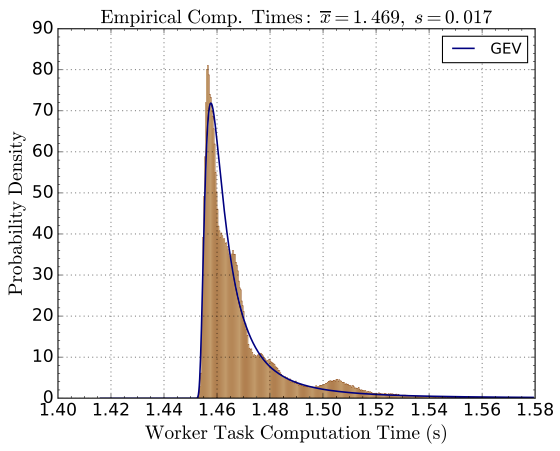
(b) block size 4000, coreV2-22-[001-024] nodes



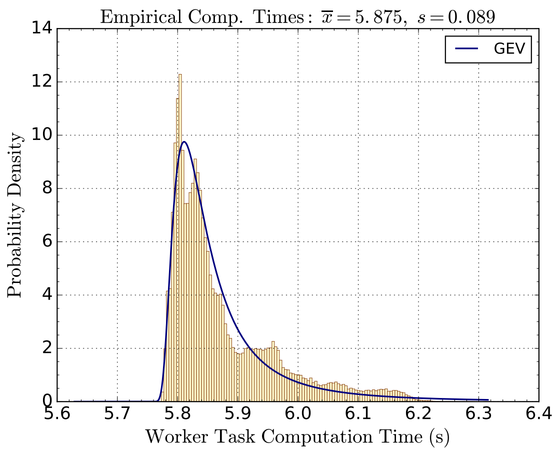
(c) block size 2000, coreV2-22-[025-034] nodes



(d) block size 4000, coreV2-22-[025-034] nodes

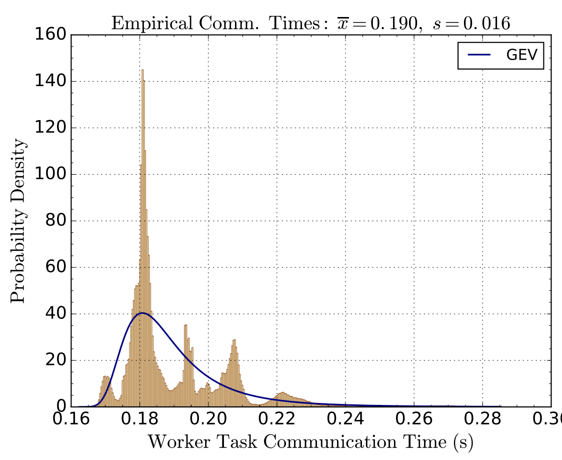


(e) block size 2000, coreV2-25 nodes

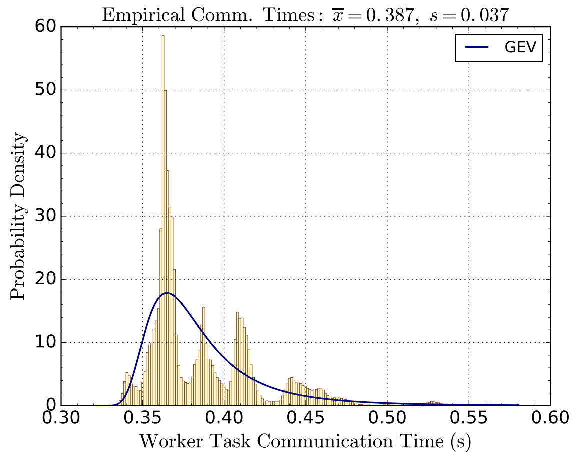


(f) block size 4000, coreV2-25 nodes

Figure 2: GEV density functions fitted to worker computation-time histograms. Histogram data are from several trials for each block-worker configuration. $\overline{x}$ and s denote population mean and standard deviation.



(a) block size 2000



(b) block size 4000

Figure 3: Generalized extreme value distribution density functions fitted to master-worker communication-time histograms. Histogram data are collected as mentioned in Fig. 2 caption.

3.3 Simulated Task Computation and Communication Time Acquisition

The application is tested on a subset of Turing's nodes, type coreV2. These nodes are further divided between types 22-[001-024], 22-[025-034], and 25. CoreV2-22 nodes contain two sockets of 10 Intel Xeon E5-2670 v2 2.50Ghz processors; coreV2-25 nodes contain two sockets of Intel Xeon E5-2660 v2 2.20Ghz processors CoreV2-22-[001-024] nodes exhibit distinct, faster performance than coreV2-22-[024-035] nodes. The cause for this different behavior is believed to be a result of the inclusion of Intel Dynamic Acceleration Technology (IDA) in coreV2-22-[024-035] nodes. IDA is a technology that increases the frequency of active cores when some cores are idle, which improves energy efficiency and single-thread performance (Nisar, Ekpanyapong, Valles, and Sivakumar 2008). IDA may be employed during multiplications, particularly for BLAS-2 operations.

SLURM assigns nodes for MPI applications from coreV2 such that coreV2-22 nodes are mapped to lower-ranking MPI processes than coreV2-25 nodes, given that both types of nodes are assigned to a job. For jobs using 24 MPI processes, the mixed node type configuration has been more commonly observed, but a purely coreV2-22 or coreV2-25 configuration is also possible. The simulation determines a fixed task computation time for each worker, to use throughout the simulated MMM computation. All possible hardware configurations are tested: (1) all coreV2-22 nodes, (2) all coreV2-25 nodes, and (3) $n-1$ configurations, given that n is the number of worker processes. For a set of nodes, the node type may switch once at any point as the MPI process rank increases. For example, for a set of three worker nodes, the node type may switch from type 22 to type 25 after either the first worker or after the second. For this example, there are two mixed node-type configurations and two pure node-type configurations. The simulation tests all four configurations. For each node type, two random variates are sampled from the appropriate task computation GEV probability distribution function (PDF) and are averaged to generate a simulated computation time. Sampled times that are deemed excessively large are redrawn and replaced with realistic samples. For each node using the corresponding node type sampled time, slight additional randomness is introduced to capture normal performance variablity among similar nodes. Two sampled task communication times are averaged to use for all nodes. Increasing the number of distribution samples used to generate computation and communication times decreases variability across runs.

3.4 MMM Simulation

Once the simulated task computation and communication times have been determined, the model simulates the calculation as described in Algorithm 1. The master seeds the workers with tasks, sending the current time with the task. The worker sums the current time and the computation time, immediately returning that value to the master queue of completed tasks. After the workers are seeded, the master parses the queue, sorting by smallest time. The master then fetches the first task, adjusting the master clock time according to one of two scenarios: (1) if the time reported by the worker is greater than the master, the master time is set to the worker time plus communication time; (2) else, if the time reported by the worker is less than the master, the master sets the clock to the current time plus communication time. The worker time may be less than the master time if the master has been processing tasks completed at similar times. If more tasks are available for distribution, the master assigns a new task to the worker, after adjusting the clock. This continues until all tasks have been completed. When all tasks have been completed, the master clock time is the simulated runtime for the MMM calculation.

Algorithm 1: MMM Simulation

```
Input: Worker nodes configuration: task compute times $t_{cp}[]$, communication time $t_{cm}$
Output: Calculation time $t_{calc}$
1  for each worker $w_i$, $i = 1,2\ldots n$ :=number of workers, do
2      Master seeds $w_i$ with first task $\tau_k, k :=$ tasknumber $= i$, master current time $t_m$.
3      Worker $w_i$ returns $\tau_i$, with task completion time $t_{\tau_i} = t_m + t_{cp_{w_i}}$.
4  for each completed task $T_j$, $j = 1,2,\ldots m$ :=number of tasks do
5      Master sorts queue, gets $T_j$ from $w_x$ such that $T_j$ has smallest task completion time $t_{\tau_k}$ in queue.
6      if current time $t_m <$ task completion time $t_{\tau_k}$ then
7          $t_m = t_{\tau_j} +$ communication time $t_{cm}$
8      else if current time $t >=$ task completion time $t_{\tau_k}$ then
9          $t_m += $ communication time $t_{cm}$
10     if number of tasks distributed $d <$ number of tasks $m$ then
11         Master distributes new task $\tau_k$, task number $k = d+1$, to worker $w_x$.
12         Worker $w_x$ returns $\tau_{d+1}$, with task completion time $t_{\tau_{d+1}} = t_m + t_{cp_{w_x}}$.
13 Calculation time $t_{calc} =$ current time $t_m$.
```

4 RUNTIME MODEL VALIDATION

Simulation results are compared to empirical data. For a sample size of 42 different MMM application configurations, simulated application runtimes are compared to cluster runtimes: considering the errors for all the configurations, the mean error is 1.43%, and the standard deviation is 1.38%. The greatest absolute percent error of all the MMM application configurations is 5.00%.

4.1 Results and Comparisons

Tables 1 and 2 display some simulation results for block sizes 2000 and 4000, and compare the simulated application runtimes to empirical data. The minimum, max, and mean simulation runtimes are the minimum, max, and mean runtimes generated by the simulation. The mean empirical runtime is the mean runtime of all cluster application runtimes for a given block-size and worker-number configuration. The mean percent error evaluates the error between simulated and real application runtimes. Error comparing minimum empirical runtime with minimum simulated runtime, and maximum empirical runtime with maximum sim-

Table 1: Simulation (columns `Sim`) vs. empirical (columns `Emp`) data for the 2000 block size. "W" denotes number of workers.

	Runtime, s								Error, %		
	`Sim`				`Emp`				`Sim` vs `Emp`		
W	Min.	Max.	Mean	Std.	Min.	Max.	Mean	Std.	Min.	Max.	**Mean**
3	228.91	273.24	249.54	12.66	223.54	249.22	237.66	8.75	2.4	9.64	**5.0**
5	137.29	168.43	151.39	6.99	136.91	152.02	144.29	4.68	0.28	10.79	**4.93**
7	103.17	120.34	109.33	4.84	98.34	111.56	104.48	3.63	4.91	7.87	**4.64**
9	80.85	94.8	86.59	3.99	77.34	94.78	85.31	3.56	4.54	0.02	**1.49**
11	69.27	95.35	78.4	4.7	72.38	90.5	78.9	5.16	-4.3	5.35	**-0.64**
13	70.86	93.17	77.39	4.11	72.43	89.16	78.15	5.0	-2.16	4.49	**-0.98**
15	71.03	93.08	77.44	3.92	72.95	90.44	74.98	2.52	-2.63	2.91	**3.28**
17	71.1	95.61	77.16	4.19	72.53	86.53	77.53	4.61	-1.97	10.49	**-0.48**
19	69.45	94.13	77.43	4.37	72.52	86.15	77.39	4.5	-4.23	9.27	**0.05**
21	70.71	95.68	77.75	4.38	72.53	85.46	78.21	4.6	-2.51	11.96	**-0.6**
23	70.0	94.04	77.36	4.46	72.37	85.14	77.34	4.43	-3.27	10.45	**0.03**

Table 2: Simulation (columns `Sim`) vs. empirical (columns `Emp`) data for the 4000 block size. "W" denotes number of workers.

	Runtime, s								Error, %		
	`Sim`				`Emp`				`Sim` vs `Emp`		
W	Min.	Max.	Mean	Std.	Min.	Max.	Mean	Std.	Min.	Max.	**Mean**
3	215.69	259.47	232.01	15.39	211.7	236.39	225.85	9.33	1.88	9.76	**2.73**
5	129.24	160.53	140.58	8.7	129.32	155.19	136.26	4.73	-0.06	3.44	**3.17**
7	93.12	115.34	101.88	6.27	94.24	106.32	100.55	3.92	-1.18	8.48	**1.33**
9	73.95	91.13	80.66	4.43	74.59	86.56	80.2	3.74	-0.86	5.29	**0.57**
11	62.17	75.98	67.56	3.61	62.18	75.84	66.86	3.89	-0.01	0.19	**1.05**
13	53.44	66.8	58.45	3.03	54.38	66.3	58.39	3.4	-1.73	0.76	**0.1**
15	47.69	59.96	52.3	2.7	48.41	57.41	51.63	2.27	-1.48	4.46	**1.3**
17	43.0	53.83	47.89	2.51	43.58	52.46	47.86	2.09	-1.33	2.62	**0.07**
19	40.91	53.91	45.55	2.21	41.76	51.73	45.8	2.42	-2.03	4.2	**-0.54**
21	40.95	53.45	44.61	2.22	41.92	51.1	45.11	2.41	-2.31	4.59	**-1.12**
23	40.31	54.05	44.52	2.49	41.91	51.0	44.81	2.34	-3.81	5.98	**-0.66**

ulated runtime, are also provided. The standard deviation of simulated runtimes and empirical runtimes are available for comparison, but sample sizes are not necessarily equivalent.

5 CONCLUSIONS

The application runtime model successfully simulates MMM calculation performance for all tested block-worker configurations. While the model performs adequately, further investigation will attempt to explain peculiarities in observed real task computation and communication times, and thus to improve the model. This work lays the foundation for the decision model to be used in resilient applications. Simulation results suggest that the model is capable of predicting partial calculation runtimes, i.e. to provide the amount of time to finish the MMM calculation with reduced workers, as an alternative to restarting the calculation, for the decision model in the resilient MMM application. When the resilient MMM application detects a fault, specifically the loss of an MPI worker process, it may utilize the decision model to compare the runtime predictions for restarting with restored nodes versus completing the calculation with reduced resources. The recent release of ULFM 2.0 may facilitate resilient application development.

Future computing platforms will rely increasingly on scaling for performance benefits. A multiple-master approach, in which task administration is distributed among several masters, may permit a resilient master-worker application to function efficiently at a larger scale. Employing multiple masters has been found to not hinder checkpointing and recovery in resilient applications (He, Watson, and Sosonkina 2007). Future work will explore the development of a multiple-master model.

ACKNOWLEDGMENTS

This work was supported in part by the Air Force Office of Scientific Research under the AFOSR award FA9550-12-1-0476, by the U.S. Department of Energy, Office of Advanced Scientific Computing Research Exascale Computing Project, through the Ames Laboratory, operated by Iowa State University under contract No. DE-AC02-07CH11358, and by the U.S. Department of Defense High Performance Computing Modernization Program, through a HASI grant. This work was also supported by the Turing High Performance Computing cluster at Old Dominion University.

REFERENCES

Agullo, E., O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and T. Samuel. 2016, June. "Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model". *Research Report, Inria Bordeaux Sud-Ouest* vol. 8927, pp. 27.

Bland, W., A. Bouteiller, T. Herault, J. Hursey, G. Bosilica, and J. Dongarra. 2012. "An Evaluation of User-Level Failure Mitigation Support in MPI". In *Lecture Notes in Computer Science*. Springer International Publishing.

Dongarra, J., T. Herault, and Y. Robert. 2015. "Fault Tolerance Techniques for High-Performance Computing". In *Fault-Tolerance Techniques for High-Performance Computing*. Springer International Publishing.

