Providing Insight into the Performance of Distributed Applications Through Low-Level Metrics

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

David Eberius

August 2020

Copyright © by David Eberius, 2020
All Rights Reserved.

To my parents Bill and Natalie Eberius, and my siblings Rachel, Adam, and Megan for their love, support, and motivation.

Acknowledgments

I want to first express my sincere gratitude to my advisor, Dr. Jack Dongarra, for taking me on as his student and giving me a position as a Graduate Research Assistant in the Innovative Computing Laboratory (ICL). His leadership and guidance always ensured that I had funding to continue my research and fueled my passion for High-Performance Computing. It has been a great privilege to work with him, and I will always treasure the knowledge and experience he has given me throughout these years.

I am also eternally grateful to Dr. George Bosilca, my group leader in the DisCo group, for always encouraging me to dig deeper and strive for improvement, saying things like: "Ok, but why does it do that?" and "That's good, but can we do better?". He was able to provide a relaxed atmosphere in the office, encouraged discussion, and always had a willingness to teach. Without his guidance, I would not have been able to finish this research!

I am so lucky to have met Dr. Olga Pearce by happenstance at SC17, and to subsequently work for her over the course of two summers at Lawrence Livermore National Laboratory. She always encouraged me to explain why my results looked the way they did, and to provide solid evidence, which led me to a much better understanding of the work. She also provided me with valuable insight into working at a National Laboratory, which led me to decide to pursue a career working at National Laboratories.

I would like to thank Dr. Yingkui Li, Dr. Michael Berry, and Dr. Gregory Peterson for serving on my dissertation committee. Their help and guidance provided me with a more clear direction in my research which has culminated in this dissertation.

My family and friends have provided me with so much love, support, motivation, and confidence throughout this process that cannot be understated. My family always pushed me to finish with phrases like, "When are you graduating?", and welcomed me home during

breaks. My friends always had much more confidence in me and my work than I did and provided much needed emotional support along the way. I am so incredibly thankful for all they have done!

Finally, I would like to thank my colleagues at ICL for teaching me all about the various aspects of HPC, providing excellent discussion and deep dives into meaningless topics during coffee breaks, great companionship, and encouragement of my competitive whistling (particularly Dr. Anthony Danalis). I really appreciate the wisdom and companionship from Dr. Thomas Herault, Dr. Aurelien Bouteiller, Dr. Damien Genet, Dr. Anthony Danalis, Dr. Heike Jagode, Dr. Hartwig Anzt, Dr. Piotr Luszczek, Dr. Jakub Kurzak, Dr. Azzam Haidar, Sam Crawford, Earl Carr, and many more staff at ICL. Last, but not least, I would like to thank my fellow students Dr. Wei Wu, Dr. Chongxiao Cao, Dr. Reazul Hoque, Dr. Thananon Patinyasakdikul, Dr. Xi Luo, Jiali Li, Mike Tsai, Dong Zhong, Yu Pei, Sangamesh Ragate, Hanumantharayappa, Qinglei Cao, Yicheng Li, and many others for their friendship and camaraderie, I will never forget it!

Abstract

The field of high-performance computing (HPC) has always dealt with the bleeding edge of computational hardware and software to achieve the maximum possible performance for a wide variety of workloads. When dealing with brand new technologies, it can be difficult to understand how these technologies work and why they work the way they do. One of the more prevalent approaches to providing insight into modern hardware and software is to provide tools that allow developers to access low-level metrics about their performance. The modern HPC ecosystem supports a wide array of technologies, but in this work, I will be focusing on two particularly influential technologies: The Message Passing Interface (MPI), and Graphical Processing Units (GPUs).

For many years, MPI has been the dominant programming paradigm in HPC. Indeed, over 90% of applications that are a part of the U.S. Exascale Computing Project plan to use MPI in some fashion [7]. The MPI Standard provides programmers with a wide variety of methods to communicate between processes, along with several other capabilities. The high-level MPI Profiling Interface has been the primary method for profiling MPI applications since the inception of the MPI Standard, and more recently the low-level MPI Tool Information Interface was introduced.

Accelerators like GPUs have been increasingly adopted as the primary computational workhorse for modern supercomputers. GPUs provide more parallelism than traditional CPUs through a hierarchical grid of lightweight processing cores. NVIDIA provides profiling tools for their GPUs that give access to low-level hardware metrics.

In this work, I propose research in applying low-level metrics to both the MPI and GPU paradigms in the form of an implementation of low-level metrics for MPI, and a new method for analyzing GPU load imbalance with a synthetic efficiency metric. I introduce

Software-based Performance Counters (SPCs) to expose internal metrics of the Open MPI implementation along with a new interface for exposing these counters to users and tool developers. I also analyze a modified load imbalance formula for GPU-based applications that uses low-level hardware metrics provided through *nvprof* in a hierarchical approach to take the internal load imbalance of the GPU into account.

Table of Contents

1 Introduction **1**

1.1 Dissertation Statement 4

1.2 Contributions 6

1.2.1 Software-based Performance Counters 6

1.2.2 GPU Load Balancing 7

1.3 Dissertation Organization 8

2 Literature Review **10**

2.1 HPC Performance Analysis Tools 10

2.1.1 Data Collection Tools 11

2.1.2 Performance Analysis Tools 15

2.1.3 MPI-Specific Tools 17

2.1.4 GPU Performance Analysis Tools 20

2.2 Load Imbalance 21

2.2.1 Distributed Load Imbalance 21

2.2.2 GPU Load Imbalance 21

3 MPI Performance Analysis and Tool Support Through Software-based Performance Counters **23**

3.1 Chapter Overview 23

3.2 Introduction 24

3.3 Background 24

3.3.1 The MPI Profiling Interface (PMPI) 25

3.3.2 The QMPI Interface 26
3.3.3 The MPI Tool Information Interface (MPI_T) 27
3.3.4 MPI Implementations 30
3.4 Motivation 31
3.5 Performance Metrics Exposed 33
3.5.1 Types of Events 33
3.5.2 SPC Metrics 35
3.6 Design and Implementation 40
3.6.1 SPC Data Structures 40
3.6.2 SPC Update Functionality 42
3.7 Overhead of SPCs 42
3.7.1 Instrumentation Overhead 42
3.7.2 Testing the Overhead Cost of SPCs 45
3.7.3 mpiP Overhead 53
3.7.4 Proxy Application Overhead 55
3.8 SPC Reporting Methods 56
3.8.1 Printing to *stdout* 56
3.8.2 MPI_T Performance Variables 57
3.8.3 SPC *mmap* Interface 57
3.8.4 SPC Snapshot Functionality 59
3.9 SPC Use Case Results 59
3.9.1 Diagnose MPI Implementation Issues: Out of Sequence Messages Case Study 61
3.9.2 Identify Application Bottlenecks 66
3.9.3 Workload Characterization 67
3.10 Conclusion 77

4 GPU Load Imbalance **79**
4.1 Chapter Overview and Acknowledgment 79
4.1.1 Auspices 79

4.1.2 Disclaimer . . . 80
4.2 Introduction . . . 80
4.3 Background . . . 81
4.3.1 The CUDA Programming Environment . . . 81
4.3.2 Load Imbalance Between MPI Processes . . . 82
4.3.3 CoMD Proxy Application . . . 83
4.4 Design and Implementation . . . 86
4.4.1 Introducing RAJA . . . 86
4.4.2 Refactoring for More Optimal GPU Usage . . . 87
4.4.3 Instrumentation and Profiling . . . 90
4.4.4 Metric-Driven Load Balancing . . . 91
4.5 Results and Analysis . . . 94
4.5.1 Experimental Setup . . . 94
4.5.2 Overview and CPU Verification . . . 97
4.5.3 Comparing Measured GPU Load and Adjusted GPU Load . . . 100
4.6 Conclusions . . . 105

5 Conclusions **106**
5.1 Conclusions . . . 106
5.2 Suggestions For Future Work . . . 109
5.2.1 Software-based Performance Counters . . . 109
5.2.2 GPU Load Imbalance . . . 111

Bibliography **112**

Appendices **120**
A List of SPCs . . . 121
B SPC Example Code . . . 134

Vita **140**

List of Tables

3.1 Some examples of SPCs from the MPI and PML levels of the Open MPI codebase. 36

3.2 Configuration of the testing system, *Arc*. 45

3.3 Counters used in the NetPIPE benchmark. 49

3.4 Results of the multirate benchmark using the pairwise communication pattern with: 2, 4, and 8 threads, a window size of 256, message size of 64 bytes, and iteration count of 100. Note: The message rate and wall time do not include the warm-up phases, but the other values do. Without warm-up messages there are $256 \times 100 \times N_t$ messages sent where N_t is the number of threads. . 62

3.5 The results of the MADNESS *moldft* tests using five water molecules. The counter values are the average of 10 runs with 18 threads per run of the simulation for each configuration. Note: the total time is the wall time reported by *moldft*. 65

3.6 The MPI collective operations used in the LAMMPS *indent* test. 68

3.7 The MPI point-to-point operations used in the LAMMPS *indent* test. 70

4.1 Detailed parameters and statistics for both the small and large problem sets. 98

1 A list of the currently available SPCs in my pull request to the Open MPI development repository. Note: '*' represents 'OMPI_SPC'. Table 1 is continued on to pages 122-133. 121

List of Figures

2.1 A state machine provided by Keller et. al. in their introductory papery for the Peruse interface [31]. This shows a potential sequence of events in the Peruse interface in Open MPI. 19

3.1 An example control flow of the hierarchical tool wrappers with pre-processing and post-processing capabilities in the QMPI interface. 27

3.2 The data layout for the contiguous SPC data buffer, with **N** being the number of SPCs, and **M** being the number of bin counters. Note: Each pair of rules and values arrays are cache line aligned in order to avoid false sharing between separate bin counters. 41

3.3 The overhead of adding SPCs to the code while leaving all of them turned off. Note: the error bars represent the standard deviation across the 10 test runs. 46

3.4 The overhead of adding SPCs to the code and turning all of them on. Note: the error bars represent the standard deviation across the 10 test runs. . . . 47

3.5 Comparing the intra-node overhead within a single socket with the counters all on, all off, only OMPI_MATCH_TIME turned on, or only the counters from Table 3.3 minus OMPI_MATCH_TIME. Note: the error bars represent the standard deviation across the 10 runs. 50

3.6 The overhead of adding SPCs to the code and turning on only the counters from Table 3.3 minus OMPI_MATCH_TIME. Note: the error bars represent the standard deviation across the 10 runs. 51