Geist, A. 2016, March. "Supercomputing's monster in the closet". *IEEE Spectrum* vol. 53 (3), pp. 30–35.

Gellens, D. 2002. "Combining regional approach and data extension procedure for assessing GEV distribution of extreme precipitation in Belgium". *Journal of Hydrology* vol. 268 (1), pp. 113 – 126.

Gilli, M., and E. këllezi. 2006, May. "An Application of Extreme Value Theory for Measuring Financial Risk". *Computational Economics* vol. 27 (2), pp. 207–228.

Jian He and Layne T. Watson and Masha Sosonkina 2007. "Algorithm XXX: VTDIRECT95: Serial and Parallel Codes for the Global Optimization Algorithm DIRECT. Association for Computing Machinery".

IEEE 2011. *DAGuE: A Generic Distributed DAG Engine for High Performance Computing*. IEEE, IEEE.

Lu, L.-H., and J. R. Stedinger. 1992. "Sampling variance of normalized GEV/PWM quantile estimators and a regional homogeneity test". *Journal of Hydrology* vol. 138 (1), pp. 223 – 245.

Nisar, A., M. Ekpanyapong, A. C. Valles, and K. Sivakumar. 2008. "The Original 45nm Intel Core Microarchitecturem". *Intel Technology Journal* vol. 12 (3).

Old Dominion University 2017 (accessed November 22, 2017). *High Performance Computing - Old Dominion University*. Old Dominion University.

The Scipy community 2017 (accessed November 22, 2017). *Statistical functions (scipy.stats) – SciPy v1.0.0 Reference Guide*. The Scipy community.

Turing 2017 (accessed November 22, 2017). *Turing SSH Session*. Old Dominion University.

Unidata 2017 (accessed November 22, 2017). *NetCDF: Parallel I/O with NetCDF-4*. Unidata.

Wiley 2011. *StarPU: a unified platform for task scheduling on heterogeneous multicore architectures*. Wiley, Wiley.

HYBRID MPI+OPENMP PARALLELIZATION OF IMAGE RECONSTRUCTION IN PROTON BEAM THERAPY ON MULTI-CORE AND MANY-CORE PROCESSORS

James Della-Giustina

School of Mathematics and Sciences
Community College of Baltimore County
800 S. Rolling Rd., Catonsville, MD, USA
jdella@umbc.edu

Carlos Barajas

Department of Mathematics and Statistics
University of Maryland, Baltimore County
1000 Hilltop Circle, Baltimore, MD, USA
barajasc@umbc.edu

Matthias K. Gobbert

Department of Mathematics and Statistics
University of Maryland, Baltimore County
1000 Hilltop Circle, Baltimore, MD, USA
gobbert@umbc.edu

Dennis S. Mackin

Department of Radiation Physics
The University of Texas MD Anderson Cancer Center
1515 Holcombe Blvd., Houston, TX, USA
dsmackin@mdanderson.org

Jerimy Polf

Department of Radiation Oncology
University of Maryland School of Medicine
655 W. Baltimore Street, Baltimore, MD, USA
jpolf@umm.edu

ABSTRACT

The advantage of proton beam therapy is that the lethal dose of radiation is delivered by a sharp increase toward the end of the beam range, known as the Bragg peak (BP), with no dose delivered beyond. By using these characteristics of the BP, radiation dose to the tumor can be maximized, with greatly reduced radiation dose to the surrounding healthy tissue. If the secondary gamma rays that are emitted through interaction of the protons in the beam with atoms in the patient tissue could be imaged in (near) real-time during beam delivery, it could provide a means of visualizing the delivery of dose for verification of proper treatment delivery. However, such imaging requires very fast image reconstruction to be feasible. This project focuses on measuring the performance of a new parallel version of the CCI (Compton camera imaging) image reconstruction algorithm. We show two conclusions: (i) The new hybrid MPI+OpenMP parallelization of the code on the many-core Intel Xeon Phi KNL processor with 68 computational cores makes fast reconstruction times possible and thus enables the use of CCI in real time during treatment. (ii) A compute node with two of the latest multi-core Intel Skylake CPUs with 24 cores performs even better in a first comparison of both types of processors available on Stampede2 at the Texas Advanced Computing Center (TACC).

Keywords: Proton beam therapy, Image reconstruction, CCI algorithm, Intel Xeon Phi, Intel Skylake.

1 INTRODUCTION

Proton beam radiation treatment was first proposed by Robert Wilson in 1946 (Wilson 1946). While x-ray radiation delivers its lethal dose throughout the patient, proton beams reach their highest dose just before they stop, at what is called the Bragg peak (BP), with little or no dose delivered beyond this point. The depth at which the BP occurs in the patient can be controlled by carefully choosing the beams initial energy. This gives the physicians and radiotherapy specialists the ability to match the depth of the BP within the patient to that of the tumor. This allows for the accurate delivery of a lethal dose of radiation to the tumor while allowing the radiation exposure to the surrounding healthy tissues to be kept at acceptably low levels. While the potential to reduce irradiation of healthy tissue is a major advantage of proton beam radiotherapy, it is currently not possible to always utilize such an advantage. Over a full course of proton therapy (1 to 5 weeks), changes may occur inside the patient's body as both the surrounding tissue and tumor can swell or shrink in response to radiation, changing the size and relative position of the target area for treatment on a daily basis. Therefore, to target only tumor volume and avoid adjacent normal tissue (thus fully utilize the advantage of proton therapy), in vivo imaging of the treatment delivery would reduce uncertainties and allow the advantages of the Bragg peak to be fully exploited (Polf and Parodi 2015, Avila-Soto, Beri, Valenzuela, Wudenhe, Blenkhorn, Graf, Khuvis, Gobbert, and Polf 2015).

Nuclear scattering of the treatment beam protons produces secondary, gamma radiation. Because this gamma radiation is emitted only when the proton beam interacts with tissues, the origin of the gamma rays will map out the exact path of the proton beam through the body. The ability to image this gamma radiation would provide a method to image and verify the delivery of dose during each daily proton radiotherapy treatment. However, in order to provide these images to the physician and treatment specialists in real time during proton beam delivery, extremely fast image reconstruction techniques are needed. Thus, a limiting factor for implementing this type of imaging for treatment verification is the need for the image reconstruction code to produce results in essentially real-time, while the patient is undergoing treatment.

This paper investigates whether a new hybrid MPI+OpenMP parallelization of a statistical-reconstruction algorithm designed for secondary gamma imaging on modern multi-core or many-core processors can reliably reconstruct a sample image in a much less time than it takes to deliver a single beam (about 30 to 100 seconds) during daily treatment (Mackin, Peterson, Beddar, and Polf 2012). Previous work demonstrated that introducing parallelism with MPI, which allowed to use multiple compute nodes in a distributed-memory cluster, sped up the code to any desired time by using more nodes. But multi-node clusters cannot be housed in a treatment room and would require an off-site system with associated problems of reliable real-time access, such as risk of lost connections, in addition to additional training for physicians and imaging technicians, which make it simply impractical. Therefore, we investigate here if a single compute server, i.e., one node of a cluster, could accomplish the desired speedup. We show two conclusions: (i) We demonstrate that this is within reach and show that the many-core architecture of the Intel Xeon Phi KNL processor with 68 cores connected by a 2D mesh network can achieve this. Since the latest generation of the Phi, code-named Knights Landing (KNL), is a self-bootable CPU, it is realistically possible to use a computer with one KNL in a clinical setting. (ii) Moreover, we compare the performance of the KNL to a compute node with two of the latest multi-core Intel Skylake CPUs with 24 cores, which performs even better, provided MPI parallelism is used in the code. This is the first report of a comparison of both types of processors available on Stampede2 at the Texas Advanced Computing Center (TACC).

This paper is organized as follows: Section 2 contrasts available multi-core and many-core processors. Section 3 explains the reconstruction algorithm and its parallel implementation in hybrid MPI+OpenMP C/C++ code. Section 4 presents both application results of the image reconstruction and parallel performance studies on multi-core and many-core processors. Section 5 collects the conclusions of the work.

2 MULTI-CORE VS. MANY-CORE PROCESSORS

Modern CPUs feature growing number of computational cores, such as, for instance, 24 cores in an Intel 8160 Skylake CPU from 2017. Contrasted to these state-of-the-art multi-core CPUs with the most modern cores, many-core processors have even more cores, for instance, 68 cores in an Intel Xeon Phi 7250 Knights Landing (KNL) processor. The cores in a many-core processor typically feature a lower clock rate, e.g., 1.4 GHz vs. 2.1 GHz, but their larger number of cores with their rich connectivity in a 2D mesh network and the fact that significantly more high-performance memory is available on the chip itself offer the potential for better performance. Additionally, each core of a KNL is capable of running 4 hardware threads simultaneously. Investigating (i) this performance and (ii) comparing it to state-of-the-art multi-core CPUs are the purposes of this paper. Our results do not depend on the cluster Stampede2 at TACC, already since the focus is on single-node performance, but using both hardware in the same cluster ensures a fair comparison, for instance also with respect to software. We use the Intel compiler version 17.0.4 and the Intel MPI implementation version 17.0.4.

2.1 Many-Core Processor: Intel Xeon Phi 7250 KNL

Figure 1 illustrates the layout of one Intel Xeon Phi 7250 Knights Landing (KNL) processor, whose cores are clocked at 1.4 GHz. Each KNL core is capable of 4 hardware threads. This model of the KNL on Stampede2 contains 68 cores, which are arranged in pairs of two in a so-called tile that share the connection to the two-dimensional mesh network that connects all tiles. Each core has a L1 cache, while the two cores of a tile share an L2 cache. Each box without text inside (in red color) in Figure 1 indicates one tile with two cores and their shared L2 cache in the center of the tile between the cores. Inside the KNL are 16 GB of Multi-Channel DRAM (MCDRAM), arranged in 8 blocks of 2 GB each, shown on top and bottom of Figure 1. Outside of the socket, the KNL is connected to 96 GB of DDR4 memory, arranged in 6 DIMMs of 16 GB each, shown connected to their memory controllers (MC) on the left and right of Figure 1. This MCDRAM is located directly in the KNL and is up to five times faster than the on board DDR4 because of its three-dimensional arrangement (Sodani, Gramunt, Corbal, Kim, Vinod, Chinthamani, Hutsell, Agarwal, and Liu 2016).

The KNL uses three distinct memory modes, which determines how the processor treats the MCDRAM; as L3 cache, RAM, or a mixture of both. The KNL also offers three distinct cluster modes that ensure all cores possess the most current data among other processes, whether from main memory or MCDRAM. The differences between these cluster modes allow users to vary the amount of explicit control of memory management. For our studies, we chose to use the 'cache' memory mode and 'quadrant' cluster mode. We use the KNL in the Stampede2 cluster at the Texas Advanced Computing Center (TACC) at The University of Texas at Austin, which has over 4,200 nodes, each with one KNL processor.

2.2 Multi-Core Processor: Intel 8160 Skylake

The Stampede2 cluster at TACC has rolled out the new Intel Xeon Platinum 8160 Skylake CPUs in 2017, clocked at 2.1 GHz and with 32 MB L3 cache. These chips contain 24 cores, totaling 48 cores on a two-socket node. This node has 192 GB of memory available in 12 DIMMs of 16 GB DDR4 memory, surpassing the memory available on the KNL. However, its on-chip L3 cache does not compare to the 16 GB of high-speed MCDRAM that the KNL chip has. Stampede2 has 1,736 Skylake nodes, available since December 2017.

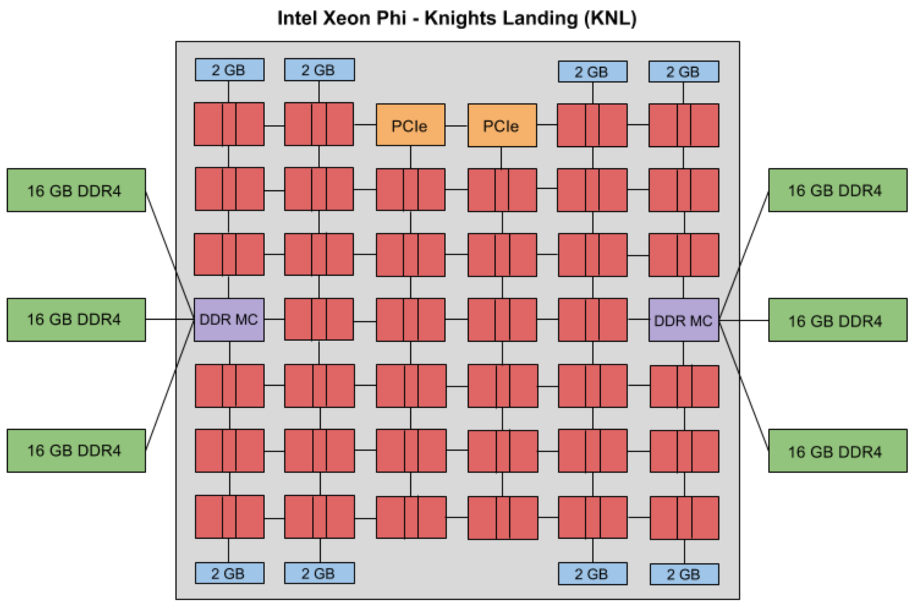


Figure 1: Intel Xeon Phi KNL schematic.

3 ALGORITHM AND IMPLEMENTATION

3.1 Compton Camera Imaging (CCI) Algorithm

During data collection, gamma rays scatter into a specially designed camera known as a "Compton Camera" (CC) which records the coordinates and energy deposited by each gamma ray that interacts with the CC. Each gamma ray must interact with the camera at least two times to be useful for imaging. The 3D coordinates and energies deposited by the gamma rays are stored in a data file which is used to initialize the conic image reconstruction software. A line is drawn between the two points of gamma ray interactions representing the central axis and using the energy deposition of the two interactions and the Compton scattering formula, an angle is calculated representing the half angle of the gammas interaction cone in Figure 2 (a). The cone's surface encompasses all the possible origins for that ray. After these cones have been constructed, a random point from the surface of the cone is chosen as an initial guess of the origin of the gamma ray. A 3D histogram, populated by the randomly chosen origins for each cone in Figure 2 (b), is used to estimate the gamma emission distribution for the reconstruction volume. Thus, one point on the surface represents each cone in the reconstruction.

The reconstruction algorithm, based on the Origin Ensembles method and later modifications (Andreyev, Sitek, and Celler 2011), uses a Markov process to iteratively move the representative points on the surface of the cones. After many iterations, the gamma emission density estimate formed by the representative points reaches a steady state and the algorithm is stopped.

3.2 Code Implementation

The algorithm is implemented in a C/C++ code with hybrid MPI+OpenMP parallelism. Structurally, parallel communication functions from MPI allow for the use of multiple nodes in a distributed-memory cluster, while OpenMP multi-threading parallelizes each MPI process within the shared-memory of a node. However, it is often the case, even on a single shared-memory node, that different combinations of MPI processes

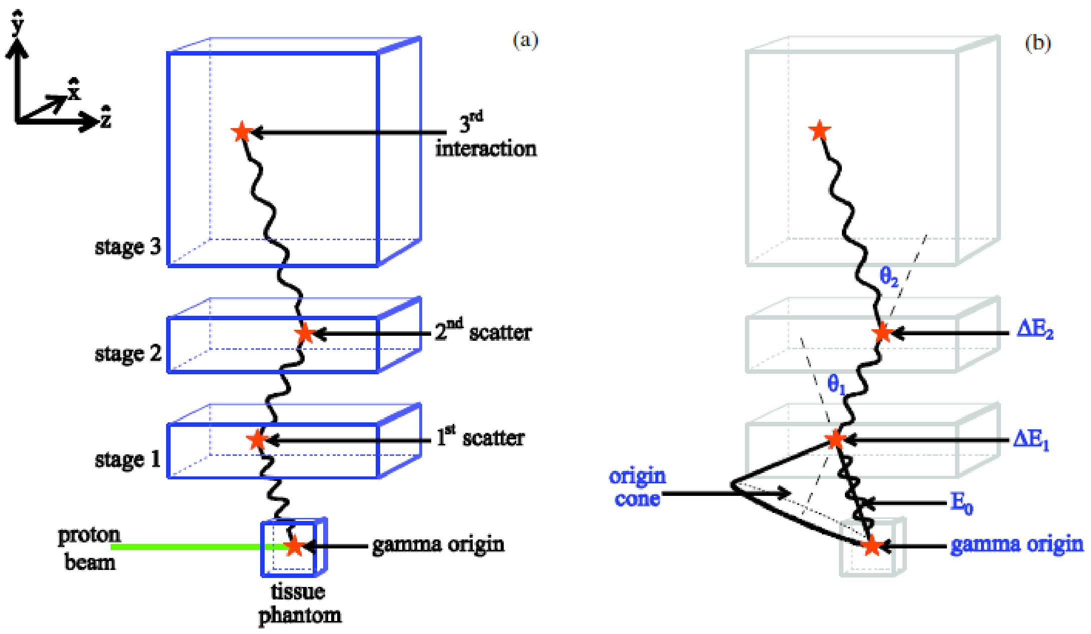


Figure 2: Gamma ray scatter and its image reconstruction.

and OpenMP threads per process result in different performance. In fact, for the KNL, it is a recommendation by TACC to use a modest number of MPI processes, combined with OpenMP multi-threading, instead of only OpenMP, even if the same number of hardware cores are used (Rosales, James, Gómez-Iglesias, Cazes, Huang, Liu, Liu, and Barth 2016). Thus, it is vital to have hybrid MPI+OpenMP code available for best performance.

First, a configuration file is parsed and data loaded from an input file with physical measurements. Important parameters include the total number of cones used, histogram coordinate boundaries in the x, y, and z directions, the total number of bins in the x, y, z directions, and the total number of iterations; notice that the number of iterations is fixed here by trial and error based on observing that the fraction of origins moved during each iteration stops changing. The total number of density histograms used for estimation is also set in the configuration file.

Before iterating, the collection of cones are distributed to the MPI processes as equally as possible. A global collection of before-iteration conic likely origins is generated and used for the initial population of the local density histograms on each process. An empty collection of after-iteration likely origins is created as well but not populated.

The CCI algorithm iteration is nearly identical to the original OpenMP algorithm as described in (Mackin, Peterson, Beddar, and Polf 2012). On each MPI process, OpenMP threads are started up using a `parallel for` pragma to iterate over the process' cones. During the iteration two situations will occur, either the thread will determine a cone's likely origin should be moved, or the cone's likely origin should remain where it was. If the cone's likely origin is to be moved, the new location is recorded into the process's local copy of the after-iteration collection. If the cone's likely origin is to remain the same, then the original likely origin is recorded into the process's local copy of the after-iteration collection.

At the end of the iteration, the local copies of the after-iteration collection are merged into a global version. Then all likely origins are updated for the local density histograms on each process. After all iterations have completed, all conic data is sent to the MPI process 0 for output to files.

4 RESULTS

4.1 Iterative Image Reconstruction

The CCI algorithm uses iterations to progressively compute the most likely origins of the 100,000 cones that were measured by the CC during the proton beam irradiation of a water phantom. The quality of the CCI algorithm is controlled also by the choice of the number of shifted histograms used, for which $h_{total} = 200$ has proven a good choice in the past. As more iterations are performed the iterated images become more clearly defined, as seen in Figure 3. Although at 100 iterations, the image has some shape, as the number of iterations increases, this shape becomes more accurate. Additionally as the number of iterations increases, the granularity of the results improve until the stopping point of 600 iterations, where a distinct beam can be seen.

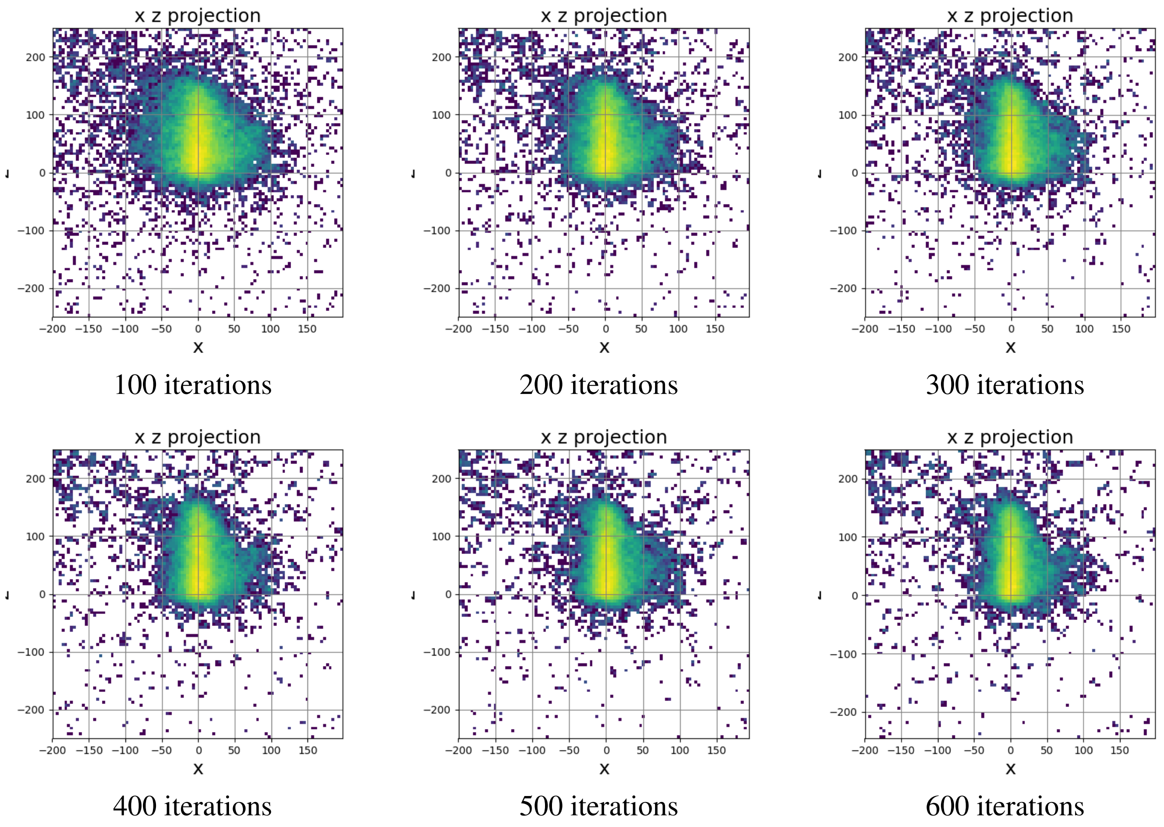
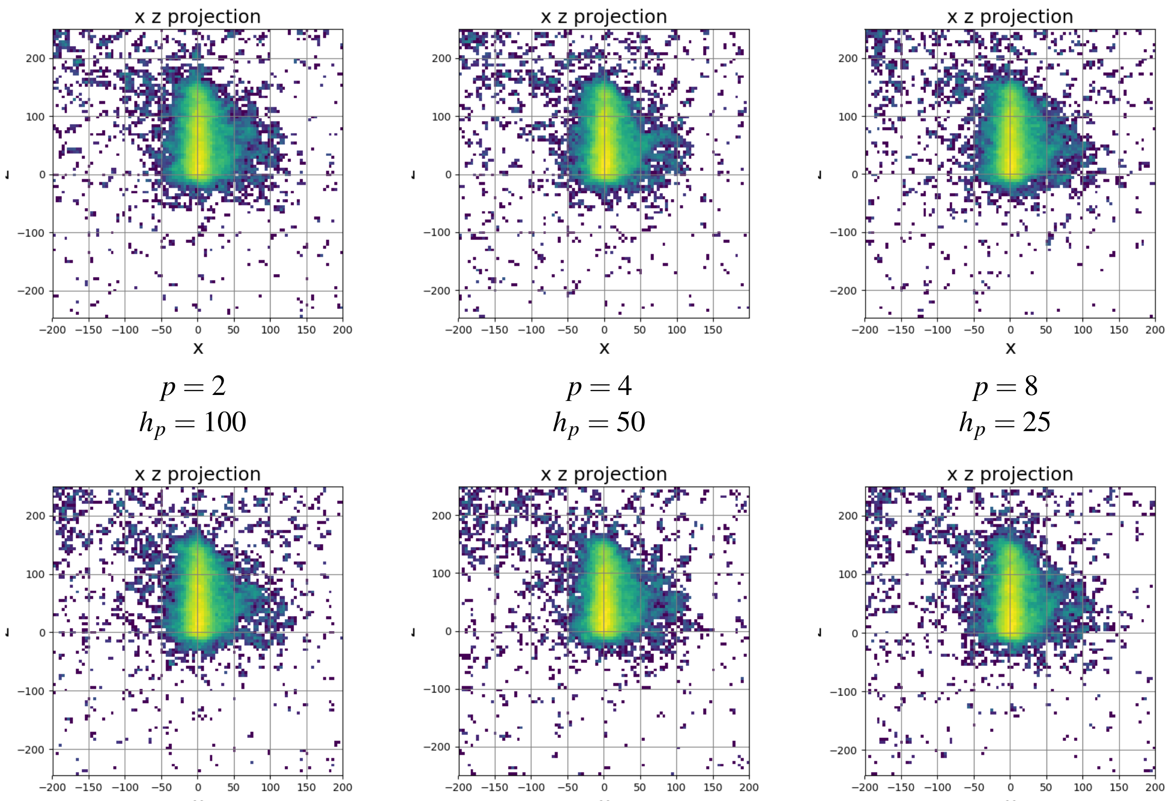


Figure 3: Reconstructed images (x-z-projections) at 100, ..., 600 iterations of the CCI algorithm using $h_{total} = 200$ shifted histograms, accessed by $p = 1$ process and $h_p = 200$ histograms per process.

The total number of shifted histograms, also referred to as total histograms controlled in the configuration file and denoted as h_{total}, is split among the p MPI processes, so that each process has the same number of shifted histograms per process denoted as h_p. The quality of the image reconstruction is controlled by the total number of $h_{total} = h_p\, p$ histograms; since the algorithm is iterative and involves random number sequences, and is thus not completely deterministic, it is possible to compare also results, for which h_{total} agrees only approximately. This is tested in Figure 4, which shows final images at 600 iterations of the CCI algorithm with the same number of $h_{total} = h_p\, p = 200$ histograms as in Figure 3, but accessed by different possible combinations of p and h_p. Comparing the 600 iteration image in Figure 3 with $p = 1$ process to the images in Figure 4 with $p = 2, 4, 8, 16, 32, 64$ processes, we can observe that any combination of p and h_p that satisfies approximately $h_{total} = h_p\, p \approx 200$ gives the same quality of the result.



$p = 2$
$h_p = 100$

$p = 4$
$h_p = 50$

$p = 8$
$h_p = 25$

$p = 16$
$h_p = 13$

$p = 32$
$h_p = 6$

$p = 64$
$h_p = 3$

Figure 4: Reconstructed images (x-z-projections) at 600 iterations of the CCI algorithm using $h_{total} = 200$ shifted histograms, accessed by various combinations of p processes and h_p histograms per process.

4.2 Single-Node Performance

In the clinical setting of a treatment room, the only realistic equipment is a computer server that is equivalent to a single, possibly high-end, node in a computer cluster, such as the ones used in our studies.

Tables 1 and 2 compare the timing performance of the studies in Section 4.1 with 100,000 cones and 600 iterations of the CCI algorithm. The times reported measure the performance of the variable portion of the algorithm, that is, the iterations of the CCI algorithm. There are additional times for initialization and for output (for post-processing), which are independent of the parallelization; however, they turn out to be significantly different in scale for the two platforms, namely about 40 seconds for the KNL and about 10 seconds for the Skylake.

Table 1 shows the performance results on one KNL processor. Recall that each KNL on Stampede2 has 68 computational cores. Thus, to take advantage of all cores when using p MPI processes, hybrid MPI+OpenMP code uses the number of threads per MPI process t_p chosen such that the total number of threads $t_{total} = t_p\, p = 68$. To be precise, the above holds exactly true for $p = 1,2,4$ only. For $p = 8,16,32,64$, for which the product $t_{total} = t_p\, p = 68$ cannot hold exactly, we actually idle some cores by using $t_{total} = t_p\, p = 64$ for these cases. Each core of a KNL is capable of running simultaneously up to 4 hardware threads. Thus, we can also choose the threads per MPI process t_p such that the total number of threads $t_{total} = t_p\, p = 136$ or 272 for running 2 or 4, respectively, threads per KNL core. In this way, each row of Table 1 uses the number of total threads t_{total} indicated in the first column. The difference between the columns lies in how the same number of t_{total} threads is accessed by combining p MPI processes with t_p threads per process. The total number of threads are as scattered as possible across all 68 hardware cores of the 34 tiles.

Each row of results in Table 1 indicates that using larger numbers of MPI processes p yields better timings, except some outlying behavior at $p = 32$ for the cases of 2 and 4 threads per MPI process. The improving run times, certainly for the case of 1 thread per MPI process, indicates that the 2D mesh network can facilitate communications between MPI processes very efficiently. The improvement of run times with growing MPI process count is not just explained by potential advantages in behavior of shared-memory multi-threading, but also by the algorithmic difference of correspondingly different numbers of shifted histograms per process. Recall these seven cases are the same simulations as in Figure 3 with $p = 1$ and in Figure 4 with $p = 2,4,8,16,32,64$, and we established already that their simulations yield equivalent, acceptable results.

The purpose of Table 1 is to determine if there is any advantage to running multiple hardware threads per KNL core. As it becomes clear, our experience is consistent with many other recommendations including TACC documentation (Rosales, James, Gómez-Iglesias, Cazes, Huang, Liu, Liu, and Barth 2016), that it is typically most beneficial to only run 1 or 2 hardware threads per KNL core. In fact, for this code, running only 1 hardware thread per KNL core is the best choice in nearly all cases. This establishes that the optimal results for this test case of the CCI algorithm is the first result row of results in Table 1; we will use these results to compare to performance on other hardware.