3.7 The overhead of adding SPCs to the code in the Core case when using cycles instead of converting to microseconds. This also looks only at the counters from Table 3.3 in all on, all off, and all on minus OMPI_MATCH_TIME. Note: the error bars represent the standard deviation across the 10 runs. 52
3.8 The overhead of using mpiP with NetPIPE. Note: the error bars represent the standard deviation across the 10 runs, however the deviation was extremely small so they appear nonexistent. 54
3.9 The overhead of having all SPCs turned on while running the LAMMPS proxy application. 55
3.10 A diagram of the operation of the SPC *mmap* interface. Note: the "_0" in the data file name refers to the process rank (rank 0). 58
3.11 A diagram of the operation of the SPC snapshot feature using the *mmap* interface. Note: the write operation always creates a new file with the current timestamp appended to the end of the name, and the "_0" in the names refers to the process rank (rank 0). 60
3.12 Heatmaps of the bytes sent/received by MPI during a run of the LAMMPS *indent* test with 40 MPI processes across two nodes. Each box represents the counter values added in a 0.5 second time slice of the application run. 69
3.13 A visualization of the recursive doubling algorithm for an MPI_Allreduce as implemented in Open MPI. This assumes that there are 10 processes, each with a starting value of 1 with a summation operation. The 'R' labels indicate the MPI rank, and the 'C' labels indicate the number of round-trip communications performed on a given rank. 71
3.14 Heatmaps of the bytes sent/received by the user during a run of the LAMMPS *indent* test with 40 MPI processes across two nodes. Each box represents the counter values added in a 0.5 second time slice of the application run. 72
3.15 An approximate representation of the heatmaps in Figure 3.14 with the processes arranged in an 8×5 processor grid. 73

3.16 Heatmaps of the time spent in the process of matching during a run of the LAMMPS *indent* test with 40 MPI processes across two nodes. Each box represents the counter values added in a 0.5 second time slice of the application run. Note: The timer counter values have been converted to microseconds. . 75
3.17 Heatmaps of the number of unexpected messages and time spent inserting those messages into the queue during a run of the LAMMPS *indent* test with 40 MPI processes across two nodes. Each box represents the counter values added in a 0.5 second time slice of the application run. Note: The timer counter values have been converted to microseconds. 76

4.1 Molecular dynamics definitions 84
4.2 Introducing load imbalance in CoMD 85
4.3 The average number of interactions per thread across all of the threablocks in the original implementation of the GPU-based CoMD. 88
4.4 GPU efficiency: SM and warp usage. 91
4.5 Atom removal histograms for 60% atom imbalance problems for both small and large amount of work per GPU. 95
4.6 Imbalance percentage for each amount of atom removal for the small test case (256K ceiling and 80K floor atoms per process). Note: The *Atoms* values are the ground truth for work imbalance. 96
4.7 Measured and adjusted time vs atom values with 60% atom imbalance for both small and large test cases. All values are normalized by the maximum of their respective data sets. 100
4.8 Measured and adjusted time value differences with atom values. Using 60% atom imbalance for both large and small test cases. All values are normalized by the maximum of their respective data sets. 101
4.9 Correlation coefficients of small and large problem sizes for the 60% imbalance case. 103

4.10 Correlation coefficients of measured/adjusted load (time) compared to workload (atom count) for the various initial atom imbalances for both small and large test cases. 104

Chapter 1

Introduction

In the pursuit of ever faster supercomputers, High-Performance Computing (HPC) research has led to a vast ecosystem of hardware and software technologies that continue to push the limits of the inherent expression of parallelism. There is a divergent relationship between hardware and software research in that hardware vendors are trying to better serve the current and future needs of their customers through new hardware, and software research, by necessity, must delve into the best utilization of current hardware. This creates a cycle where software research delves into one set of hardware until new hardware is released, and since not all of the previous research still applies, the software has to adapt to the new hardware.

In order to maximize parallelism, modern supercomputers often combine a variety of computation, data movement, and data storage resources. A modern system could have multiple CPUs with dozens of cores per CPU, multiple GPUs with several nested levels of parallelism within, several levels of cache per CPU, multiple NUMA nodes with their own RAM, on-node solid state burst buffers, spinning disk storage on the node, multiple NICs, a hierarchical network interconnect, and a parallel file system. To run at the full capacity of such a system, an application will most likely combine several programming paradigms such as MPI[20] for the data movement and distributed communication, OpenMP[40] for managing CPU threads, and CUDA[39] for performing GPU computation. With all of these moving parts, it can be overwhelming for a programmer to understand why their application

performs the way it does, especially since much of the underlying complexity is handled by the various programming paradigms to ease the difficulty of development.

When developing on a modern supercomputer, parallelizing an application is one of the first steps to a potential shorter time to solution. The parallelization step is usually supplemented by a performance analysis stage where performance bottlenecks (either in the algorithm itself or in the communication pattern) are identified and addressed. There are many approaches to analyzing the performance of parallel applications. One of the predominant approaches is to add some form of instrumentation to the code and to use the data from this instrumentation to understand how the code is operating.

MPI

For many years now, the Message Passing Interface (MPI) has been the standard paradigm for implementing parallel applications in a distributed memory setting. The MPI Standard has expanded over time to not only include point-to-point message-passing, but also topics such as collective communications, group and communicator concepts, one-sided communications, I/O, a profiling interface, and a tool information interface [20]. These capabilities, along with the high performance provided by many MPI implementations, have led to MPI being used extensively for writing parallel applications on distributed systems across academia and industry alike.

Profiling of MPI applications often uses the MPI profiling interface (PMPI) that allows tools to preempt all MPI function calls and add instrumentation or other functionality around a call to a name shifted version of the MPI function with a prefix of 'PMPI_' instead of 'MPI_'. Many tools like Vampir [9], Paraver [30], TAU [51], and mpiP [60] use the PMPI interface to profile MPI applications, mainly through inserting timing functionality to track when MPI functions start and complete. This information is generally stored in a binary trace file and is available to the tools post-mortem for thorough analysis. This method provides an overview of how the application progressed overall but cannot expose low-level details and therefore provides little insight into what was happening within the MPI implementation.

The MPI performance revealing extension interface (*Peruse*) [31] was developed as a means to complement this lack of fine grained details in the PMPI interface, and to provide

more insight into MPI implementation performance. For example, using the Peruse interface, a tool could have access to detailed MPI state change information such as when a send request enters the queue of posted messages or when a communication request is completed. Essentially, Peruse allows for a tool to follow the life cycle of an MPI communication through the library. This interface had great potential for performing in-depth analysis of the state changes experienced by each communication, however it does not provide information about what is happening within those states. The Peruse interface was not accepted into the MPI Standard and has not seen widespread adoption.

The MPI_T interface was introduced to the MPI Standard as an official way to expose low-level information in the MPI implementation. The MPI_T interface allows MPI implementation developers to expose internal implementation variables to tools and users in the form of control variables and performance variables. Control variables are meant to contain properties and configuration settings of the MPI implementation such as the current eager limit or transport protocol [20]. Performance variables are meant to store implementation specific performance information such as internal queue sizes or data usage [20]. The MPI Standard does not specify any default variables, so it is up to the MPI implementation to determine which internal variables to expose through the MPI_T interface.

The Open MPI [22] implementation uses a Modular Component Architecture (MCA) [61], which allows for dynamic loading of different components of the library depending on the configuration that is specified through MCA parameters at run time. Many of these MCA parameters are registered as MPI_T control variables, which allows for dynamic configuration of Open MPI through the MPI_T interface. Before my work, there were almost no MPI_T performance variables registered in Open MPI.

GPUs

On the hardware side of things, one of the growing trends in HPC is the use of accelerator technologies such as Graphical Processing Units (GPUs) for the bulk of the computational workload. In the November 2019 top500 list of supercomputers, 145 machines used some form of accelerator, and 137 of those were GPUs (136 NVIDIA and 1 AMD)[53]. The main

advantage of GPU accelerators over standard CPUs is their capability to employ massive levels of parallelism on the order of thousands of concurrent hardware threads compared to tens of hardware threads on a modern CPU. The hardware threads on GPUs are much more lightweight than on a CPU, with minimal resources such as registers and cache memory attributed to each thread. The GPU hardware minimizes the impact of such restrictions through optimizations like maximizing memory bandwidth to supply work to all of the threads simultaneously, and providing fast context switching to hide memory stalls.

When it comes to profiling GPU applications, developers are typically limited to the profiling capabilities provided by the GPU vendors. For NVIDIA GPUs, this typically means using NVIDIA's support software for their CUDA programming environment such as the *nvprof* profiling tool and the CUDA Profiling Tool Interface (*CUPTI*) [39]. There are also some third party tools such as PAPI [38], Vampir [9], Caliper[8], and TAU[51] that provide access to CUDA profiling information.

MPI and GPU Internal Runtime Systems

Both the MPI and CUDA programming environments have runtime systems that take care of a lot of the complexities of getting their programming models to work. The MPI runtime handles all of the process management, data movement, buffer allocation and management, and network and transport protocols among other things. The CUDA runtime hides the complexity of things like scheduling kernel launches across the streaming multiprocessors (SMs) and compute cores and managing data movement between host and device. All of these hidden operations can have huge effects on the performance of a user application, so it is essential to have a way to get information about what is going on within these hidden operations so you can analyze why the program is behaving a certain way.

1.1 Dissertation Statement

In order to understand the performance of parallel applications in a distributed environment, it is essential to have access to low-level profiling information about the programming

environments provided on distributed systems. In this study, I will be focusing on the MPI and CUDA programming environments.

The MPI Standard provides the MPI_T interface to allow MPI implementations to expose such low-level information to tool developers, however this does not require MPI implementations to take advantage of this capability. The MPI_T interface can also be quite difficult to use as it requires the user to register a context in which an MPI_T variable exists along with potentially attaching that variable to an MPI object such as a communicator and then performing a read operation which adds overhead by copying the value of the variable into a user buffer.

When programming with GPUs in the CUDA environment, many of the specifics of the various levels of scheduling from kernels down to threads are hidden from the user. This can make it difficult to understand how the workload is spread across the GPU. The metrics provided by the CUDA profiling system can provide some insight into what is going on inside the GPU, however the descriptions of these metrics can be vague, and the values stored in the metrics can lack the level of detail necessary to get the full picture.

In this study, I explore the implementation, expression, and usage of low-level metrics for understanding the MPI and CUDA programming environments in a distributed system. For my study of the MPI programming environment, I will focus on the Open MPI [22] implementation of the MPI Standard, which makes extensive use of MPI_T control variables, but does not take advantage of MPI_T performance variables. The Open MPI codebase is open source, which allows me to explore the implementation and expression of low-level performance metrics while also providing a platform for testing their usage. With the CUDA programming environment, the implementation is not provided as open source, so I will study the usage of the existing capabilities. I use load balancing in a molecular dynamics simulation as a case study for using metrics provided by CUDA profiling tools to understand what is happening within the GPU.