Table 2 shows the direct comparison of run times of the best results of the KNL simulations in Table 1 with corresponding studies of a node with two multi-core Intel Skylake CPUs. Specifically, the KNL results are those running 1 hardware thread per KNL core from the first result row of Table 1. One node with two 24-core Intel Skylake CPUs on Stampede2 allows for a maximum of 48 threads. The second result row in Table 2 shows the timings using $t_{total} = 32$ total threads, with t_p chosen such that $t_{total} = t_p\, p = 32$; we stay with the power of 2 for consistency in this row, although 48 cores are available. The third result row in Table 2 shows the timings using all available $t_{total} = 48$ total threads, with t_p chosen such that $t_{total} = t_p\, p = 48$; in the $p = 32$ column for this row, we actually record the result for $p = 48$ and $t_p = 1$. The "N/A" indicate that the number of cores is not available for the hardware. Utilizing all 48 cores, that is,

Table 1: Observed wall clock time in seconds for reconstruction at 600 iterations of the CCI algorithm using $h_{total} = 200$ shifted histograms, accessed by various combinations of p processes and h_p histograms per process. The number t_p of OpenMP threads per MPI process is chosen such that the total number of threads is the specified t_{total}.

MPI processes $p =$	1	2	4	8	16	32	64
Histograms per MPI process $h_p =$	200	100	50	25	13	6	3
KNL with 1 hardware thread per core: $t_{total} = 68$	156	78	39	21	13	9	8
KNL with 2 hardware threads per core: $t_{total} = 136$	173	84	41	34	20	67	8
KNL with 4 hardware threads per core: $t_{total} = 272$	181	103	57	31	19	134	11

$t_{total} = 48$, produces faster results for smaller numbers of MPI processes p, but as p increases, performance improvements are negligible. Notice that on both the KNL and Skylake, as MPI processes p increase, so does performance. In fact, on the Skylake node, performance profits most from larger numbers of MPI processes p to the point of OpenMP single-threading ($t_p = 1$ per MPI process). Notice again that this is not just the effect of parallel performance, but also due to the algorithmic design of distributing the h_{total} shifted histograms to the p MPI processes. While the KNL's best performance is able to complete iterating in 8 seconds, the Skylake is able to achieve this in 3 seconds. In every case, the Skylake significantly outperforms the KNL no matter the combination of p and t_p are chosen.

Table 2: Observed wall clock time in seconds for reconstruction at 600 iterations of the CCI algorithm using $h_{total} = 200$ shifted histograms, accessed by various combinations of p processes and h_p histograms per process. The number t_p of OpenMP threads per MPI process is chosen such that the total number of threads is the specified t_{total}, which is the optimal value for each hardware platform.

MPI processes $p =$	1	2	4	8	16	32	64
Histograms per process $h_p =$	200	100	50	25	13	6	3
KNL with 1 hardware thread per core: $t_{total} = 68$	156	78	39	21	13	9	8
Two 24-core Intel Skylake: $t_{total} = 32$	62	19	11	7	5	3	N/A
Two 24-core Intel Skylake: $t_{total} = 48$	51	16	9	5	4	3	N/A

5 CONCLUSIONS

First, the application results confirm that the parallelized algorithm takes advantage of the problem structure correctly in the way that it distributes the total number of shifted histograms to the parallel processes. Second, for these simulations with equivalent application results, we show the power of the new hybrid MPI+OpenMP parallelization of the code to potentially dramatically reduce run times on both the Intel Xeon Phi KNL and new Intel Skylake CPUs. In the present results, it turned out that this code on the Skylake hardware did not profit from multi-threading, but this feature will likely be necessary if using multiple nodes of a distributed-memory cluster. Third, the performance results with the new Intel Skylake CPUs are sufficiently fast to move the image reconstruction close to the timeframe suitable for the clinical environment of a patient having to lie still for this amount of time, even with some additional time for setup, output, and post-processing. This result is far more achievable with the Skylake, it seems, as the additional time for setup and post-processing of 40 seconds on the KNL is significantly larger than the 10 seconds on the Skylakes.

In summary, while we see that the many-core KNL allows for impressive scaling, a dual-socket node with the most modern Intel Skylake CPUs yet beats the KNL, provided MPI parallelism is used in the code. This is both due to the core count being nearly as high as on the KNL and the higher clock rate of each core. The KNL's advantages should be the ability to run several threads per KNL core and the faster on-chip memory (as opposed to Skylake accessing main memory on the node), but they are not sufficient to beat the Skylakes. There also seems to be less efficient connection to the node's main memory, given the much larger additional time for setup and post-processing on the KNL. TACC has just released the Intel Skylake CPUs with 24 cores in December 2017, and we hope this paper to be the first to contrast the performance of both processor types, many-core KNL and multi-core Skylake, available on Stampede2 for the near future.

ACKNOWLEDGMENTS

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI–1053575 (Towns et al. 2014). We acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing the HPC resources Stampede2. The hardware in the UMBC High Performance Computing Facility (HPCF) is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS–0821258, CNS–1228778, and OAC–1726023) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from the University of Maryland, Baltimore County (UMBC). See hpcf.umbc.edu for more information on HPCF and the projects using its resources. Co-author James Della-Giustina was supported in part, by the Math Computer Inspired Scholars program, through funding from the National Science Foundation and also the Constellation STEM Scholars Program, funded by Constellation Energy. Graduate assistant and co-author Carlos Barajas was supported by UMBC as HPCF RA. For co-author Matthias Gobbert, this material is based upon work supported while serving at the National Science Foundation. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. The research reported in this publication was supported by the National Institutes of Health National Cancer Institute under award number R01CA187416. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health.

REFERENCES

Andreyev, A., A. Sitek, and A. Celler. 2011. "Fast image reconstruction for Compton camera using stochastic origin ensemble approach". *Med. Phys.* vol. 38 (1), pp. 429–438.

Avila-Soto, F. X., A. N. Beri, E. Valenzuela, A. Wudenhe, A. R. Blenkhorn, J. S. Graf, S. Khuvis, M. K. Gobbert, and J. Polf. 2015. "Parallelization for Fast Image Reconstruction using the Stochastic Origin Ensemble Method for Proton Beam Therapy". Technical Report HPCF–2015–27, UMBC High Performance Computing Facility, University of Maryland, Baltimore County.

Mackin, D., S. Peterson, S. Beddar, and J. Polf. 2012. "Evaluation of a stochastic reconstruction algorithm for use in Compton camera imaging and beam range verification from secondary gamma emission during proton therapy". *Phys. Med. Biol.* vol. 57 (11), pp. 3537–3553.

Polf, J. C., and K. Parodi. 2015. "Imaging particle beams for cancer treatment". *Physics Today* vol. 68 (10), pp. 28–33.

Rosales, C., D. James, A. Gómez-Iglesias, J. Cazes, L. Huang, H. Liu, S. Liu, and W. Barth. 2016. "KNL Utilization Guidelines". Technical Report TR–16–03, Texas Advanced Computing Center, The University of Texas at Austin.

Sodani, A., R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu. 2016. "Knights Landing: Second-Generation Intel Xeon Phi Product". *IEEE Micro* vol. 32 (2), pp. 34–46.

Towns, J., T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R. Scott, and N. Wilkins-Diehr. 2014. "XSEDE: Accelerating Scientific Discovery". *Comput. Sci. Eng.* vol. 16 (5), pp. 62–74.

Wilson, R. R. 1946. "Radiological Use of Fast Protons". *Radiology* vol. 47 (5), pp. 487–491.

AUTHOR BIOGRAPHIES

JAMES DELLA-GIUSTINA is an undergraduate student pursuing a B.S. in Computer Science, currently at the Community College of Baltimore County. His interests lie in parallel computing, embedded systems, and cryptography. His email address is jdella@umbc.edu.

CARLOS BARAJAS is a graduate student pursuing his Ph.D. in Applied Mathematics at the University of Maryland, Baltimore County. He received his B.A. in Computer Science and Pure Mathematics from Olivet College. His email address is barajasc@umbc.edu.

MATTHIAS K. GOBBERT received his Ph.D. from Arizona State University. He is a Professor in the Department of Mathematics and Statistics at the University of Maryland, Baltimore County and currently serves as program director in the Division of Mathematical Sciences at the National Science Foundation. His email address is gobbert@umbc.edu.

DENNIS S. MACKIN is currently Assistant Professor at the University of Texas, MD Anderson Cancer Center. He holds a Ph.D. in High Energy Particle Physics from Rice University. His email address is dsmackin@mdanderson.org.

JERIMY POLF is Associate Professor at the Marlene and Stewart Greenebaum Cancer Center at the University of Maryland, Baltimore. He holds a Ph.D. in Photonics from Oklahoma State University and is board certified in Medical Physics by the American Board of Radiology. His email address is jpolf@umm.edu.

SOFTWARE ARCHITECTURE FOR PARALLEL PARTICLE TRACKING WITH THE DISTRIBUTION OF LARGE AMOUNT OF DATA.

Florent Bonnier

Li-Parad, MDLS,
Versailles/St-Quentin University
ONERA
Department of Information Treatment and Systems
8 chemin de la Hunière et des Joncherettes
Palaiseau, Essonne, France
florent.bonnier@onera.fr

Nahid Emad

Li-Parad, MDLS,
Versailles/St-Quentin University
45 avenue des Etats-Unis
78035 Versailles cedex
nahid.emad@uvsq.fr

Xavier Juvigny

ONERA
Department of Information Treatment and Systems
8 chemin de la Hunière et des Joncherettes
Palaiseau, Essonne, France
xavier.juvigny@onera.fr

ABSTRACT

This study is about the architecture of a library designed for particle tracking. This library exposes common features used to track particles in large meshes using parallel algorithms to localize, manage, distribute and move particles over computing units. The proposed library design ables particles to be tracked using multiple heterogeneous parallel paradigms with component reusability. A customized algorithm to distribute particles over processes has been developed that uses different features of this architecture and shows a high impact on particle localization and movement execution time.

Keywords: particles, algorithms, architecture, load-balancing.

1 INTRODUCTION

Numerical simulations are often used for particle tracking with Lagrangian approach in domains such as following sand transport in a fluid (Kang and Guo 2006, Soulsby, Mead, Wild, and Wood 2010), solving gas-liquid interactions (Darmana, Deen, and Kuipers 2006, Schäferlein, Keßler, and Krämer 2017) or simulating spray of diesel drops (Kärrholm 2008).
The Lagrangian approach consists in computing the next particle position with :

$$\frac{dx_i}{dt} = u_i(t) \tag{1}$$

with x_i is the coordinate of the i^{th} particle, t is time step and u_i is the velocity vector of the i^{th} particle (Riber, Moureau, García, Poinsot, and Simonin 2009).
Particle tracking can be very easily parallelized as it is an embarrassingly parallel workload on shared memory architectures. The particles can be easily distributed on computing units such as GPUs (Nishimura 2011, Fitzek 2015) or Xeon-Phi and other many-cores architectures. The workload is well balanced.
Many numerical simulations come with velocity fields associated to a mesh. These fields are distributed on computing cores using the geometry of the mesh to lighten the workload on all available cores and memory. The flowfield can change between two time steps of the simulation.
On distributed architectures, mesh and particles are partitionned and distributed on different memory units. The mesh and particles are not distributed with the same method to ensure an acceptable load-balancing, meshes are partitionned using graph techniques preconditioners or mapping algorithms(Schloegel, Karypis, and Kumar 2000) and particles are managed with distribution strategies (Fonlupt, Marquet, and Dekeyser 1998). Here lies the main difficulty of load-balancing because particles need their environnement, the part of the mesh where they are, to be followed. As a particle can be located on a different memory unit than its environment, a smart strategy is often required for an efficient parallel performance. Two common strategies exist: the first consists in partitioning the computing mesh and flowfield in a well balanced way and distribute it on remote memories. Particles are initially attached to their environment. Depending on initial conditions, this can lead to an unbalanced distribution. It means that particles work with local data (from the local mesh) but some cores might have no particles to track (Darmana, Deen, and Kuipers 2006). In the worst case, this approach leads to a sequential computation if all particles are on the same partition of the mesh. The second approach consists in distributing particles over the different processes but their computation need the complete part of the mesh (Darmana, Deen, and Kuipers 2006). So all computing cores have a copy of the computing mesh which is very greedy in terms of memory occupancy. Both approaches have important consequences in terms of computing efficiency and performances: the first approach can lead to a sequential computation with important messages when particles leave a memory space to another one and the second approach particles move in a copy of the mesh which removes communications during particles movements but give a limitation for the number of particles or mesh sizes the memory spaces can store in the same time. The approach used is almost the same as the second approach but instead of giving all processes a copy of the entire mesh, this mesh is partitionned and smaller pieces of the partitionned mesh are import/export from/to other processes.
In order to develop particle tracking functions used in multiple environments, simulation cases and scientist teams, the different objects created to compute particle distribution are gathered in a high performance library. This library will be hardly used in different way and with a large panel of data types, structures and objects so that it has to be modular, highly extensible and upgradable.

In this study, we propose a parallel library to localize, track and distribute particles on heterogeneous high performance machines. We propose a design approach for this library in order to prepare the particle tracking field for future Exascale machines providing modularity and maintainability. We also give a customized algorithm to distribute particles on remote memory spaces. This algorithm performs some cache optimizations with spatial and temporal data locality.