1.2 Contributions

In this work, I contribute to performance analysis in HPC in two main ways: **(1)** Introducing an interface for defining and exposing low-level performance metrics in MPI both through the existing MPI_T interface and through a new mmap-based interface; and **(2)** Evaluating a novel hierarchical GPU load imbalance formula using a composite GPU efficiency metric derived from existing GPU metrics, within my extension to a molecular dynamics proxy application as a case study and comparison of CPU vs. GPU load imbalance.

1.2.1 Software-based Performance Counters

I address the need for low-level performance information about the operation of implementations of the MPI Standard by analyzing the capabilities of the currently available methodologies and then providing my own method which expands upon previous work. The PMPI interface remains the primary method for profiling MPI applications, which only allows for high-level performance information about the operation of MPI such as time spent performing MPI functions and call-site analysis. The introduction of the MPI_T interface allows for internal MPI variables to be exposed as performance variables, however with no default variables defined in the MPI Standard, many MPI implementations have been slow to add their own MPI_T performance variables. I investigate the capabilities of the MPI_T interface and assess its strengths while also identifying the limitations and drawbacks of this interface as it is designed.

The primary strengths of performance variables in the MPI_T interface are: they provide a generic way for tools to expose internal variables; tracking of individual variables can be enabled or disabled at any time; variables can be defined such that they are attached to a particular MPI construct (such as an MPI communicator); and the variables can be read at any time during the application. The MPI_T interface is designed to simply provide internal variables to performance analysis tools and the tools are responsible for storing and reporting the information in a useful manner. This means that MPI_T variables are only accessible during runtime, and only through the MPI_T interface.

In this work, I introduce Software-based Performance Counters (SPCs) [15] to a particular MPI implementation, Open MPI. These SPCs act as a complementary interface to MPI_T within Open MPI to provide access to low-level performance metrics. At their core, SPCs are integer counters that keep track of various information in the implementation. All of these counters are registered as MPI_T performance variables so they can be accessed through that interface, however they can also be stored in a shared data file allocated using the *mmap* function such that any MPI process can attach to this file and have read-only access to the counters. The SPC interface also has the capability to store snapshots of these data files and keep a persistent copy of the data file, which allows for post-mortem analysis of the SPC values. This work looks into several uses cases of SPCs, including diagnosing issues in Open MPI, identifying application performance bottlenecks, and machine workload characterization.

1.2.2 GPU Load Balancing

I address the need for an accurate formula for load imbalance in distributed systems using GPUs by showing that the existing formula for load imbalance in CPU-only systems can be inaccurate when applied to GPU computation, and I use a new composite GPU efficiency metric to evaluate a proposed formula that attempts to improve accuracy by taking into account internal GPU imbalance. On CPU-only architectures, load imbalance is defined as the scaled maximum load on any CPU core minus the average [44]. This formula relies on two major assumptions: (1) all CPU cores perform computations at roughly the same rate; and (2) a CPU core is the smallest unit upon which work can be scheduled.

Assumption (1) can be problematic in general when you consider systems in which CPUs and GPUs can have their frequencies scaled dynamically. For simplicity, I will be assuming that the CPU and GPU frequencies are held constant. Assumption (2) works nicely for CPU-only machines, though there are cases where a system has the capability to use hyperthreading to schedule multiple logical threads one physical core. This is done through duplicating some CPU resources; however the execution units are typically not duplicated, effectively allowing only one thread to perform computations at a time. For simplicity, I will be ignoring hyperthreading.

On a GPU-based system, assumption (2) becomes problematic because each GPU is broken down into several additional levels of parallelism. On a system with multiple GPUs, if one were to assume that each GPU is equivalent to a CPU core for the purposes of calculating load imbalance, there would be a significant loss of accuracy with the formula used in CPU-only systems.

In this work, I evaluate a modified load imbalance formula for use on GPU-based systems. I created a composite GPU efficiency metric to use in this formula, which is composed of GPU metrics from NVIDIA's *nvprof* tool. This is used to estimate load within the GPU's various levels of parallelism to provide a hierarchical look at the load imbalance. This formula provides increased accuracy over the CPU-only formula and provides a way forward for making more generalized load imbalance formulas.

1.3 Dissertation Organization

This dissertation shall be laid out as follows:

- Chapter 2 encompasses a literature review of research related to the topics introduced in this dissertation with particular focus on data collection, profiling, and performance analysis tools for MPI and GPU-based applications.

- Chapter 3 introduces Software-based Performance Counters; and includes an in-depth discussion of relevant background information from the MPI Standard as well as best practices in profiling MPI applications, a discussion of the motivations for creating the SPC interface, an evaluation of the types of events of interest and corresponding SPC metrics, a detailed look at the design and implementation of SPCs, an evaluation of the overhead introduced to Open MPI by SPCs, a comparison to existing approaches to profiling MPI applications, an evaluation of the various reporting methods for SPCs, and a discussion of use cases for SPCs.

- Chapter 4 includes a discussion of the relevant background information on the CUDA programming environment and load imbalance in MPI applications, provides an introduction to the CoMD proxy application, details my extensions to the CoMD

proxy application to provide a test bed for CPU and GPU imbalance studies including performance enhancements and my profiling and instrumentation efforts, introduces a new GPU efficiency metric derived from *nvprof* metrics, evaluates a load balancing formula for GPU-based systems, a discussion of the insufficiency of the existing formula for CPU-only systems, and an evaluation of the accuracy of the new formula.

- Chapter 5 will wrap up this dissertation with a summary of the conclusions I arrived at throughout my research of distributed performance analysis in HPC through low-level metrics, and a discussion of future work in this area.

Chapter 2

Literature Review

There is an abundance of research that has been conducted into performance analysis of distributed applications and the software stack that supports such analysis. There are several areas of focus in this research such as: the collection of data about an application, visualization methods for collected data, and analysis techniques for understanding the data. In this section, I will be discussing the existing research into distributed performance analysis with a particular focus on the HPC field in the areas of MPI and GPU performance tools and analysis methods.

2.1 HPC Performance Analysis Tools

Conducting performance analysis of HPC applications provides a unique challenge for HPC performance analysis tools in dealing with the massive scale inherent to supercomputers. In application performance analysis, there are two primary phases: data collection, and data analysis. There has been extensive research into both of these phases, which has resulted in a wealth of tools that are available to application developers for providing different views of application performance. These HPC performance analysis tools tend to either specialize in collecting data during an application's lifetime, or providing support for postmortem analysis, though there are some tool suites that provide both.

2.1.1 Data Collection Tools

In order to conduct performance analysis, one must first collect information about the performance of the application. Data collection tools are specialized for gathering this performance information, and typically follow one of two approaches to data collection: instrumentation-based or interrupt-based. Instrumentation-based approaches require additional code to be added to an application in order to gather information at specific points in the code, which can be done manually by a programmer or dynamically through libraries like Paradyn [36]. The interrupt-based approach does not require additional code to be added to the application, it simply interrupts the application's execution periodically and gathers information at those points in time.

PAPI

The Performance Application Programming Interface (*PAPI*) provides an interface for accessing hardware performance counters available from many modern microprocessors [38]. PAPI is highly portable with support for most hardware vendors and operating systems and provides a wealth of information with its counters such as the number of cache misses and the total number of instructions issued. PAPI can gather information from the lowest level possible with information that is stored in reserved registers on a given hardware platform.

Originally, the PAPI library was focused on hardware counters from the CPU and memory on a system, but it has been extended over the years to include a variety of additional components for collecting and manipulating counters from other sources all of which can be monitored simultaneously[57]. Some components of note are the components for monitoring hardware events on both NVIDIA and AMD GPUs, network interface counters, counters associated with monitoring and capping power on a variety of architectures, and a software defined events (SDE) component [35] [24] [28].

The SDE component is an interesting departure from the typical focus on hardware-based events in PAPI. The SDE component is designed to allow library developers to provide PAPI access to the library's internal variables, so those variables can be exposed as PAPI events. This interface operates through weak symbols for SDE interface functions in the target library

and sharing pointers between the target library and PAPI. In this way, the SDE interface is able to add no overhead to the target library when PAPI is not linked with the application, and only minimal setup overhead in the majority of cases where PAPI is linked with the application. This approach shares some similarities with the MPI_T interface discussed in Section 2.1.3.

Paradyn

The *Paradyn* tool provides a method for dynamically instrumenting applications [36]. Essentially, this works by inserting instrumentation into an existing binary at runtime. The idea is that potential locations for instrumentation can be determined through the use of some form of monitoring daemon, and instrumentation can be added to these areas of interest when needed and removed when there is no longer a need.

The Paradyn project has made this technology available through the *Dyninst* API, which allows for more general purpose code insertion into a running program [41]. This can of course be extended to uses outside of inserting profiling code, such as dynamic algorithm selection [3].

TAU

The Tuning and Analysis Utilities (*TAU*) tool framework is designed to allow for profiling and tracing of parallel applications in a variety of languages and provides a graphical user interface for analysis [51]. TAU supports both manual code instrumentation and dynamic instrumentation through the Dyninst API [41].

TAU maintains information about code constructs like functions and blocks for several different levels of parallelism in an application such as threads and nodes [56]. The standard use case is for TAU to provide timing information of these code constructs; however it is also possible to track other information such as hardware performance counters through PAPI. The TAU framework also includes visualization and analysis tools and a capability to create binary trace files designed to resemble Gantt charts or parallel timelines when visualized by tools like *Vampir* and *Paraver*, discussed in further detail in Section 2.1.2.

Scalasca

The *Scalasca* tool specifically targets performance analysis of applications using the MPI and OpenMP programming paradigms on large-scale systems [23]. The focus of performance analysis with Scalasca is in identifying bottlenecks introduced by communication and synchronization events. Scalasca operates in one of two analysis modes, profiling mode and tracing mode, each with their own type of data collection [49].

Profiling mode collects data such as hardware counters or other metrics for each function call path. This provides a way to quickly identify hotspots in the codebase. In tracing mode, Scalasca additionally records specific events that can help identify program wait states such as a delayed sender which could force the corresponding receiver to wait on that data. Scalasca also provides custom visualization and analysis tools and can export binary trace files which can be visualized with tools like Vampir and Paraver as discussed in Section 2.1.2.

Score-P

The *Score-P* performance measurement infrastructure was born out of a need for providing a common interface for redundant capabilities across the different performance analysis tools [33]. These redundant capabilities are things like code instrumentation, data collection, and data storage. Score-P is able to provide a common infrastructure for these tools, and supports several tools discussed in this chapter such as TAU, Scalasca, Pariscope, and Vampir [51] [23] [5] [9].