2 LIBRARY ARCHITECTURE

Many software designs have been implemented to solve a large number of problems in many fields and a lot of software architectures and design patterns exists (Gorton 2006). Particle tracking is based on two main operations: localization and movement computation. Figure 1 is a sketch of operations used in particle tracking. This figure shows two things: particle tracking is an embarassingly parallel problem and as particles are independants, they call independants operations and datasets. We can sketch processors or computing units with the same graph as the different colums can stand for a group of particles or a group of processes. Dashed arrows between processes represent the communications as a process may need a dataset, a part of

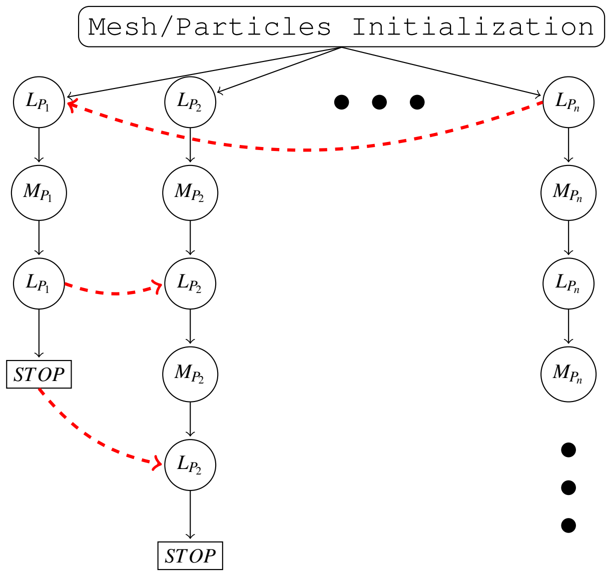


Figure 1: Particle tracking task graph. P_n stand for particle$_n$, a group of particles or a group of processes, L stands for Localization operation and M for Particle Movement.

the mesh to track its particles. In this example the first imports data owned by the n^{th} process to localise the particle during the first localization step and exports data to help the second process to localize its particle during the second step. This figure gives an good overview of the main difficulty to express high parallelism: a well balanced workload. As processes would work during different duration depending on particle's lives, a process that have lost all work (first process in figure 1), will wait other processes to export data to them. This is an opportunity to give it some particle to track. As the particle localization is the operation in which a process need to import data, this step is implicitly a synchronization step. According to these comments, we can extract components from the task graph. There are three main components showed in Figure 2. The components are the data component which represents all data structures used in the library as the mesh, the flowfield and particles, the computation component that gathers computation functions as localization computation, particle moving, and the communication components that take care of communication functions to import and export data to remote memory spaces. This pattern design, known as Component-based

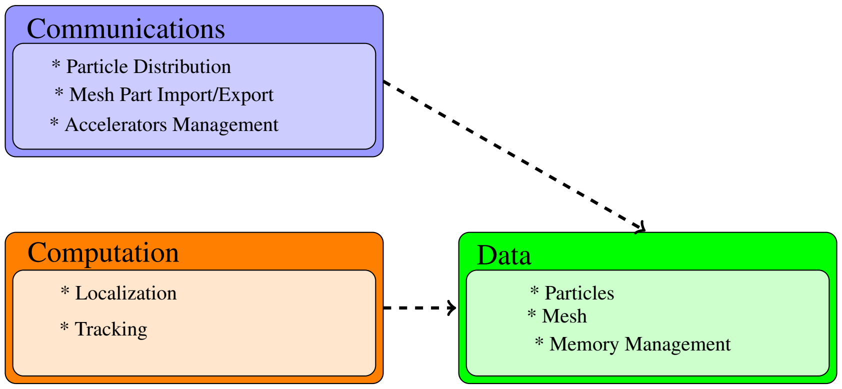


Figure 2: Component Graph of the library.

Architecture, consists in decomposing software services into components that are independants, reusable, extensible, replaceable and upgradable. These independant compenents are used to give services for example in our case the computation of a ray/plane intersection, the localisation of a vertex in a polyhedron

or the communication of a CPU and the GPU accelerator on the same node. We tried to render the three components as independant as possible by using primitive data types. Homemade classes and structures are only used inside a single component causing each component to be encapsulated.
According to figure 2, components depend on the Data component. Indeed, the Communication component import and export data from the Data component and functions of the Computation component operate on the same data flow.

3 DOMAIN DISCRETIZATION FOR PARTICLE LOCALIZATION AND DATA EXPORT.

Localize particles in mesh parts is a complex operation that includes communications, load-balancing and memory management. This operation can be done using multiple methods including containment problems, graph searches, repartitioning and particle distribution. Two standard approaches are gathering mesh parts and copying them on all memory devices, or make the particles move through memory devices. The first approach involves a lot of memory that is used to store all the computing mesh on all memory devices and the second one involve that particle workload is not well balanced over the cores. To localize particles we choose to partition the computing mesh using a Hilbert curve method (Castro, Georgiopoulos, Demara, and Gonzalez 2005) and create an overlapping structured grid with the minimum and maximum coordinates of the computing mesh. Figure 3 shows the structured grid overlapping the computing mesh as a bounding

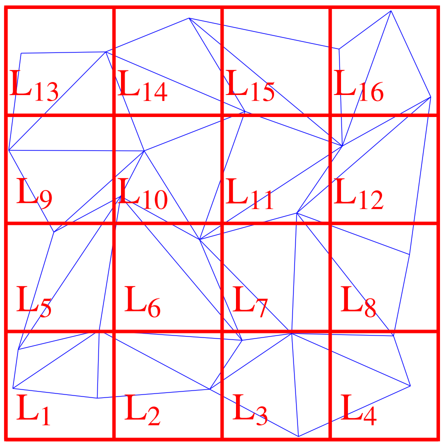


Figure 3: Structured Grid overlapping a computing mesh.

box. This structured grid is used for particle localization and mesh data export. For these both usage, the data structure is built once and for all during the initialization phase, that way initialization cost does not appears during particle movement. In the other hand, the memory space dedicated to the structured grid is a constant cost as long as the data structure is allocated. The memory usage can be a problem as mesh data that is going to be exported is equivalent to a copy of the local mesh partition.

3.1 Particle Localization.

The main hotspot of the particle tracking problem is the particle localization. In a meshed context, this problem is known as containment problem, which consists in determining if a polyhedra contains a point or a particle. This containment problem is geometrically complex to solve.
To determine the cell where a particle is, a set of cells is selected, then a containment check is computed with these candidate cells. The most important parameter is the number of candidate cells, the more this number is reduced, the faster the particle localization is. To reduce the number of candidate cells, we implement

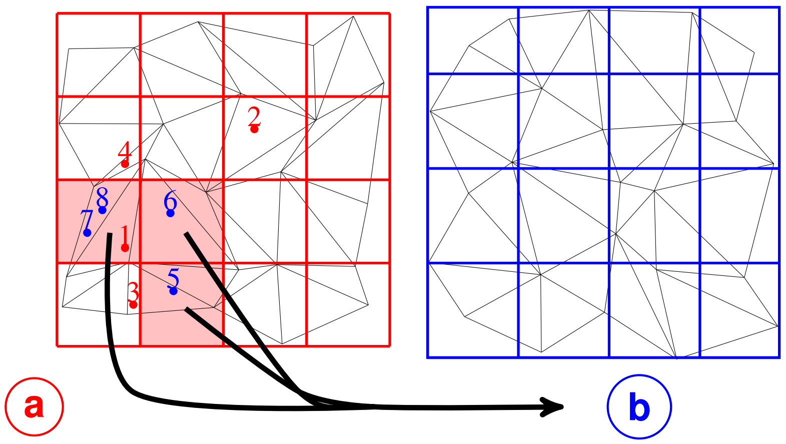


Figure 4: Example of structured cells import and export to localize and track removed particles. ⓐ and ⓑ are processes that owns a different mesh partition and all particles are localized in ⓐ. Particles 1, 2, 3 and 4 are tracked by ⓐ whereas 5, 6, 7 and 8 are tracked by process ⓑ.

two independant methods that reduce this number in two steps: a overlapping structured grid can localize a particle in a coarse grain step then the minimum distance vertex is applied to localize the particle in a neighbouring cell. This coarse grain localization step can be achieved by another method than a structured grid like octrees or kd-trees with a similar memory impact. We have choosen the structured grid because the particle localization is done with a division per dimension, the data structure is quiet simple to implemment and the initialization cost is very low.

As nowadays a large computing mesh is often partitionned and distributed on remote memory spaces, a structured grid is build on all these partitions. That implies that to localize a particle on a remote partition the corresponding structured grid have to be import. We choose to copy the dimensions of all known structured grids in the parallel context because of the low memory space needed to describe the structured grids. In fact, the memory usage is equivalent to 2 coordinates per dimension (6 floats in 3D) and 1 integer per dimension to desccribe the discretization in each direction.

3.2 Mesh parts export.

The structured grid is also useful in particle distribution and to track particles on remote memory devices. The overlapping grid is composed of cells that contain multiple cells of the original computing mesh. These cells of the structured grid (L_{1-16} in figure 3) can be considered as independent mesh parts and can be send or received to or from another process (a remote CPU socket or accelerator).

Figure 4 shows an example of the structured grid cells in mesh data communication phase. ⓐ and ⓑ represent two computing units such as remote CPUs or accelerators. The two units own a piece of the global mesh as well as the corresponding structured grid. In this example, all particules are in the red mesh (owned by ⓐ). In this worst case particles are distributed on the two processes: each process has 4 particles. Process ⓑ (blue), commands process ⓐ to send pieces of the mesh where the particles $5, 6, 7$ and 8 are. The 3 cells of the structured grid ⓐ are sent to process ⓑ which can track its particles until they leave their imported environment.
When a particle of ⓑ comes out of an imported cell, process ⓑ localize the particle in the structured grid of ⓐ and import the new environment.
This approach has the advantage to export and store mesh pieces and not have a copy of the whole computing mesh on each memory device. The worst case happens when a process need to import all pieces of an other

process. This happens when particles are randomly distributed and this can be solved by ordering particles or distribute particles taking into account their localization in memory.

4 ALGORITHM FOR PARTICLE DISTRIBUTION.

As sketches the tasks graph, the particle tracking problem is an embarassingly parallel problem. This parallelism is brought with the task distribution which corresponds to the particle distribution over computing units. The more particle are well distributed, the higher the parallelism rate is. To distribute particles over remote memory spaces, we use and customize the partner processor algorithm described by O'Brien, Brantley, and Joy (2012). This algorithm consists in iteratively assigning to each process a partner process. The

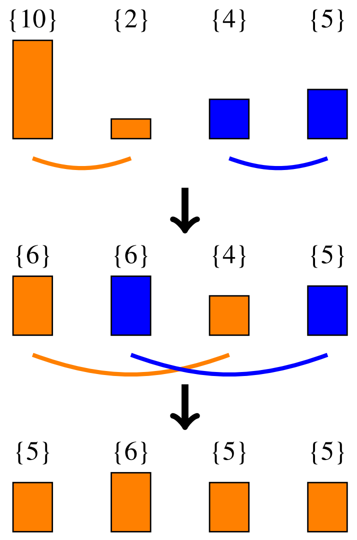


Figure 5: Particle Distribution on 4 processes.

two partners communicate and balance their particles. The maximum number of partners of a process is up to $log(N)$ where N is the total number of processes.
This algorithm gives well balanced processes in a few number of iterations but moves randomly particles through remote memory spaces. Indeed particles that moves on remote processes are not ensured to find on the new memory space all needed data.

Figure 5 is a sketch of the distribution algorithm on an example with 21 particles on 4 processes. The algorithm runs on 2 iterations. Colors represent the different associated of processes during iterations.
We modified the algorithm in order to take into account the geometrical locality of the particles. In that way, particles are rearranged to stay on processes that own their environment in order to optimize spatial and temporal locality.

Figure 6 gives the same example of figure 5 but with our algorithm modification. In this example, 21 particles are tracked on 4 processes. Particles are localized as follows: 3 particles are located in the mesh on the first process (orange), 5 particles are on the second process (green), 11 particles belong to the third process(blue) and 2 particles belong to the fourth process (red). There are two main problems in this example : most of the particles are located on the third process and most of them are stored on the first process. The idea of the algorithm is to keep the maximum number of particles on the process where these particles can be localized. In figure 6, the first iteration makes the process couples $\{1,2\}$ and $\{3,4\}$. The ideal workload of the first couple is 6 particles per process, so each process will send to its partner the maximum number of particles that are localized on this partner. At the same time, each process send the difference between

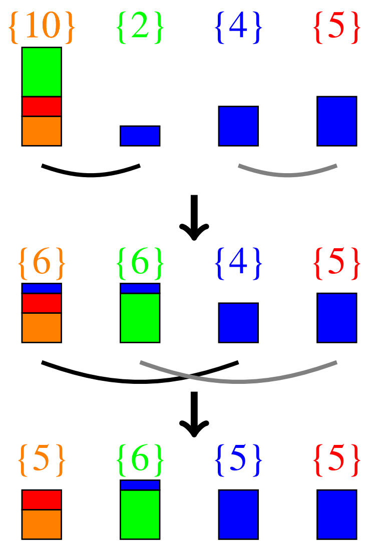


Figure 6: Particle Distribution on 4 processes with the customized algorithm.

the ideal number of particles and the maximum number of recieved particles. To illustrate this operation, the first process sends 5 particles to the second process and the second process sends 1 remaining particle to process 1. At the end of the run, particles are well balanced and an effort have been done to send particles that are not on their mesh partition. It can be noticed that this algorithm can only work with a number of processes equal to a power of 2 as the number of partner (and iterations) of a process is given by:

$$\lceil log_2(nProcs) \rceil \tag{2}$$

This maximum iteration allows to be very performant on machines with very high number of cores as expectited in an exascale future. It means that the number of communications and messages during the distribution step is limited.

5 EVALUATION OF THE PARTICLE DISTRIBUTION ALGORITHM.

The performances of the distribution algorithm are measured with the execution time of a parallel particle tracking simulation. Particle coordinates and their direction vectors are randomly initialized on process of rank 0, which means that all other processes have no particles to track at the beginning of the simulation. The mesh represent a three-dimensional cube discretized with cubic cells. When a particle comes out of the computing mesh, it bounces on the cube walls. The cube is 99 vertices length so 98 cubic cells length. To move the particles, we choose to set a number of time steps represented by the particle/face intersection computation. Each time step, the particle move to the intersection point between the particle and the face of the cell it is going to cross. This computation allows to know the next cell of the computing mesh the particle will cross. The number of time step is set to 100. Performances are measured on the same problem size which corresponds to the track of 64 millions particles and on different number of processes. The more processes there are, the more particles and mesh are distributed on cores. To perform communications we used Message Passing Interface(MPI) on all cores initializing one MPI thread per physical thread. For this evaluation, we do not use different sizes of the structured grid cells, we only use unit bounding boxes, in other words, structured grids are made of a unique cubic cell per local mesh partition.

Figure 7 shows the execution time in seconds of the distribution operation, localization and particle movement. This figure shows the that the localization and particle movement is highly impacted by the distribution algorithm. The customized distribution algorithm that takes into account the particle localization in

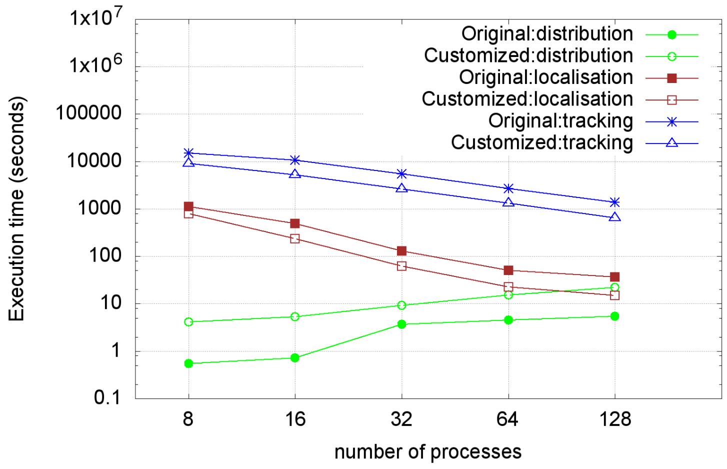


Figure 7: Execution Time of the two distribution algorithms in Localization and track computation.

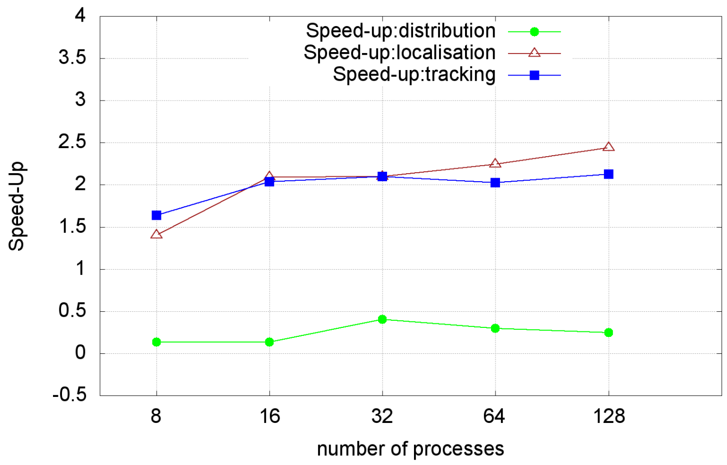


Figure 8: Speed-up.

the different structured grids always gives a speed-up during the localization and the movement steps. The algorithm we developed has an overhead compared to the original distribution algorithm because of the localization understep to localize particles in different structured grids. The next figure 8 shows the speed-up the algorithm coustomization brought comparing with the original distribution algorithm. The speed-up is computed with:

$$Speed - up = \frac{T_{original}}{T_{customized}} \tag{3}$$

This figure 8 shows that our algorithm make the particle movement two times more efficient and the particle localization up to four times more efficient than if particles are randomly distributed. The developed algorithm has a higher impact on particle localization because during this step, particles are localized in different structured grids and for each particle, if it is localized in the local mesh, this will limit communications between processes and overhead due to irregular cache access. During particle movement, there is no communications between computing units, mesh data is already on local memory and particles are already localized. The obtained speed-up is due to cache reusage between two particles that have more chance to need neigbouring data or the same data to compute the next position. More generally, the speed-up obtained during the particle movement is due to spatial and temporal locality of data.

6 CONCLUSION

The library architecture is really simple, based on three simple components, the different components used to localize, distribute and track particle work independently and are easily upgradable. The components can be used with multiple data types (thanks to $C++templates$) and implement parallel algorithms for heterogeneous machines to distribute particles and optimize spatial and temporal locality of data. The distribution algorithm helps to reduce mesh parts copies to be scattered, prefering to send foreign particles and keep neighbouring particles close to the core and the cache. This algorithm has multiple advantages, reduce message sizes because it allows two partner processes to communicate each iteration so that a core can not receive a huge number of messages from different partner during the same iteration. Unfortunatly the algorithm has an important drawback: it only works with a number of processes equal to a power of 2. This drawback contrasts with the idea of a complete library running on heterogenenous machines. Other algorithms to control work-balance will be study to find a strategy that mixes heterogeneousity and parallel efficiency. On the other hand, the distribution algorithm has been optimized to take into account particles locality. This optimization have demonstrated a real impact on computing performances during localization step and particle movement computation. Parallel work is done on the impact of the structured grid discretization and the minimum size of a structured cell that can be sent to a process to be efficient in terms of message size and temporal locality during movement commputation.

REFERENCES

Castro, J., M. Georgiopoulos, R. Demara, and A. Gonzalez. 2005. “Data-partitioning using the hilbert space filling curves: Effect on the speed of convergence of Fuzzy ARTMAP for large database problems”. *Neural Networks* vol. 18 (7), pp. 967–984.

Darmana, D., N. G. Deen, and J. Kuipers. 2006. “Parallelization of an Euler–Lagrange model using mixed domain decomposition and a mirror domain technique: Application to dispersed gas–liquid two-phase flow”. *Journal of Computational Physics* vol. 220 (1), pp. 216–248.

Fitzek, J. 2015. “Parallelization of the Particle-in-cell-Code PATRIC with GPU-Programming”. *PARS-Mitteilungen: Vol. 32, Nr. 1.*

Fonlupt, C., P. Marquet, and J.-L. Dekeyser. 1998. “Data-parallel load balancing strategies”. *Parallel Computing* vol. 24 (11), pp. 1665–1684.

Gorton, I. 2006. *Essential software architecture*. Springer Science & Business Media.

Kang, L., and L. Guo. 2006. “Eulerian–Lagrangian simulation of aeolian sand transport”. *Powder technology* vol. 162 (2), pp. 111–120.

Kärrholm, F. P. 2008. *Numerical modelling of diesel spray injection, turbulence interaction and combustion*. Chalmers University of Technology Gothenburg.

Nishimura, H. 2011. “GPU Computing For Particle Tracking”. *Lawrence Berkeley National Laboratory*.

O’Brien, M. J., P. S. Brantley, and K. I. Joy. 2012. “Scalable load balancing for massively parallel distributed Monte Carlo particle transport”. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

Riber, E., V. Moureau, M. García, T. Poinsot, and O. Simonin. 2009. “Evaluation of numerical strategies for large eddy simulation of particulate two-phase recirculating flows”. *Journal of Computational Physics* vol. 228 (2), pp. 539–564.

Schäferlein, U., M. Keßler, and E. Krämer. 2017. “Aeroelastic Simulation of the Tail Shake Phenomenon”.

Schloegel, K., G. Karypis, and V. Kumar. 2000. *Graph partitioning for high performance scientific simulations*. Army High Performance Computing Research Center.

Soulsby, R., C. Mead, B. Wild, and M. Wood. 2010. “Lagrangian model for simulating the dispersal of sand-sized particles in coastal waters”. *Journal of Waterway, Port, Coastal, and Ocean Engineering* vol. 137 (3), pp. 123–131.

AUTHOR BIOGRAPHIES

FLORENT BONNIER is currently a PhD student at the ONERA(French Aerospace Lab), the University of Versailles and affiliated to Maison de la Simulation. His PhD subject is "Parallel algorithms for particle tracking" that includes software engineering, geometry problems, high performance computing, work distribution and memory management. He holds a BS in computer science and a master of science in HPC and Simulation. His email address is florent.bonnier@onera.fr

NAHID EMAD received the Habilitation to Direct Researches (HDR) in Computer Science from the University of Versailles in 2001, the PhD and MS in Applied Mathematics from Pierre et Marie Currie University (Paris VI) in 1989 and 1983 and BS in Pure Mathematics from Arak University (Iran). She is Professor and affiliated in Maison de la Simulation and Li-Parad laboratory where she leads the Intensive Numerical Computation group in Li-Parad. She is the advisor of 18 PhDs and authored more than 150 papers in international journals and conferences. Her main researches concern programming models for extreme scale computing, parallel and distributed computing, software engineering for parallel and distributed computing, numerical algorithms, linear algebra. Her email address is nahid.emad@uvsq.fr

XAVIER JUVIGNY is a research engineer at ONERA aerospace laboratory. He holds a PhD in Applied Mathematicals from Paris VI University. His research interests lie in linear algebra algorithms, high performance computation and distributed algorithms. His email address is juvigny@onera.fr.

COMPARISONS OF CORE AND UNCORE FREQUENCY SCALING MODES IN QUANTUM CHEMISTRY APPLICATION GAMESS

Vaibhav Sundriyal and Masha Sosonkina

Department of Modeling, Simulation and Visualization Engineering, Old Dominion University, Norfolk, VA, USA.

Bryce M. Westheimer and Mark Gordon

Department of Chemistry, Iowa State University, Ames, Iowa, USA.

ABSTRACT

Energy efficiency and energy-proportional computing have become a central focus in modern supercomputers. With the exascale computing throughput purported to be bound by the 20 MW power wall, there is an urgent need for power efficiency in modern computing systems. Apart from processor cores and DRAM, the other chip components (typically collectively denoted as *uncore*) become increasingly important contributors to the total system power. In this paper, the uncore frequency scaling (UFS) is explored with respect to its effect on latencies and bandwidths. Next, UFS and core dynamic voltage and frequency scaling (DVFS) are compared as to their energy-saving potential through experiments on a 20-core Haswell-EP machine using the quantum chemistry application GAMESS. Results depict that UFS is comparable to DVFS in terms of power saving capability and when used in conjunction with DVFS, it can save energy up to 21% for GAMESS execution.

Keywords: Uncore Frequency Scaling (UFS), Dynamic Voltage and Frequency Scaling (DVFS), Power, GAMESS, Energy Savings.

1 INTRODUCTION

Power consumption has become a major concern for modern and future supercomputers. For the current topmost petascale computing platforms[1] in the world, it is typical to consume power on the order of several megawatts (15 megawatts for Sunway Taihulight in 2017), which at current prices may cost on the order of several million dollars. In the quest for the exascale performance, the power consumption growth rate must slow down and deliver more calculations per unit of power, giving rise to power-bounded computing in which components of a computing system operate under a fixed power budget such that performance is maximized.

To be able to operate a system under a given power budget, it is imperative that it is comprised of components which have the necessary capability to limit their power consumption. The current generation of Intel processors provides various P-states for dynamic voltage and frequency scaling (DVFS) and T-states for introducing processor idle cycles (throttling) to limit the power consumption of the processor. With the advent of the Sandy Bridge family of processors, Intel has introduced capabilities for both onboard power meters and power clamping. The Intel Running Average Power Limit (RAPL) (David, Gorbatov, Hanebutte,

[1]TOP500 list: http://top500.org/ .

Khannal, and Le 2010) provides a standard interface for measuring and limiting processor and memory power by HW, OS, applications etc.

Previous Intel processor generations used either a fixed uncore frequency or a common frequency for cores and uncore. The uncore describes the functions of a processor that are not handled by the core such as L3 cache, on-chip Interconnect etc. Starting from the Intel Haswell microarchitecture, the core and uncore frequency domains have been decoupled with the result that uncore frequency can be modified independently of the core frequency. The uncore frequency has a significant impact on on-die cache-line transfer rates as well as on memory bandwidth. By default it is set by hardware and can be specified via the MSR UNCORE_-RATIO_LIMIT (Lento 2014). The latest Intel CPUs work with at least two clock speed domains: one for the core (or even individual cores) and one for the uncore, which includes the L3 cache and memory controllers. Both are subject to automatic changes; in the case the of AVX code on Haswell and later CPUs the guaranteed baseline clock speed is lower than the standard speed rating of the chip.

GAMESS (Gordon and Schmidt 2005, Schmidt, Baldridge, Boatz, Elbert, Gordon, Jensen, Koseki, Matsunaga, Nguyen, Su, Windus, Dupuis, and Montgomery 1993), is a widely used package for *ab initio* molecular quantum chemistry to compute a variety of molecular properties, ranging from simple dipole moments to frequency dependent hyperpolarizabilities. This paper studies the energy savings in GAMESS using both UFS and DVFS; a power-bounded runtime strategy is proposed which maximizes the performance of GAMESS execution under a given power budget.

While DRAM frequency scaling has been studied in a previous work (Sundriyal and Sosonkina 2016), to the authors' knowledge, this is the first work that evaluates efficacy of uncore frequency scaling on a real-world application. In a nutshell, contributions of this work include

- Exploring the effects of Uncore Frequency Scaling (UFS) on component latency and bandwidth.
- Comparing UFS and DVFS as to their energy-saving potential on a real-world application.

The rest of the paper is organized as follows. Section 2 summarizes the electronic structure calculations existing in GAMESS and its communication model. Section 3 remarks on the UFS application and studies its effect on latency and bandwidth of certain uncore components. Section 4 discusses the different process mappings and the application of UFS and DVFS to GAMESS. Section 5 shows experimental results, while Section 6 and Section 7 provide related work and conclusions, respectively.

2 GAMESS OVERVIEW

GAMESS is one of the most representative freely available quantum chemistry applications used worldwide to do *ab initio* electronic structure calculations. A wide range of quantum chemistry computations may be accomplished using GAMESS, ranging from basic Hartree-Fock and Density Functional Theory computations to high-accuracy multi-reference and coupled-cluster computations.

The central task of quantum chemistry is to find an (approximate) solution of the Schrödinger equation for a given molecular system. An approximate (*uncorrelated*) solution is initially found using the Hartree-Fock (HF) method via an iterative *self-consistent field* (SCF) approach, and then improved by various *electron-correlated* methods, such as second-order Möller-Plesset perturbation theory (MP2). The SCF-HF and MP2 methods are implemented in two forms, namely *direct* and *conventional*, which differ in the handling of electron repulsion integrals (ERI, also known as *2-electron integrals*). Specifically, in the conventional mode all ERIs are calculated once at the beginning of the interactions and stored on disk for subsequent reuse whereas in the direct mode ERIs are recalculated for each iteration as necessary. The SCF-HF iterations and the subsequent MP2 correction find the energy of the molecular system, followed by evaluation of energy gradients.

Data Server Communication Model. The parallel model used in GAMESS was initially based on replicated-data message passing and later moved to MPI-1. Fletcher *et al.* (Fletcher, Schmidt, Bode, and Gordon 2000) developed the Distributed Data Interface (DDI) in 1999, which has been the parallel communication interface for GAMESS ever since. Later (Olson, Schmidt, Gordon, and Rendell 2003), DDI has been adapted to symmetric-multiprocessor (SMP) environments featuring shared memory communications within a node, and was generalized in (Fedorov, Olson, Kitaura, Gordon, and Koseki 2004) to form groups out of the available nodes and schedule tasks to these groups. In essence, DDI implements a PGAS programming model by employing a data-server concept.

Specifically, two processes are usually created in each PE (processing element) to which GAMESS is mapped, such that one process does the computational tasks while the other, called the *data server*, just stores and services requests for the data associated with the distributed arrays. Depending on the configuration, the communications between the compute and data server processes occur either via TCP/IP or MPI. A data server responds to data requests initiated by the corresponding compute process, for which it constantly waits. If this waiting is implemented with MPI, then the PE is polled continuously for the incoming message, thereby being always busy. Therefore, when the waiting is implemented in MPI, it is preferred that a compute process and data server do not share a PE to avoid significant performance degradation. When executing on an N-processor machine, each PE is assigned compute (C) and data server (D) process ranks as follows: $C_i \in [0, N-1]$ and $D_i \in [N, 2N-1]$, where $(i = 0 \ldots N-1)$. Thus, the data server D_i associated with the ith compute process C_i is $N+i$.

3 UNCORE FREQUENCY SCALING

In this paper, a compute node having two Intel Xeon E5-2630 v3 10 core processors with 32 GB (4 $\times$ 8GB) of DDR4 was used as the experimental testbed. The core and uncore frequency range from 1.2–2.3 GHz and 1.2–2.9 GHz, respectively. The wall power consumption was measured using the Wattsup meter.

3.1 How to Apply UFS

Each component of an Intel processor belongs either to the “core” or the “uncore” part of the chip. The core part comprises the individual computational cores and their private L1 and L2 caches. The uncore part comprises everything else, such as the shared last-level cache, memory controllers, and Intel Quickpath interconnect (QPI). Before the Intel Haswell microarchitecture, the core and uncore parts of the chip shared the same frequency domain. This meant setting the CPU cores to run at a certain clock frequency resulted in the uncore running at the same frequency.

Starting with the Haswell microarchitecture, processors feature separate frequency domains for the cores and the uncore which means they can operate at different clock speeds. Together with this change, Intel has introduced a feature named uncore frequency scaling (UFS), which allows the processor to dynamically change the uncore frequency based on the current workload. When UFS is disabled the uncore is clocked at its maximum clock frequency. As explained in (Hoffmann 2017), the uncore frequency can be adjusted using the lower 16 bits of MSR (model-specific register) with hardware address 0x620. The register contents represent upper and lower bounds for the uncore frequency: Bits 15 through 8 encode the minimum and bits 7 to 0 the maximum frequency. To derive a frequency from each of the two eight bit groups the integer encoded in the bits has to be multiplied by 100 MHz. To set a new uncore frequency, the value of the frequency is first divided by 100 MHz and then converted to hexadecimal and concatenated (to make the minimum and maximum frequency equal) to be written to MSR 0x620.