Score-P provides utilities that facilitate adding code instrumentation and data collection, such as the *scorep* command-line tool. This instrumentor tool is used as a prefix to the normal command line string for compiling the application. The idea is that this tool will detect which of the supported programming paradigms is being used, such as OpenMP or MPI, and add the appropriate flags to allow for appropriate instrumentation for that paradigm. This step only needs to happen once, and still allows for switching the Score-P configuration properties such as the mode (profiling and/or tracing). In addition, the data storage for Score-P uses pre-allocated thread-local storage to alleviate the overhead added for memory allocation and data movement at run time.

Caliper

The *Caliper* tool is an abstraction layer for performance introspection that is built on the principle of separating mechanism from policy [8]. The idea is that Caliper provides access to a wide variety of mechanisms by which data can be collected such as instrumentation, sampling, hardware counters, call stack information, and traces, which can then be used in various combinations as data collection policies at runtime.

Standard Caliper operation involves three primary concepts: *attributes*, the *blackboard*, and *snapshots*. The *attributes* represent individual points of data, and these attributes are stored in the *blackboard* which is a global buffer. A *snapshot* is a measurement event, which essentially means that the current contents of the *blackboard* are written to a *snapshot* record along with any measurements provided by data collection services that are on-demand such as collection of time stamps or reading hardware counters.

HPCToolkit

The *HPCToolkit* provides a series of tools that target the different stages of performance analysis [1]. In this section, I will focus on the two data collection tools within HPCToolkit: *hpcrun* and *hpcstruct*.

The *hpcrun* tool uses a sampling approach that collects samples both at certain time intervals, and at triggered events based on performance metrics. This provides calling-context-sensitive performance measurements that can be used by the analysis tools in HPCToolkit [17]. The *hpcstruct* tool is a companion to the *hpcrun* tool that associates the performance measurements with the underlying source code.

LDMS

The Lightweight Distributed Metric Service (*LDMS*) is a system monitoring tool that allows for high-fidelity monitoring of a number of different metrics across the different levels of the system [2]. LDMS is designed to run continuously across an entire system and collect data at a sufficiently high frequency to provide for useful analysis. One of the motivating factors for

this work is that applications running on large systems can be affected by other applications on the system interfering with shared resources such as the network and the file system.

LDMS operates using three main components: *Samplers*, *Aggregators*, and *Storage*. The *Samplers* collect information periodically from sampling plugins, each of which collect a set of performance metrics. The *Aggregators* pull information from the samplers, and potentially other *Aggregators*, periodically and can stage this data for storage. The *Storage* component supports a number of different file formats, and only allows writing of valid updated metric data from an *Aggregator* that has been configured to write data to storage.

2.1.2 Performance Analysis Tools

Of course, collecting performance data is just the first step, there must be tools for analyzing and visualizing that data in order to understand the performance. There are several different approaches to visualizing performance data, but two of the more common ones are: trace visualizers that have information associated with events in a timeline, and program callstack trees with information associated with specific program elements. When it comes to performance analysis, there are various approaches, some that aim to be more general purpose, and others that are tailored to identify a specific class of performance characteristics.

Trace Files

One of the more popular formats for representing performance data is with trace files. These files typically include a series of time stamps for the beginning and end of events of interest, along with performance metrics and metadata such as the processing unit associated with the event. The *Paraver* and *Vampir* tools provide the capability to parse the data from such trace files, and display that data, typically in a timeline that is similar to a Gantt chart [30] [9].

Vampir

The *Vampir* toolset primarily relies on the Score-P tool for providing performance data, but can accept data from a number of different sources [59]. Vampir provides many highly

polished visualization options for performance data, including a hierarchical timeline that can be edited to highlight particular functions or blocks with certain properties, the capability to combine multiple metrics into composite metrics, and much more. Unlike most of the tools discussed here, which are open source, Vampir has a commercial licence.

Periscope

The *Periscope* tool provides a method for performing online rather than static analysis of distributed performance with a particular focus on the MPI and OpenMP programming paradigms [5]. Periscope operates by preprocessing user code files in order to add instrumentation, and then generating new instrumented object files which are linked with the Periscope monitoring library.

Once this preprocessing is done, the application can be run through Periscope and can take advantage of repetitive code regions or simply reexecute the application entirely to construct detailed analysis on code regions. This analysis is based on metrics taken throughout the various executions by different metrics such as hardware counters from PAPI or MPI function timing information.

TAU's *paraprof* Tool

TAU provides a tool called *paraprof* that allows for a wide variety of visualization and analysis options for performance data generated from TAU instrumentation. Depending on the type of instrumentation that was added to the code, *paraprof* can display the data in graphical visualizations such as 3-D scatter plots, thread-based displays such as thread statistics tables, breakdowns of individual functions like histograms for metrics about the function across all calls, and many other options [56].

Scalasca

The built-in analysis capabilities in Scalasca are focused on identifying performance bottlenecks, but the main focus is specifically on identifying wait states introduced by communication and synchronization [49]. This wait state analysis looks for events such

as an MPI_Send operation that occurred much later than it should, or a thread that got to an OpenMP barrier late.

HPCToolkit

Once data has been gathered through HPCToolkit's *hpcrun* and *hpcstruct* tools, it can be processed through the analysis tools within HPCToolkit. The first step is to combine the output from the two previous tools using the *hpcprof* tool, which can form a link between the performance measurements and the source code and then creates a database of this combined performance data.

Once this database has been created, HPCToolkit provides two additional tools to visualize and analyse the data called *hpcviewer* and *hpctraceviewer*. With *hpcviewer*, the focus is on providing code-focused analysis by providing insight into both hotspots in the code and potential bottlenecks [17]. The *hpctraceviewer* tool is designed to present the data in a timeline format where the information is organized by the hierarchy of the parallelism of the original execution.

2.1.3 MPI-Specific Tools

MPI Standard Tools

Since its inception, the MPI Standard has included a profiling interface, called *PMPI* [20]. The PMPI interface has been one of the most common methods for profiling MPI applications for a long time since it provides a simple interface for preempting MPI functions. The idea is that all MPI functions are available with an alternate naming starting with 'PMPI_' instead of 'MPI_', and a profiling tool can simply write their own MPI_X function and then call the equivalent PMPI_X function within their MPI_X function surrounded by any profiling code the tool wishes to add. Unfortunately, since this is simply using a naming shift, only the first library linked into the application will be able to overload the MPI_X function. There is a more in-depth discussion of the PMPI interface in Section 3.3.1.

There is a proposed successor to the PMPI interface called the *QMPI* interface which is meant to remedy the issue of only one tool being able to overload an MPI function for a

given application [16]. This is done through providing a stack of tool instrumentation code surrounding the PMPI function invocation to allow for multiple tools to add profiling code before and after a given MPI function. There is a more in-depth discussion of the QMPI interface in Section 3.3.2.

The MPI Standard has recently been extended to include the MPI Tool information interface (*MPI_T*), which is meant to provide low-level internal MPI information [20]. This is done through exposing internal MPI implementation variables through two different constructs: control variables and performance variables. Control variables are meant to store variables that affect the configuration of the MPI implementation such as the data sizes used for different internal protocols. Performance variables are more focused on internal performance metrics of the MPI implementation such as the number of messages that arrived unexpectedly. There is a more in-depth discussion of the MPI_T interface in Section 3.3.3.

MPI_T Introspection

One of the intended use cases of the MPI_T interface is to use the MPI_T performance variables for introspection of the operation of the MPI implementation and then use that information to inform dynamic updates to the MPI_T control variables. Ramesh et. al. [48] provide an infrastructure for performing such introspection in the MVAPICH2 implementation of the MPI Standard using TAU and the Backplane for Event and Control Notification (*BEACON*), which comes from the Argo project [51] [45]. This work uses the integration of TAU and BEACON to conduct online monitoring of MPI_T performance variables and implements TAU plugin extensions to adjust the MPI_T control variables at runtime to improve performance.

mpiP

The *mpiP* tool specifically targets MPI applications and uses the PMPI interface to instrument all MPI functions in the application [60]. The mpiP tool counts and times all MPI function calls and then performs a stack trace in order to associate each MPI call with its call site in the application. This allows MPI application developers to identify hot spots

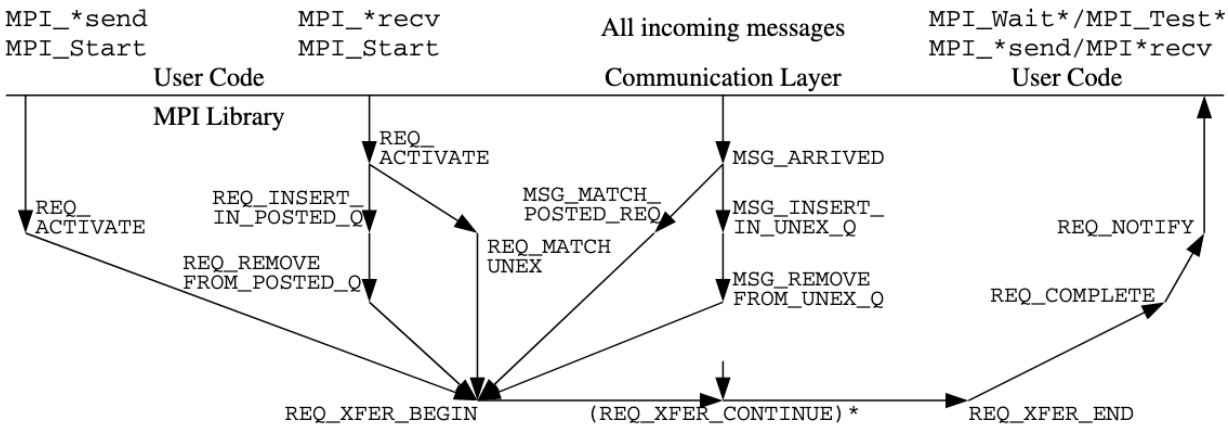


Figure 2.1: A state machine provided by Keller et. al. in their introductory papery for the Peruse interface [31]. This shows a potential sequence of events in the Peruse interface in Open MPI.

in their code from an MPI perspective, by not only showing that they had an issue with a particular function taking a lot of time, but which particular invocation of that function.

Peruse and the MPI_T Events Interface

Before the MPI_T interface was added to the MPI Standard, there was a proposed addition called the *Peruse* interface [31]. Essentially, Peruse was meant to expose internal MPI performance information through instrumentation of the MPI implementation with hooks to function callbacks. The idea was that whenever certain events happened within the MPI implementation a function callback would be triggered to keep track to the current MPI state. This detailed state information could provide a road map of sorts for the execution of MPI functions as shown in Figure 2.1. This interface was not accepted into the MPI Standard but is still in use by the Open MPI implementation.

The *MPI_T events* interface has been proposed as an addition to the MPI Standard and is something of a spiritual successor to the Peruse interface [26]. The idea of the MPI_T events interface is to provide a way for tools to access information about asynchronous events that happen within an MPI implementation through function callbacks. Essentially, performance tools would be able to query possible events, and to attach to those events to be notified when that event occurs.

2.1.4 GPU Performance Analysis Tools

In this work, I am focusing specifically on NVIDIA GPUs, so I will be looking at tools that support NVIDIA's CUDA programming environment for their GPUs. There are many tools that have support for profiling NVIDIA GPUs, some directly from NVIDIA, and others from third party developers. NVIDIA has been releasing more tool support over the years, however the information available on these GPUs is often limited, partially due to the limited monitoring resources available on the GPU hardware.

NVIDIA Tools

NVIDIA provides several tools to assist with performance analysis of CUDA applications such as *nvprof*, Nsight, NVIDIA Visual Profiler, and the CUDA Profiling Tools Interface (*CUPTI*). The *nvprof* tool is a command line utility that allows users to run a CUDA-based application with automatic instrumentation. In addition to simple timing information, *nvprof* can also collect a wide variety of different metrics from the GPU for each CUDA kernel. Since there are limited registers on the GPU for storing this profiling information, each kernel may be run multiple times with different metrics enabled in order to collect all of the enabled metrics [39].

At the end of the program's execution, *nvprof* dumps a profile file that can then be displayed by the Visual Profiler. The Visual Profiler provides a timeline view of the application with the capability to expand each GPU kernel and investigate its properties and any metrics collected for that kernel. In addition, the Visual Profiler can create an analysis report that attempts to identify potential bottlenecks or inefficiencies in each of the CUDA kernels and provides suggestions for improvement. The Nsight development environment combines these features by providing the capability to develop and run CUDA applications with the option to profile and visualize those applications.

When it comes to support for third-party tools, NVIDIA provides the CUPTI interface. Essentially, CUPTI provides an API that allows tools to access various information about the CPU and GPU usage throughout an execution such as timing and usage of CUDA API functions and GPU metrics like instruction and memory access counts all with consistent

timestamps across CPU and GPU events. There are also CUPTI utilities that provide analysis of the collected data and attempt to identify potential bottlenecks.

Tools With CUDA Support

The TAU, Vampir, Caliper, and HPCToolkit tools provide access to CUDA profiling information through the CUPTI interface. They all provide robust timing and metric information of the GPU usage seamlessly with any other instrumentation they have added to the code. The PAPI tool provides a CUDA component that specifically focuses on the hardware counters available on the GPU through CUPTI. These counters are exposed through the standard PAPI API.

2.2 Load Imbalance

2.2.1 Distributed Load Imbalance

The ability to identify load imbalance on a distributed system is a critical need for distributed application development and optimization. There are several tools that can help identify this load imbalance such as Scalasca and HPCToolkit.

Scalasca helps identify load imbalance by analyzing traces to identify inefficiencies that could lead to a load imbalance such as late senders or receivers [23]. This analysis is conducted through extensive trace replays. HPCToolkit uses a technique called blame shifting introduced by Tallent et. al. [55] to shift blame for inefficiencies to their proper source [1]. In other work by Tallent et. al. [54], this concept of blame shifting is applied to call path profiles specifically for identifying the particular point in a program where load imbalance was introduced.

2.2.2 GPU Load Imbalance

Load imbalance studies including GPUs require the help of performance profiling tools that provide access to GPU information that is relevant to imbalance. I have discussed a number

of different tools that are capable of providing GPU performance information in Section 2.1.4, however there is another tool that provides analysis of load imbalance called CASITA.

The CASITA tool provides the capability to conduct root-cause and critical-path analysis on heterogeneous applications through event traces [50]. In addition, CASITA contains a *critical blame* metric that applies a blame shifting technique to both CPU and GPU events on the critical path that cause idleness in future synchronization operations, and thus indicate a potential load imbalance.

Chabbi et al. [10] use HPCToolkit [1] to create a sampling-based approach for conducting performance analysis on GPU-based applications. This is done through the usage of a blame-shifting technique to match sources of idleness or wait states, such as those caused by device synchronization, to the CUDA kernels that caused them rather than the CPU. This allows for a more fair analysis of the influence that CUDA kernels have on the application.

Farooqui et al. [19] add the capability to apply instrumentation procedures to PTX modules in the GPU version of Ocelot to report profiling information at run time. This extended version of GPU Ocelot is used to assess the load imbalance of several CUDA applications by counting the number of cycles executed per SM. The number of cycles executed per SM can be used to calculate a metric similar to *sm_efficiency* from *nvprof*, but not a measure of inter-SM load imbalance.

The blame-shifting techniques used in HPCToolkit and CASITA allow for identifying potential load imbalance between MPI processes introduced at the GPU kernel level. The cycles per SM metric from Farooqui et al. goes a little deeper by providing a look at the internal GPU usage. These two approaches each provide a portion of the picture of how the GPU influences load imbalance, one from a high level and one from a lower level. In Chapter 4, I explore a novel GPU load imbalance metric that combines internal GPU metrics and high-level timing information.

Chapter 3

MPI Performance Analysis and Tool Support Through Software-based Performance Counters

3.1 Chapter Overview

This chapter provides a discussion of my work on MPI performance analysis with a particular focus on providing tool support in the form of low-level performance metrics. I introduce Software-based Performance Counters (SPCs), which are designed to expose internal performance metrics in the Open MPI implementation of the MPI standard. I use these SPCs as a vehicle for furthering my study of the topic of MPI performance analysis.

I will start by providing an introduction of this topic in Section 3.2. This will be followed by a discussion of the relevant background information in Section 3.3. Section 3.4 will elaborate on the motivation for the creation of SPCs. Section 3.5 will explain what sorts of metrics are exposed by SPCs. Section 3.6 will provide a detailed discussion of the design and implementation of SPCs. Section 3.7 will provide analysis of the overhead cost of SPCs. Section 3.8 will show the different methods available for reporting SPCs to users and tools. Section 3.9 will provide results from usage of SPCs for performance analysis in several different use cases. I will then conclude this work in Section 3.10.

3.2 Introduction

With the collapse of Dennard scaling around 2006 [14], chip manufacturers have increasingly relied on multi-core processors in order to improve performance. This trend is reflected on the Top500 list of supercomputers, which shows that on the November 2001 list, 100% of the machines had only 1 processing core per socket, whereas on the November 2019 list, none of the machines had 1 processing core per socket, and the most common configuration was 20 cores per socket with 178 machines [53]. With Moore's Law still in effect (though progress is slowing), the number of cores per socket is expected to continue to increase, at least in the near future [37]. With this increase in parallelism, it has never been more important to have tools for performance analysis, particularly in large distributed systems which can have hundreds of thousands of cores.

3.3 Background

The Message Passing Interface (MPI) is the primary paradigm for writing parallel programs in large distributed memory systems. As such, much of the performance analysis and tool support for these systems is focused on MPI. Since its inception, the MPI Forum has made sure the MPI Standard includes support for native performance analysis of MPI [20]. This started with the MPI Profiling Interface, or PMPI Interface, which allows for high-level profiling of MPI through use of name-shifted MPI functions (more information in Section 3.3.1). The second major performance analysis support added to the MPI Standard was the MPI Tool Information Interface, or MPI_T Interface, which allows for MPI implementations to expose internal variables as MPI_T control and performance variables (more information in Section 3.3.3). There are many implementations of the MPI Standard with varying degrees of compliance to the MPI Standard, each with their own design philosophies (more information in Section 3.3.4).

3.3.1 The MPI Profiling Interface (PMPI)

The MPI Standard defines hundreds of functions with the prefix 'MPI_', for functionalities ranging from sending and receiving data to timer functions. The MPI Profiling Interface provides a means for tools to intercept calls to these 'MPI_' functions and add their own functionality. This works by providing a name-shifted version of the MPI functions that use 'PMPI_' instead of 'MPI_'. The idea is that the tool will be linked into the application before the MPI library, so the tool's MPI_X function is called instead of the MPI implementation's MPI_X function. This allows the tool to add profiling information surrounding a call to the name-shifted version of the function PMPI_X, or to provide their own implementation of the MPI_X function.

A typical usage of the PMPI interface for profiling an MPI function might look like the following:

```
1  /* Global value declarations in the tool */
2  static long long mpi_send_count = 0;
3  static long long mpi_send_bytes = 0;
4  static double mpi_send_time = 0.0;
5
6  /* Tool's implementation of MPI_Send */
7  int MPI_Send(const void* buffer, int count, MPI_Datatype datatype,
8               int dest, int tag, MPI_Comm comm)
9  {
10     double start_time = MPI_Wtime();
11     /* Call the PMPI_Send to perform the actual MPI_Send operation */
12     int return_value = PMPI_Send(buffer, count, datatype, dest, tag, comm);
13     /* Update the global mpi_send_time with calculated duration */
14     mpi_send_time += MPI_Wtime() - start_time;
15
16     int type_size;
17     MPI_Type_size(datatype, &type_size); /* Calculate datatype size */
18     /* Update the global mpi_send_bytes */
```

```
    mpi_send_bytes += type_size * count; /* Calculate the bytes sent */
    /* Update the global mpi_send_count */
    mpi_send_count += 1;

    /* Use the return value from the PMPI call */
    return return_value;
}
```

Listing 3.1: An example implementation using the PMPI interface to profile the MPI_Send function.

The PMPI interface is designed to be MPI implementation agnostic, so tool developers can collect performance information without access to the MPI implementation's source code[20]. This design principle makes it so that only high-level information such as counts of MPI function calls, timing information, and call-site information can be collected through the PMPI interface. Another major drawback of the PMPI interface is that the name-shift design only allows a single tool to preempt a given function for an application. Essentially, whichever tool gets linked first will get the first opportunity to have its overloaded functions used. This problem has been addressed by a new potential addition to the MPI Standard, the QMPI interface [16].

3.3.2 The QMPI Interface

The QMPI interface has been proposed as a potential successor to the PMPI interface, with its name intended to evoke this idea as the 'Q' was chosen because it comes after 'P' in the alphabet [16]. The QMPI interface was designed to allow for multiple tools to use the interface at once, which address the biggest weakness of the PMPI interface. There were several requirements proposed for the QMPI interface: it must support all PMPI tool functionality; it must allow for implementing both pre-processing and post-processing steps; it must allow for replacing MPI function call functionality; it must allow for several tools to be in use at once; the set of tools and the order in which they are used must be configurable at both the user and system level; and the interface itself must have low overhead [16].