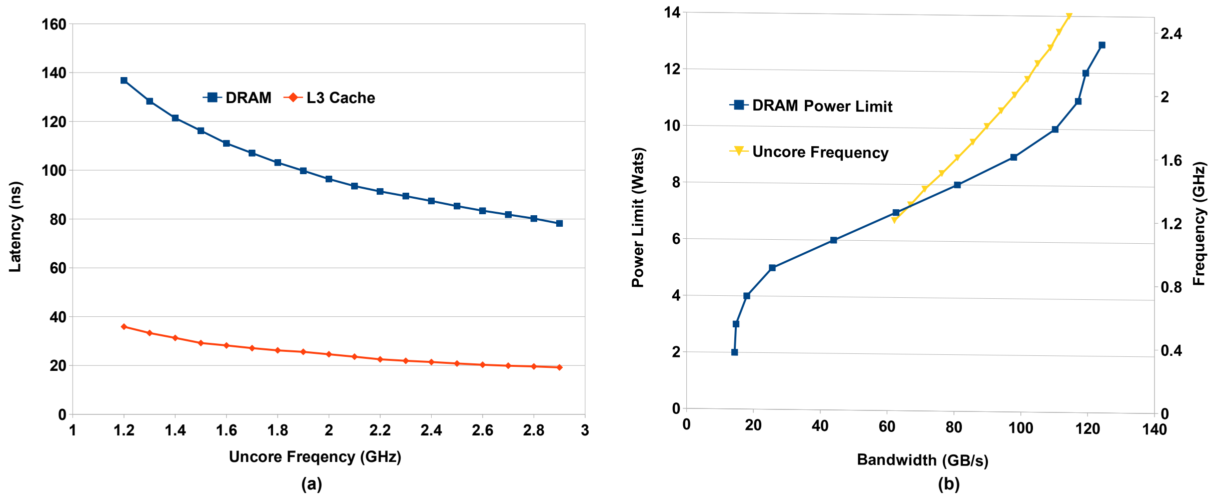


Figure 1: (a) DRAM and L3 cache latency change with variation in uncore frequency and (b) Variation in DRAM peak bandwidth with variation in uncore frequency and DRAM RAPL power limit.

3.2 Effect of UFS on Component Latency, Bandwidth, and Idle Power Consumption

Figure 1(a) depicts DRAM and L3 cache read latency measured through the *lat_mem_rd* utility in the lmbench (McVoy and Staelin 1996) suite, when the uncore frequency is modified from 1.2 GHz to 2.9 GHz in steps of 100 Hz, keeping the core frequency at its maximum value of 2.3 GHz. It can be observed that the latency of both components decreases uniformly with the increase in uncore frequency. The increase in latency between the maximum and minimum uncore frequency for L3 cache and DRAM from Fig. 1 was 102% (19.2 μs to 39.8 μs) and 96% (78.2 μs to 136.8 μs), respectively.

Figure 1(b) shows the change in peak memory bandwidth (Vishwanathan 2017) with the change in uncore frequency and DRAM power limit set through RAPL. The DRAM power limit was modified from 13 to 2 watts, decreasing a watt each step and it was observed that reducing the power limit reduces the memory bandwidth drastically to the extent that the bandwidth at 13 watts (124.3 GB/s), is roughly nine times the bandwidth at 2 watts (14.1 GB/s). On the other hand, the reduction in bandwidth at the highest uncore frequency is nearly double when compared to the bandwidth at lowest uncore frequency. This can be explained by the fact that RAPL limits the DRAM power consumption by essentially crippling its bandwidth as per a power-performance model (David, Gorbatov, Hanebutte, Khannal, and Le 2010). The uncore frequency scaling, on the other hand, reduces the operating frequency of the DRAM controller(s) and does not directly affect the bandwidth of the memory modules. Therefore, power limiting can degrade the performance of memory intensive applications much more then uncore frequency scaling.

From Fig. 2, the change in idle power consumption of a compute node measured through the Wattsup meter, with varying uncore (Fig. 2(a)) and core (Fig. 2(b)) frequencies can be observed. Regarding the methodology for measuring the power consumption, both the uncore and core frequencies were set to their respective maximum values with the power consumption being measured for 5 minutes. After every 5 minutes, the core and uncore frequencies were reduced by 100 MHz till they reach their respective minimum values. Moreover, while measuring variation of idle power with the uncore frequency, the core was operated at its maximum possible frequency and vice-versa. It can be observed that reducing the uncore frequency does provide a higher reduction in the idle power consumption compared to reduction of the core frequency. More specifically, the idle power consumption was reduced by 2.5% when the uncore frequency was changed from

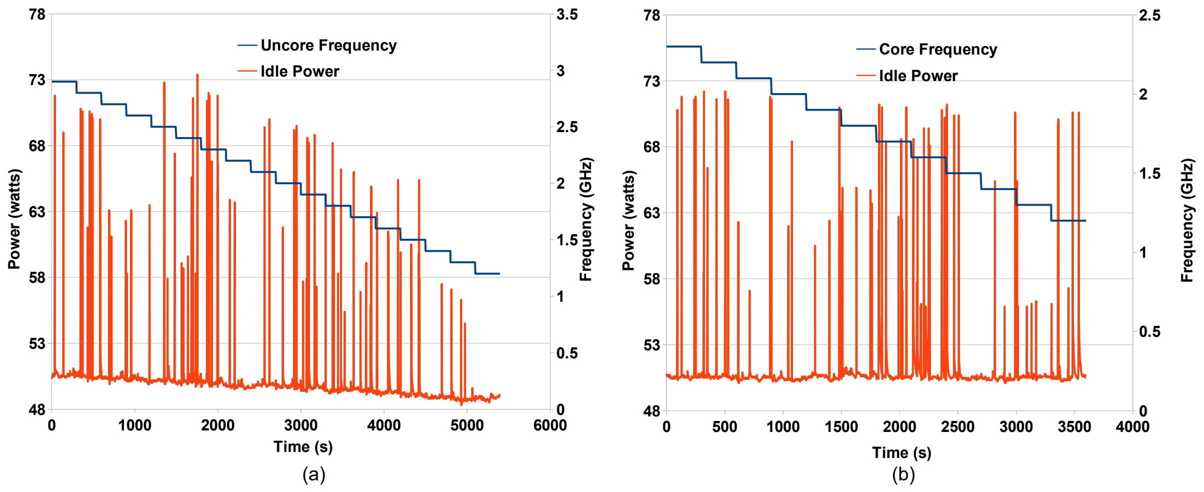


Figure 2: Change in node idle power consumption with variation in (a) uncore and (b) core frequency.

2.9 to 1.2 GHz. On the other hand, the reduction in the core frequency from 2.3 to 1.2 GHz reduced the idle power consumption only by 0.8%.

4 GAMESS PROCESS MAPPING

The data servers used in GAMESS are involved in global communications only and they do not perform any calculations, Therefore, reducing the frequency of the cores to which data servers are mapped would not affect the overall execution time by much. Keeping in mind the twin-core level DVFS granularity in the Intel Harpertown processors, process bindings were proposed in (Sundriyal, Sosonkina, Gaenko, and Zhang 2013) to effectively apply core frequency scaling during GAMESS execution.

The proposed process bindings were termed *Disjoint* and *Slave*. In Disjoint binding, the compute processes and data servers are mapped to different sockets and the data servers are operated at the lowest processor frequency whereas the compute processes were operated under an energy saving strategy. For the *Slave* binding, the compute processes and the data servers were mapped such that they were positioned alternatively on to the processors. Since DVFS was available only at twin-core level, the compute processes were operated under an energy saving strategy and the data servers operated at the same frequency the compute processes executed on, hence the term *Slave* binding.

Figure 3 depicts *Disjoint* (Fig. 3(a)) and *Slave* (Fig. 3(b)) in the context of the Haswell-EP processor used in this work with compute processes (C) and data servers (D) mapped to the 20 cores (10 cores per socket) with each core having its L1i, L1d and L2 cache. The default binding for GAMESS execution is *Disjoint* and by using the *MV2_CPU_MAPPING* parameter, *Slave* binding can be achieved.

In this work, the energy saving potential of UFS and DVFS is evaluated for GAMESS execution under *Disjoint* and *Slave*. The main difference between application of frequency scaling for *Slave* binding between (Sundriyal, Sosonkina, Gaenko, and Zhang 2013) and this work comes from the underlying granularity of DVFS available since the Haswell-EP processor used in this work supports per-core P state (PCPS) (Hackenberg, Schone, Ilsche, Molka, Schuchart, and Geyer 2015) along with the UFS capability.

Therefore, instead of the data servers simply executing at the frequency selected for the compute processes, they can be operated at the lowest processor frequency independently. The compute processes were operated under an energy saving strategy described in (Sundriyal and Sosonkina 2016). In addition to operating the

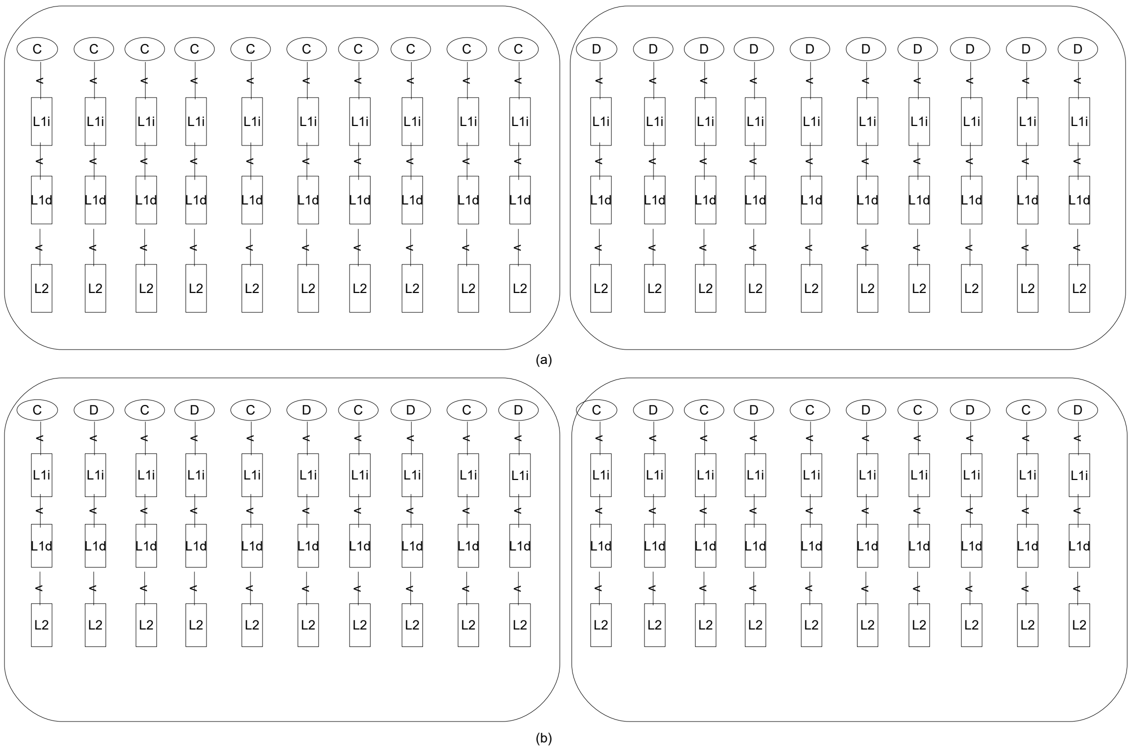


Figure 3: (a) Disjoint and (b) Slave binding in GAMESS.

data servers at the minimum core frequency, the data servers are aslo operated at the lowest uncore frequency. A total of 20 processes (ranks) were spawned for the GAMESS execution: ten data servers and ten compute processes. MVAPICH2 was used as the MPI library, which is tuned to take advantage of the shared memory whenever possible.

5 EXPERIMENTS

The GAMESS input was constructed to perform the third order Fragmental Molecular Orbital (FMO) (Fedorov and Kitaura 2004) calculation—in the conventional mode—for a cluster of 64 water molecules at the RHF/6-31G level of theory. As such, it involves calculations of fragment monomers, dimers, and trimers. The system is partitioned into 64 fragments so that each fragment is a unique water monomer. The input is referred to as h2o-64 in the rest of the paper.

5.1 Performance Degradation and Energy Savings with UFS and DVFS

Disjoint Binding. For each binding, the performance degradation and energy savings for the h2o-64 execution are evaluated for three configurations:

- `UFS Only`: Only uncore frequency scaling is applied to the data servers.
- `DVFS Only`: Only DVFS is applied to the compute processes and data servers.
- `UFS+DVFS`: UFS is applied to the data servers and DVFS is applied to both the compute processes and data servers.

Figure 4(a) depicts the 15 second power consumption trace for the `UFS Only`, `DVFS Only` and `UFS+DVFS` configurations operating under the *Disjoint* binding. The configurations `UFS Only`, `DVFS`

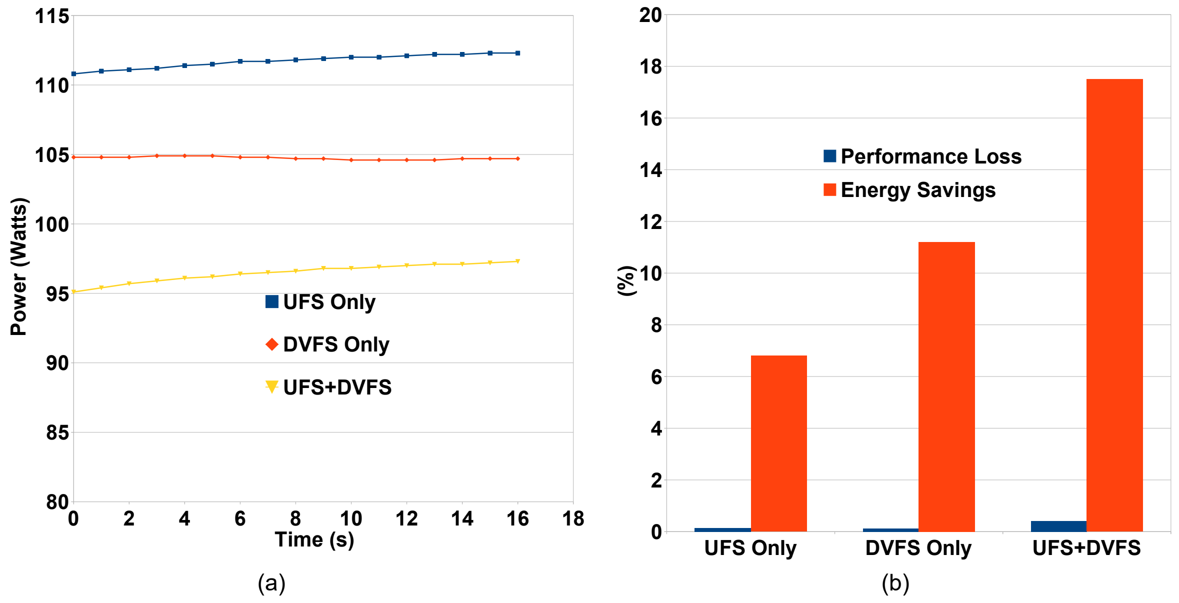


Figure 4: (a) Power consumption trace and (b) performance degradation and energy savings for UFS, DVFS and UFS+DVFS strategies under *Disjoint* binding.