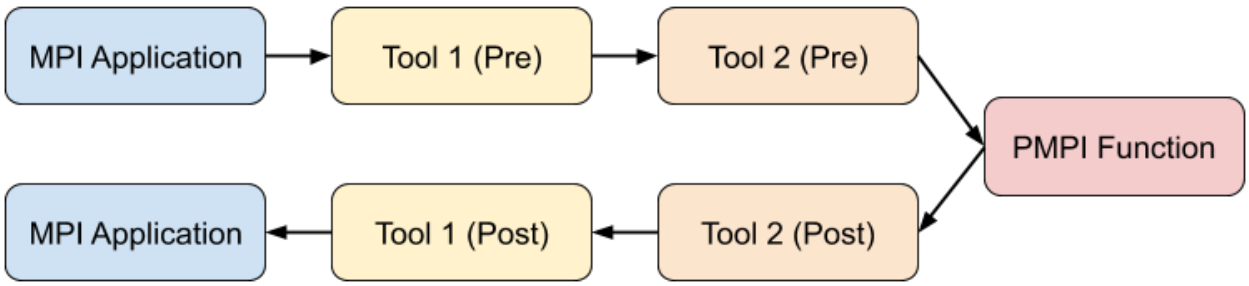


Figure 3.1: An example control flow of the hierarchical tool wrappers with pre-processing and post-processing capabilities in the QMPI interface.

The main concept of the QMPI interface that allows it to support many concurrent tools is that it creates a hierarchy of wrappers to surround the PMPI function call. This way, each tool can have its own pre-processing and post-processing step, while still allowing for many tools. Figure 3.1 shows an example control flow of a usage of the QMPI interface, where two tools were loaded and both of those tools had a pre-processing and post-processing step. With this design, the QMPI interface is much more powerful than the original PMPI interface, though it still only allows for high-level profiling of the MPI library. The QMPI interface will likely be included in the next version of the MPI Standard.

3.3.3 The MPI Tool Information Interface (MPI_T)

With the PMPI interface covering the high-level profiling capabilities of MPI, the MPI Forum decided to include the MPI_T interface as a standardized way to provide low-level tool support in MPI. Since the MPI_T interface focuses on internal information, much of the specifics of what is exposed is necessarily left up to the MPI implementations. The MPI_T interface does provide some guidance through its design in that there are two broad categories of variables that can be exposed: control variables and performance variables.

Control Variables

MPI_T control variables are meant to contain configuration settings and other properties of the MPI implementation [20]. Most MPI implementations already had some way of

getting and setting custom configuration settings, whether that was through environment variables, configuration files, or some other interface, however they are typically not easily accessible to the user within an application. The introduction of MPI_T control variables provides a consistent interface for accessing these MPI properties at runtime that is MPI implementation agnostic. These control variables can also be registered to be modifiable so these settings can be adjusted throughout a run, which can be helpful for tasks like dynamic performance optimization. Many MPI implementations use some form of internal breakpoints for determining which protocols to use for completing user requests. Some common examples in major MPI implementations would be things like the *eager limit* which determines the upper bound for using an eager protocol (sending the payload along with the header), or ranges for message or communicator sizes within which to use different algorithms for collective communications.

Performance Variables

Unlike MPI_T control variables, which are meant to expose guidelines for how the MPI implementation *can* operate, MPI_T performance variables are meant to provide insight into how the MPI implementation *is* operating. As such, performance variables are often MPI implementation internals that provide information on the state of the implementation both at the current moment and earlier in the run. This could be information like what the implementation is currently doing (or not doing), aggregations of data such as number of bytes sent eagerly, which protocols and algorithms are being used, current and previous internal queue usage, etc...

For many MPI implementations, the infrastructure to expose this performance information was not as well developed as it was for control variables. This has led to slower adoption of performance variables by MPI implementations. Since these metrics would typically be related to performance-critical portions of the implementation, special considerations need to be taken to avoid excess overhead in tracking this information. Thus, MPI implementations often opt to have MPI_T performance variables disabled by default to avoid the overhead cost.

MPI_T Usage

The usage of the MPI_T interface requires the user or tool to initialize and finalize the MPI_T interface and specify which variables they are interested in tracking through several MPI_T interface functions. There are slightly different process for accessing MPI_T control and performance variables as well. The initialization and finalization of the MPI_T interface is done through the *MPI_T_init_thread* and *MPI_T_finalize* functions respectively. No MPI_T functions can be called on a given process until the initialization is complete.

Once the MPI_T interface has been initialized, the typical usage of the interface boils down to looping through the available variables, getting their information, choosing the variables of interest, setting up context and handle information for the variables, reading and writing the variables, and then freeing contexts and handles for the variables. This process starts with querying the number of available control or performance variables with the *MPI_T_[cvar/pvar]_get_num* function. The user can then loop over the number of available control or performance variables with indices from 0 until the number of counters, and use the *MPI_T_[cvar/pvar]_get_info* function, which provides information such as the variable name, verbosity, datatype, description, binding, and scope. If the user knows the name of the variable already, they can also use the *MPI_T_[cvar/pvar]_get_index* function to get the appropriate index for that variable which can be used to query the variable's information.

With the appropriate metadata for the variables of interest, the user must then allocate a handle for each variable with the *MPI_T_[cvar/pvar]_handle_alloc* function. This handle will bind the variable to an appropriate MPI object provided by the user, unless the variable was registered by the MPI implementation as MPI_T_BIND_NO_OBJECT. The handle allocation function will also provide the count of the number of elements the variable has of its datatype so users can allocate appropriately sized buffers to store a copy of the variables. For control variables, this is all of the preparation that is required in order to access the variables, so the *MPI_T_cvar_[read/write]* functions can be used to access the variables.

For performance variables, there is a bit more information that is required in order to access the variables. Performance variables are separated into a variety of classes, which dictate their general behavior such as MPI_T_PVAR_CLASS_STATE which represents

discrete states in the MPI implementation, or MPI_T_PVAR_CLASS_COUNTER which counts the instances of a given event. So, the *MPI_T_pvar_get_info* function also provides the variable class information for performance variables, and the *MPI_T_pvar_get_index* function requires the class information to return an index. The scope information is also more detailed for performance variables and is broken down into flags for whether the variable is read-only, is continuously active or can be started and stopped, and is capable of being updated atomically.

The concept of sessions is also introduced for performance variables. Sessions are meant to provide an isolated context within which performance variables are accessed in order to avoid collisions in accesses by different tools [20]. This introduces the *MPI_T_pvar_session_[create/free]* functions for managing contexts for performance variables. Sessions must be used when allocating handles for, and accessing, performance variables.

With all of the different classes of performance variables, there are considerably more ways to interact with the variables than just reading and writing them. If the variable is not continuous, the *MPI_T_pvar_[start/stop]* functions can be used to control their operation. If the variable is not read-only, *MPI_T_pvar_[reset/readreset]* functions can be used to reset the value of the variable to its initial value.

With all of this functionality, the MPI_T interface is very powerful and allows for well-defined variables that makes it easier for users to understand what they are working with and generally how that information is updated. This does have the unfortunate drawback of requiring a lot of API calls in order to register and use each variable.

3.3.4 MPI Implementations

When it comes to MPI implementations, there are essentially two base implementations: Open MPI and MPICH. Both of these implementations are open source, and most other implementations are based on one of these two. The other major distinction between the various MPI implementations is open source versus closed source. The open source implementations such as Open MPI, MPICH, and MVAPICH tend to be collaborations between academia, government, and private companies, whereas most of the closed source implementations come from supercomputer vendors and are optimized for their machines

such as Intel MPI (MPICH-based), Cray MPICH, and IBM Spectrum MPI (Open MPI-based).

For my work, I have chosen to develop my code within the Open MPI implementation because The University of Tennessee Knoxville (UTK) is one of the founding members of Open MPI, and there is a lot of expertise in Open MPI available in the Innovative Computing Laboratory at UTK.

Open MPI

In order to allow for a wide range of different functionalities, the Open MPI implementation is based on the Modular Component Architecture (MCA), which is designed to allow for development of several well contained components that make extending Open MPI easier, and for run-time decisions of which components to use [61] [22]. By default, Open MPI makes decisions on which components to use during an application, however users can control which components are loaded and some of the properties of how those components operate through MCA parameters, typically supplied on the command line or in an MCA parameter file.

The standard stack of components used for communication starts with MPI at the top level; the Point-to-Point management layer (PML) below that; the BTL management layer (BML) below that; and the byte transfer layer (BTL) at the lowest level. When a user calls an MPI communication function, Open MPI transfers control to the PML. The PML uses the BML to determine the appropriate BTL implementation for a particular transfer, and then the BTL handles the hardware transfers of data between MPI processes.

3.4 Motivation

When profiling MPI applications, it can often be difficult to tell what is causing performance issues, particularly when the problem lies within MPI itself. There are many factors that can affect the performance of an MPI implementation, such as management of internal queues, algorithms used for collective communications, and transport protocols used.

MPI implementations often have to deal with data that is received under sub-optimal circumstances such as with unexpected and out-of-sequence messages. An unexpected

message is one that arrived before the corresponding receive was posted. It is well known that searching the unexpected message queue can quickly become a bottleneck, particularly when there are a large number of messages in the queue [58]. Out-of-sequence messages are messages that were delivered out of the MPI-imposed order. MPI is expected to deliver the messages in first-in first-out (FIFO) order between each pair of processes within a communicator. Messages can be delivered out of the proper sequence due to multiple network paths between the processes and potentially because of how the transport software is written. These out of sequence messages block the matching queue on the target process, as all matching must be delayed until the FIFO order can be guaranteed.

There is a need for a tool that would be able to report such internal MPI information to users and tools alike, to provide a more precise picture of what particular conditions could have affected the application performance. Such a tool has the potential to be generic enough to be of use not only to MPI users, but also to be particularly useful to those who are developing an MPI implementation. Having metrics on the internal MPI behavior can help identify bugs and inefficiencies in the implementation, and correct performance critical bottlenecks before they impact production-level scientific applications. One active area of MPI development is in implementing efficient multi-threaded MPI communication, which requires extra care to enforce thread safety and ensure that messages are received in the order they were sent[4]. Being able to easily access internal metrics like when data transfers are initiated or when a message arrives out of sequence can help decrease the burden on multi-threaded MPI developers by giving an explanation for the performance they are seeing. Thus, I implemented Software-based Performance Counters (SPCs) as a means to provide such capabilities.

A crucial benefit to having these metrics internal to the MPI implementation is that they can be exposed without using the PMPI interface. The PMPI interface works by preempting MPI functions, while SPCs work through instrumentation of Open MPI code and do not need to interfere with this preemption. Many existing MPI tools use the PMPI interface to perform their profiling, so keeping this interface free allows those tools to be used concurrently with SPCs. Users can also leverage the PMPI interface for supplementing MPI functions with their own code.

These SPCs are modeled after *PAPI*'s hardware counters due to their simplicity and familiarity within the HPC community. The idea is to have a similar system for exposing low level information as *PAPI* does, but for software-based events, specific to MPI, rather than hardware events. It is worth noting that *PAPI* now has an interface for software-defined events that allows for libraries to define their own events to be exposed through *PAPI* [11].

3.5 Performance Metrics Exposed

3.5.1 Types of Events

When thinking about what sorts of metrics to create, it is helpful to first think about what types of events can happen and what information about those events is relevant to performance analysis. There are many events that can happen in an MPI implementation. I have decided to focus on events related to the transmission process of messages since these tend to be the most performance sensitive events. I have distilled these events into five major categories: cumulative, state, temporal, watermark, and categorical.

Cumulative Events

Cumulative events are those that derive their importance from the number of times they occurred. There are many potential use cases for cumulative events such as keeping track of event counts to allow for calculating simple statistics and identifying proportions of events with positive and negative performance impact. For example, imagine a scenario where there are two cumulative events, A and B. Event A represents a program executing the fast path, and event B represents the program executing a sub-optimal path. In this example, the proportion of the counts of events A and B could be useful in informing performance optimization efforts. Cumulative events are tracked in SPCs through *regular counters*. Regular counters are represented by an integer and can be updated both positively and negatively by an event.

State Events

State events are events that are relevant to the current program state. In this context, program state could refer to a number of things such as the current values stored in memory, or what the program is currently doing in a broad sense. As discussed in Section 3.6.1, SPCs are stored as *long long* integer values, and do not have contextual information such as timestamps associated with them. For this reason, it is difficult to create SPCs that are specifically representative of the current program state, and there is no specialty counter type for state events.

With no specialized state counters, the state of MPI must be derived through a combination of a variety of different counters taken as a snapshot at a particular point in the application. Many SPCs are monotonically increasing, and are not representative of the current program state, however some *regular counters* can provide some insight, such as the current number of messages in internal queues or the amount of data currently allocated for internal queues.

Temporal Events

For temporal events, the time at which they happened, or their duration are the area of interest. These could be the amount of time spent matching messages or time spent waiting at a barrier for example. These temporal events are represented by *timer counters*, so only temporal events that can be represented by a cumulative duration are available at this time. SPC timer counters use a cycle-precision timer under the hood by default to keep track of the duration of temporal events within Open MPI.

Watermark Events

For some types of events, it is important that they are pushing the boundary of a maximum or minimum value. I refer to this type of events as *watermark events*. These events are represented by a special type of SPCs called *watermark counters*. Watermark counters are paired with a regular counter, called a sentinel value, and are updated whenever the sentinel value exceeds the current watermark. At this time, only high watermarks are currently

supported. An example of a high watermark could be the maximum number of items stored in a queue. High watermark SPCs are reset to the current sentinel value whenever they are read through the MPI_T interface.

Categorical Events

There are some events where it is important to qualify the event in some way to provide a distinction between this event and similar events. For instance, when MPI performs a broadcast, this is important information, but it is also important which algorithm was used to perform the broadcast and what the parameters of the broadcast were, such as the size of the communicator or the message size. In order to keep track of such events, I have created *bin counters*. Bin counters operate by having a top-level counter that keeps track of how many total times the counter was updated as well as a series of bins that represent subcategories of the event. Each bin counter has a series of rules associated with it that determine the circumstances under which a particular bin is updated. For example, a bin counter could be created for the MPI_Send function that has two bins, one for messages less than or equal to 1000 bytes in size, and one for messages greater than 1000 bytes. So, if MPI_Send was called twice with message sizes of 1 and 10000 bytes, the top-level counter would be 2 and each bin would have a value of 1.

I have also created a special subcategory of bin counters called *collective bin counters*, which are specifically for keeping track of the context surrounding collective algorithm usage. The idea is that for each collective algorithm there are four bins arranged in a 2×2 grid. The rows of the grid represent message size, small or large, and the columns represent communicator size, small or large. The breakpoints for small and large communicators and message sizes are MCA parameters for user tuning.

3.5.2 SPC Metrics

I implemented a variety of counters that expose information from two different levels within the Open MPI stack. The first level I added counters to is the MPI layer. Some examples of these counters can be seen in Table 3.1 (full list in Appendix A). In this layer, the counter

Table 3.1: Some examples of SPCs from the MPI and PML levels of the Open MPI codebase.

MPI Level	PML Level
OMPI_SPC_SEND	OMPI_SPC_BYTES_RECEIVED_USER
OMPI_SPC_RECV	OMPI_SPC_BYTES_RECEIVED_MPI
OMPI_SPC_ISEND	OMPI_SPC_BYTES_SENT_USER
OMPI_SPC_IRECV	OMPI_SPC_BYTES_SENT_MPI
OMPI_SPC_BCAST	OMPI_SPC_BYTES_PUT
OMPI_SPC_REDUCE	OMPI_SPC_BYTES_GET
OMPI_SPC_ALLREDUCE	OMPI_SPC_UNEXPECTED
OMPI_SPC_SCATTER	OMPI_SPC_OUT_OF_SEQUENCE
OMPI_SPC_GATHER	OMPI_SPC_MATCH_TIME
OMPI_SPC_ALLTOALL	OMPI_SPC_OOS_MATCH_TIME
OMPI_SPC_ALLGATHER	

values fall into two categories: those that count how many times each of the user-level MPI functions has been called, and those that keep track of collective algorithm usage.

The MPI function call information is useful for showing an overview of the types of communications that appear in an MPI application. The information from these counters could have been collected with the PMPI interface, but with SPCs keeping track of these values, a PMPI-based tool would no longer need to count the instances of each MPI function. These values can also be used to provide context to some of the other lower level counter values.

The collective algorithm counters are more low-level than the basic MPI function counters and provide insight into how Open MPI is performing a given collective. In Open MPI there are many algorithms for performing each collective operation, and a decision needs to be made at runtime which algorithm will be used when that collective is called. This decision is typically based on the message size and the communicator size. Thus, I have added a collective bin counter for each collective algorithm available in the base collective component, which is at the MPI level.

The other level I added counters to was the PML layer (see Section 3.3.4). This was done in order to expose more low-level information, particularly about the process of sending and receiving messages. Some examples of counters for this level are shown in Table 3.1 (full

list in Appendix A). These lower level counters focus on things like bytes sent and received, internal queue usage and properties, and usage of eager protocols.

Bytes Sent and Received by the User vs. MPI

The counters for bytes sent and received are split into two subcategories: bytes sent or received by the user, and bytes sent or received by MPI (OMPI SPC BYTES [SENT/RECEIVED] [USER/MPI]). This is an important distinction, because some of the data transmissions performed by MPI are not explicitly requested by the user. For bytes sent and received by the user, the bulk of the values come from explicit point-to-point messages such as MPI_Send and MPI_Recv. For bytes sent and received by MPI, most of the values are from MPI collective operations, which are managed by MPI. This can also include additional data transmitted for the process of data transfer management, data transmitted for MPI internals, topology information detection and exchange, and particular algorithms for communicators, windows, and file creation.

In the Open MPI implementation, messages can be broken down into separate fragments for internal processing depending on their size. This allows for optimizations such as pipelining and allows for more fine-grained control of the transmission of messages. The SPCs for bytes sent/received are updated at the message fragment granularity in that as soon as a fragment is given to or taken from the BTL level, the counters are updated. The aforementioned methodology works well for smaller messages, but the process becomes more complicated with larger messages. Open MPI uses remote direct memory access (RDMA) operations such as *Put* and *Get* operations for large memory transfers (OMPI SPC BYTES [PUT/GET]). These Put and Get operations are handled by the BTL, so rather than add detailed counters to all of the BTL implementations, I decided to simply add Put and Get counters at the PML level and update them when a Put or Get operation is initiated. There may be more time between initiation and when the data is actually transferred, so these counters can be more coarse-grained in their updates than the bytes sent/received counters.

Internal Queue Counters

The primary function of MPI is to move data from one place to another within a system, so it stands to reason that managing that data tends to be one of the more expensive operations within an MPI implementation. In many cases, the MPI implementation can begin moving data right away, but there are some cases where this is not possible and data needs to be stored, often in a queue, for later processing. There are two major cases where this happens in Open MPI on the receiver side: unexpected messages, and out of sequence (OOS) messages.

For both unexpected and out of sequence messages, Open MPI needs to perform a matching process to pair a receive request with its corresponding arrived data. The time it takes to perform this matching process can have a big impact on latency, particularly when there are a large number of messages in the queues. It can therefore be interesting to gain a more precise understanding of the state of these internal queues, and the performance of the matching process.

In order to assess the performance of the matching process, I have broken down the matching process into two parts: attempting to find a match and handling a failure to match. The performance of an attempted match is a factor of how many posted receives there are, and potentially how many messages are in queues such as the OOS message queue. If no match is found, the message is inserted into the unexpected message queue for later processing. To quantify the performance of these two parts of the matching process, I added two SPC timer counters: one for time attempting to match, and one for inserting messages into the unexpected message queue.

When inserting messages into internal queues, there is additional overhead in the form of the additional memory usage required to store the messages. In Open MPI, there is a certain small amount of memory that is allocated for each message request in addition to the size of the metadata for the request. This is to facilitate including the payload of very small messages in this allocated memory to avoid memory allocation overhead. In some cases, the payload exceeds this small buffer, and additional memory must be allocated to store the payload, particularly when using an eager protocol where some, if not all, of the payload is delivered with the request. To quantify this memory overhead for unexpected and OOS

messages, I added SPCs such as the current and maximum amount of memory allocated to store messages (in addition to the memory allocated as part of the request), and the current and maximum amount of payload data in the queues.

Typically, MPI implementations handle unexpected messages by pushing them into a queue that will be checked each time a receive is posted. Since handling of unexpected messages can become a significant bottleneck [58], I have added several SPCs to the unexpected message queue portions of the Open MPI implementation such as: the total number of unexpected messages received, the current number of messages in the unexpected queue, and a high watermark of the number of messages in the unexpected queue. This can give the user an idea of what percentage of messages are arriving unexpectedly, and what the current and worst-case queue search lengths are. If the user or tool keeps track of counter values over time, they can also determine information such as how often unexpected messages are arriving, and which portions of their code are producing more unexpected messages.

One of the core principles laid out in the MPI standard is that messages must be received in the order that they are sent [20]. Open MPI enforces this ordering using a *sequence number* for each message between two MPI processes in the same communicator. In Open MPI, order is sometimes enforced by the BTL implementation (see Section 3.3.4) such as with the TCP BTL, but other BTL implementations such as *openib* for InfiniBand do not necessarily enforce ordering in every use case. The InfiniBand hardware does enforce ordering of the messages, however the *openib* BTL implementation in Open MPI allows for messages to be sent out of sequence when there are messages that failed to send. Essentially, when a message fails to send it is put into a queue for resending later.