`Only`, and `UFS+DVFS` reduce the node average power consumption by 7.4%, 12.6% and 19.1%, respectively. The energy saving strategy (Sundriyal and Sosonkina 2016) operating on the compute processes does not reduce the core frequency of the compute processes since they were determined to be extremely processor intensive. Noticing the performance degradation and energy savings (Fig. 4(b)) for the three configurations, it can be observed that all the three configurations result in miniscule performance degradations for the h2o-64 execution. In terms of energy savings, the highest energy savings were obtained for `UFS+DVFS` (17.5%) whereas `UFS Only` and `DVFS Only` ended up reducing the energy consumption by 7% and 11.2%, respectively.

The energy savings obtained through `UFS Only` can be explained through Table 1 which depicts average L2 cache misses and the DRAM bandwidth usage for the first 10 seconds of the h2o-64 execution for socket 0 (column `S0`), corresponding to the compute processes, and socket 1 (column `S1`), coresponding to the data servers. The L2 cache misses metric depicts the accesses made to L3 cache and main memory, i.e, the uncore memory components, which is why it was chosen to describe uncore utilization. It can be observed from Table 1 that both the L2 cache misses and DRAM usage are much higher for the compute processes as compared to the data servers. Therefore, the uncore infrastructure does not get utilized sufficiently for the data servers, and applying UFS to socket 1 does not degrade performance by much while substantially reducing uncore power consumption.

Slave Binding. For the *Slave* binding, UFS is applied to both the sockets for the *Slave* binding in the `UFS Only` configuration as data servers get scattered on both the sockets unlike the *Disjoint* binding. While this results in `UFS Only` reducing power consumption by a higher margin compared to `DVFS Only` (Fig. 5(a)), it also causes a relatively higher performance loss (8.1%) compared to `DVFS Only` (0.3%) as observed from Fig. 5(b). The reduction in power consumption—by 27.5%—is the highest for the `UFS+DVFS` configuration. The energy savings obtained for the three configurations `UFS Only`, `DVFS Only` and `UFS+DVFS` were 4.5%, 8.2%, and 21.1%, respectively. It can be summarized from the results that UFS is quite potent when it comes to reduction in power consumption compared to DVFS and in conjunction with DVFS, it can provide substantial energy savings. Also, both *Disjoint* and *Slave* bindings reduce

Table 1: Memory component utilization for the first 10 seconds of the h2o-64 execution on each socket.

	`S0`	`S1`
L2 Cache Misses (K/s)	2948	5.4
DRAM Read+Write (GB/s)	1.1	0.02

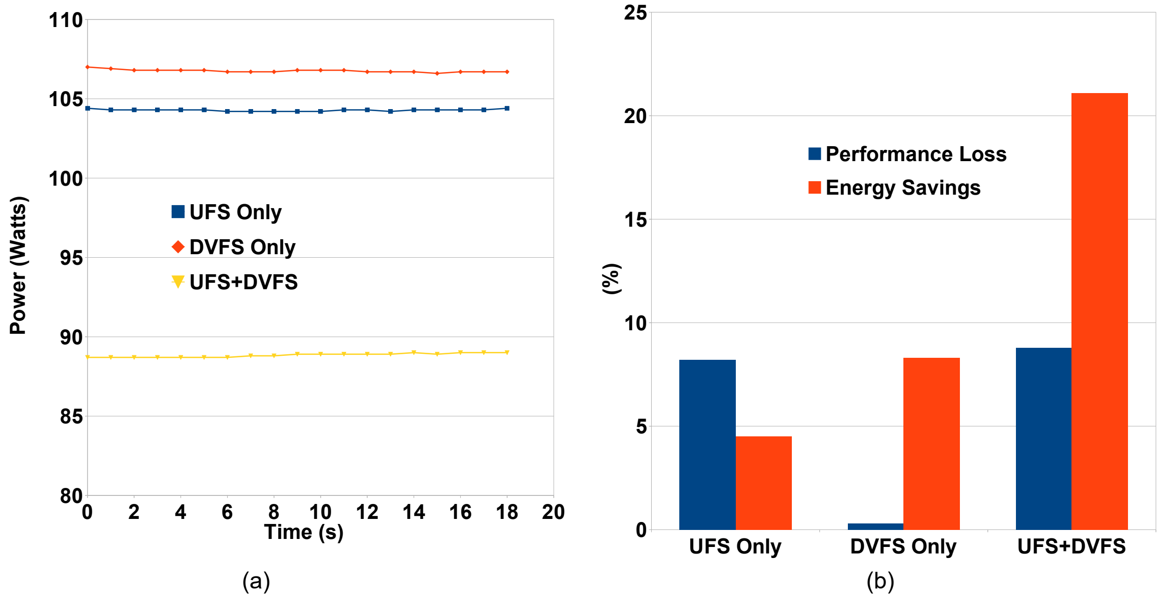


Figure 5: (a) Power consumption trace and (b) performance degradation and energy savings for UFS, DVFS and UFS+DVFS strategies *Slave* binding.

the energy consumption substantially for GAMESS execution under `UFS+DVFS` configuration but *Disjoint* is more efficient since it provides comparable energy saving at a much lower performance degradation.

The matrix-matrix multiplication is used in GAMESS for single (as `SGEMM`) and double (as `DGEMM`) precision calculations. Since the increase in precision leads in an increase in compute intensity of an application, the UFS application potential may not change drastically with the increase.

6 RELATED WORK

Power has become one of most important challenges in the design of next generation exascale system forcing the research community to continuously evaluate and redefine objectives of HPC power management. There have been two general approaches to obtaining energy savings during parallel application execution. The first approach is to focus on identifying stalls during the the execution by measuring architectural parameters from performance counters as proposed in (Ge, Feng, Feng, and Cameron 2007, Hsu and Feng 2005, Huang and Feng 2009). Rountree *et al.* (Rountree, Lownenthal, de Supinski, Schulz, Freeh, and Bletsch 2009), apart from using performance counters, do the critical path analysis to determine which tasks may be slowed down to minimize the performance loss in the parallel execution. Besides communications, Adagio also monitors computation parts of the application to determine suitable opportunities to apply DVFS. The second approach determines the communication phases to apply DVFS as, for example, in (Lim, Freeh, and Lowenthal 2006) and (Freeh and Lowenthal 2005).

With the 20 MW power limit set for exascale performance, many power limiting strategies have been proposed to operate a computing system under a given power budget. (Marathe, Bailey, Lowenthal, Rountree, Schulz, and de Supinski 2015) discusses *conductor*, which is a runtime system that dynamically distributes available power to the different compute nodes and cores based on the available slack to improve performance. (Ge, Feng, He, and Zou 2016) explores the coordinated power allocation between different components within a node and based on their observations, an optimal power allocation strategy is proposed. Authors in (Tiwari, Schulz, and Carrington 2015) propose performance and power models which predict the behavior of different HPC computations under varying power caps for different components in a node.

With the increase in cache sizes along with the placement of SoC (system-on-a-chip) components on the CPU die, the uncore power consumption is becoming quite significant considering the total processor power consumption (Loh 2008). Authors in (Gupta, Brett, Koufaty, Reddy, Hahn, Schwan, and Srinivasa 2012) study the uncore power consumption in heterogeneous platforms consisting of both high and low power cores using client device workloads and determine that potential energy savings are very much affected by the uncore component. (Chen, Xu, Kim, Gratz, Hu, Kishinevsky, Ogras, and Ayoub 2013) presents a strategy for applying frequency scaling to network-on-chip and the last level cache in the multicore processors using a PI controller. In this paper, the focus is on exploring the UFS capability on Haswell-EP processors and their effect on the component latency and power consumption.

7 CONCLUSIONS

In this paper, uncore frequency scaling (UFS) on the Haswell-EP platform was explored with respect to its application, effect on the DRAM, L3 cache latency and the node idle power consumption. It was also shown that, compared to the DRAM power limit, UFS decreases the DRAM bandwidth at a much lower rate. Finally, UFS and DVFS were compared in terms of their energy saving potential on quantum chemistry application GAMESS. On a 20 core Haswell-EP platform with different process bindings for GAMESS, experiments depicted that UFS is comparable to DVFS when it comes to energy savings and in conjunction with DVFS, it can save energy up to 21% during GAMESS execution. Future work will focus on developing runtime strategies which can dynamically apply UFS at runtime based on component utilization.

Performance models will be explored to link uncore frequency to the L2 cache miss rate, QPI bandwidth, memory controlle usage and, eventually, platform instructions per cycle (IPC). The power models are to quantify the effect of UFS on node power consumption while varying the degree of DVFS.

References

Chen, X., Z. Xu, H. Kim, P. V. Gratz, J. Hu, M. Kishinevsky, U. Ogras, and R. Ayoub. 2013. "Dynamic Voltage and Frequency Scaling for Shared Resources in Multicore Processor Designs". In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pp. 114:1–114:7. New York, NY, USA, ACM.

David, H., E. Gorbatov, U. Hanebutte, R. Khannal, and C. Le. 2010. "RAPL: memory power estimation and capping". In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, ISLPED'10, pp. 189–194. New York, NY, USA, ACM.

Fedorov, D., R. Olson, K. Kitaura, M. Gordon, and S. Koseki. 2004. "A new hierarchical parallelization scheme: Generalized distributed data interface (GDDI), and an application to the fragment molecular orbital method (FMO)". *Journal of Computational Chemistry* vol. 25, Issue 6, pp. 872–880.

Fedorov, D. G., and K. Kitaura. 2004. "The importance of three-body terms in the fragment molecular orbital method". *The Journal of Chemical Physics* vol. 120 (15), pp. 6832–6840.

Fletcher, G., M. Schmidt, B. Bode, and M. Gordon. 2000. “The Distributed Data Interface in GAMESS”. *Computer Physics Communications* vol. 128 (1–2), pp. 190–200.

Freeh, V., and D. Lowenthal. 2005. “Using multiple energy gears in MPI programs on a power-scalable cluster”. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 164–173.

Ge, R., X. Feng, W. Feng, and K. Cameron. 2007, Sep.. “CPU MISER: A Performance-Directed, Run-Time System for Power-Aware Clusters”. In *Parallel Processing, 2007. ICPP 2007. International Conference on*, pp. 18.

Ge, R., X. Feng, Y. He, and P. Zou. 2016, Aug. “The Case for Cross-Component Power Coordination on Power Bounded Systems”. In *2016 45th International Conference on Parallel Processing (ICPP)*, pp. 516–525.

Gordon, M., and M. Schmidt. 2005. “Advances in Electronic Structure Theory: GAMESS a Decade Later”. *Theory and Applications of Computational Chemistry:the first forty years, C.E.Dykstra, G.Frenking, K.S.Kim, G.E.Scuseria (Editors)*.

Gupta, V., P. Brett, D. Koufaty, D. Reddy, S. Hahn, K. Schwan, and G. Srinivasa. 2012. “The Forgotten ’Uncore’: On the Energy-efficiency of Heterogeneous Cores”. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC’12, pp. 34–34. Berkeley, CA, USA, USENIX Association.

Hackenberg, D., R. Schone, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer. 2015, May. “An Energy Efficiency Feature Survey of the Intel Haswell Processor”. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pp. 896–904.

J. Hoffmann September 2017. “Manually setting the Uncore frequency on Intel CPUs”. https://www.linkedin.com/pulse/manually-setting-uncore-frequency-intel-cpus-johannes-hofmann/. [Online; accessed 27-November-2017].

Hsu, C., and W. Feng. 2005, Nov.. “A Power-Aware Run-Time System for High-Performance Computing”. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pp. 1.

Huang, S., and W. Feng. 2009, May. “Energy-Efficient Cluster Computing via Accurate Workload Characterization”. In *Cluster Computing and the Grid, 2009. CCGRID’09. 9th IEEE/ACM International Symposium on*, pp. 68–75.

G. Lento September 2014. “Optimizing Performance with Intel Advanced Vector Extensions”. https://computing.llnl.gov/tutorials/linux_clusters/intelAVXperformanceWhitePaper.pdf. [Online; accessed 11-November-2017].

Lim, M., V. Freeh, and D. Lowenthal. 2006. “Adaptive, transparent frequency and voltage scaling of communication phases in MPI programs”. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*.

Loh, G. 2008. “The cost of uncore in throughput-oriented manycore processors”. In *In Proc. of Workshop on Architectures and Languages for Throughput Applications.*, ALTA.

Marathe, A., P. E. Bailey, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. de Supinski. 2015. *A Run-Time System for Power-Constrained HPC Applications*, pp. 394–408. Cham, Springer International Publishing.

McVoy, L., and C. Staelin. 1996. “Lmbench: Portable Tools for Performance Analysis”. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC ’96, pp. 23–23. Berkeley, CA, USA, USENIX Association.

Olson, R., M. Schmidt, M. Gordon, and A. Rendell. 2003. “Enabling the Efficient Use of SMP Clusters: The GAMESS/DDI Model”. In *SC*, pp. 41.

Rountree, B., D. Lownenthal, B. de Supinski, M. Schulz, V. Freeh, and T. Bletsch. 2009. "Adagio: making DVS practical for complex HPC applications". In *Proceedings of the 23rd international conference on Supercomputing*, ICS'09, pp. 460–469. New York, NY, USA, ACM.

Schmidt, M. W., K. Baldridge, J. Boatz, S. Elbert, M. Gordon, J. Jensen, S. Koseki, N. Matsunaga, K. Nguyen, S. Su, T. Windus, M. Dupuis, and J. J. Montgomery. 1993, Nov.. "General atomic and molecular electronic structure system". *J. Comput. Chem.* vol. 14, pp. 1347–1363.

Sundriyal, V., and M. Sosonkina. 2016. "Joint Frequency Scaling of Processor and DRAM". *The Journal of Supercomputing* vol. 72 (4), pp. 1549–1569.

Sundriyal, V., M. Sosonkina, A. Gaenko, and Z. Zhang. 2013. "Energy saving strategies for parallel applications with point-to-point communication phases". *Journal of Parallel and Distributed Computing* vol. 73 (8), pp. 1157–1169.

Tiwari, A., M. Schulz, and L. Carrington. 2015, May. "Predicting Optimal Power Allocation for CPU and DRAM Domains". In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pp. 951–959.

V. Vishwanathan July 2017. "Intel Memory Latency Checker". https://software.intel.com/en-us/articles/intelr-memory-latency-checker. [Online; accessed 27-November-2017].