There are many ways that a message can be delivered out of the proper sequence, such as race conditions with multithreading, delivering messages with multiple networks, multiple routing paths through the network between endpoints, and more. For example, there was a bug in the *openib* BTL in Open MPI that allowed for messages in the fast path to bypass checking the failed message queue, thus these fast path messages were sent before the failed messages, causing them to be delivered out of sequence in most cases. At the receiver side, posted receives can only be matched once all prior sequence numbered requests have been received. Out of sequence messages can cause a significant bottleneck

due to increased memory management and time spent searching through the queue of out of sequence messages, similar to unexpected messages. To identify this bottleneck, I have added several counters to the out of sequence message handling code in Open MPI, mirroring the counters for unexpected messages with a total count of out of sequence messages as well as current and high watermark values for the out of sequence messages in the queue.

3.6 Design and Implementation

The implementation of SPCs enables users to see useful performance metrics that range from the number of times MPI_Send was called to more detailed internal MPI metrics such as the number of messages that were unexpected or which algorithm was used for an MPI_Bcast. The driver code for SPCs was implemented in the MPI runtime layer within Open MPI which acts as general manager for the operation of Open MPI. The driver code consists of data structures for storing the counter information and functions for managing allocated memory and updating the counters. The instrumentation code for the various counters appears in both the MPI and PML layers, depending on how low level the information is. Some counters are only updated in one location, while others are updated in multiple places to most accurately reflect the metric they represent.

3.6.1 SPC Data Structures

The SPC driver code relies on four main data structures for operation: an enumeration of all SPCs, bitmaps for enabled counters and counter properties, a buffer for storing counter values, and an array of offset structures. There are two primary design principles behind the the SPC data structures: usage of SPCs should incur minimal overhead, and the data exposed to the user should be contained within a single contiguous buffer. The usage of a single contiguous buffer has performance benefits for data accesses by helping to minimize page faults and cache misses. Another benefit, from a design standpoint is that this data can be mapped into shared pages using the *mmap* function. Each MPI process has its own SPC data structures, so the counters are updated separately for each process.

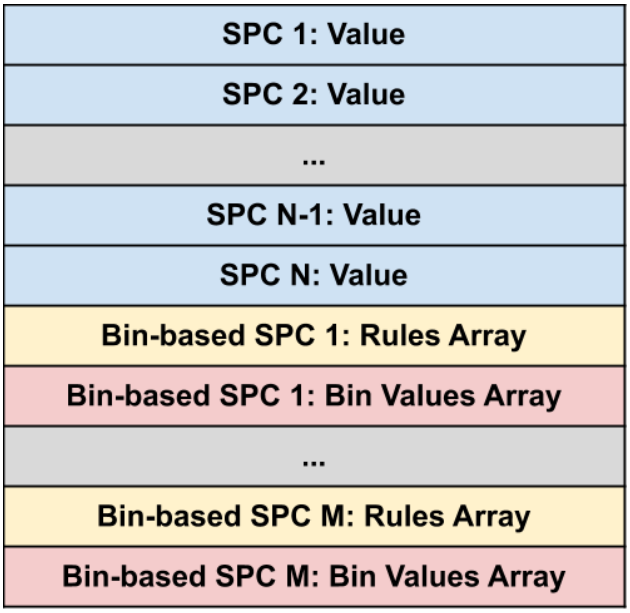


Figure 3.2: The data layout for the contiguous SPC data buffer, with **N** being the number of SPCs, and **M** being the number of bin counters. Note: Each pair of rules and values arrays are cache line aligned in order to avoid false sharing between separate bin counters.

The enumeration of SPC names serves as a method for indexing into most of the other internal SPC data structures and serves as a form of *counter ID*. To minimize overhead, SPC properties such as whether they are enabled and whether they are a special type of counter are stored in bitmaps. The enumeration provides the offset for the appropriate bit in a given bitmap. The bitmap denoting which SPCs are enabled gets particularly heavy usage because it needs to be checked each time Open MPI instrumentation tries to update a counter.

The counter value data can be broken down into three components: base value, rules, and bin values. The rules and bin values are used for storing additional information associated with special *bin counters*. At this time, the counter values are stored as **long long** integer values, so floating point counters are not supported. The rules are stored as **int** type values since they typically don't require particularly large values. The counter values are represented as integer values rather than an array of values to reduce memory overhead. Due to this, there is no context associated with individual updates to a counter such as a timestamp or which portion of the Open MPI code caused the update.

Since all of the counter values, rules, and bin values are stored in one contiguous buffer with multiple data types inside, there is a need to index into this data properly. The data buffer is organized as shown in Figure 3.2, with all of the SPC values contiguously stored at the beginning of the buffer followed by adjacent rules and values arrays for each bin counter. To avoid false sharing between bin counters, each rules array is aligned such that it begins a new cache line. The SPC offsets structure array is added to allow for easy indexing into the proper locations within this buffer for the rules and bin values.

3.6.2 SPC Update Functionality

In order to support all of the different types of counters that are available, there are several helper functions for updating the SPCs. All of these functions are called through macros that become *no-ops* and get optimized out if Open MPI is built without support for SPCs. In order to minimize overhead, these macros are constructed to check whether a particular counter is turned on before calling any functions, so the worst case for a disabled counter is the overhead of an **if** statement.

Since MPI allows for multi-threading within a process, most updates to the SPC data structures are performed using atomic operations. The exception to this is the update for watermark counters, which assumes that the watermark update was performed with a lock already acquired. This is done to avoid having to acquire a lock within the SPC driver code which would incur a huge overhead penalty. At this point, all watermark counters are already within a locked region of the Open MPI code base, so no additional locks were required.

3.7 Overhead of SPCs

3.7.1 Instrumentation Overhead

One of the biggest concerns when implementing SPCs within Open MPI is minimizing the overhead of adding instrumentation throughout the Open MPI code, particularly within the fast path. If there is too much overhead in adding instrumentation, the overhead could sometimes outweigh the benefit of getting the profiling information and would lessen

usage of the instrumentation. For this reason, I took particular interest in making SPC instrumentation as inexpensive as possible.

The overhead added for instrumentation depends primarily on the type of counter, and whether or not the counter is activated. When a counter is activated, the instrumentation can add a variety of operations into the code such as: *if* statements, function calls, bitmap queries, value comparisons, assignments, atomic add operations, add operations, subtraction operations, and cycle timer queries denoted as follows:

Definition 3.1. $T_{if} \rightarrow$ *The cost of an if statement.*

Definition 3.2. $T_{func} \rightarrow$ *The cost of a function call.*

Definition 3.3. $T_{bitmap} \rightarrow$ *The cost of a bitmap query.*

Definition 3.4. $T_{comp} \rightarrow$ *The cost of a value comparison.*

Definition 3.5. $T_{assign} \rightarrow$ *The cost of a value assignment.*

Definition 3.6. $T_{atomic} \rightarrow$ *The cost of an atomic add operation.*

Definition 3.7. $T_{sub} \rightarrow$ *The cost of a subtraction operation.*

Definition 3.8. $T_{cyc} \rightarrow$ *The cost of a cycle timer query.*

These operations can be combined to express the overhead incurred by SPCs in different circumstances. The overhead will be different for counters that are turned on versus turned off, and additional overhead will be incurred by some of the specialized counters like bin counters, timer counters, and watermark counters. Equations 3.1 to 3.5 show the overhead of these scenarios through a combination of the operations listed above in Definitions 3.1 to 3.8. It is worth noting that Equations 3.3 to 3.5 build upon the overhead of Equation 3.2 in that they are additional overhead on top of the counter being turned on.

$$O_{Off} = T_{if} + T_{bitmap} \tag{3.1}$$

$$O_{On} = T_{if} + T_{bitmap} + T_{func} + T_{atomic} \tag{3.2}$$

$$O_{Watermark} = T_{atomic} + T_{if} + T_{bitmap} + T_{if} + T_{comp} + T_{assign} \tag{3.3}$$

The overhead for watermark counters varies depending on the situation. Equation 3.3 assumes that both the watermark counter and the sentinel counter are turned on, otherwise the additional overhead would the same as the overhead of a counter that is turned off, shown in Equation 3.1. If the updated sentinel value is greater than the current watermark, then the assignment overhead (T_{assign}) is incurred.

$$O_{Timer} = T_{if} + T_{bitmap} + (2 \times T_{cyc}) + T_{sub} \tag{3.4}$$

The overhead for timer based SPCs includes two queries to a cycle precision timer for the start and stop times, and as such there is an additional *if* statement overhead associated with these counters since there need to be two checks for whether the counter is turned on (one for start and one for stop).

$$O_{Bin} = \sum_{1}^{NumBins-1} T_{if} + T_{comp} \tag{3.5}$$

The overhead for bin counters varies depending on which bin needs to be updated. Since bin counters have a series of rules denoting which ranges of values go in each bin, the rules for each bin are checked sequentially until the correct bin is found. Equation 3.5 shows the worst case overhead where all of the rules need to be checked. If the value belongs in the final bin, an *if* statement is not required because all of the previous *if* statements have ensured that the update belongs to the final bin, thus the summation from one to the number of bins minus one.

The SPC overhead varies for each MPI function since not all SPCs fall within the code path executed by each function. With the overhead formulas defined in Equations 3.1 to 3.5, I can provide an equation for the overhead added to an MPI function in Equation 3.6.

Table 3.2: Configuration of the testing system, *Arc*.

Property	Arc Configuration
Processor	Dual 10-core Intel Xeon E5-2650 v3 @2.3 Ghz
Interconnect	InfiniBand EDR (100 Gb)
Compiler	gcc 6.3.0
Open MPI	optimized, dynamic build

$$O_{SPC} = \sum_{0}^{Off} O_{Off} + \sum_{0}^{On} O_{On} + \sum_{0}^{TimerOff} O_{Off} + \sum_{0}^{TimerOn} O_{Timer} + \sum_{0}^{WatermarkMiss} O_{Watermark} - T_{assign} + \sum_{0}^{WatermarkHit} O_{Watermark} + \sum_{0}^{Bin} O_{Bin} \tag{3.6}$$

In Equation 3.6, each summation represents the number of counters from a particular class of counters or enabled status that are encountered in the code path of a particular MPI function. For example, an MPI_Send operation might hit SPCs such as the number of times MPI_Send was called, bytes sent by the user, point to point message size (bin counter), and number of eager messages. Assuming all counters are turned on, the overhead added to this MPI_Send operation would be $(4 \times O_{On}) + O_{Bin}$.

3.7.2 Testing the Overhead Cost of SPCs

It is critical to ensure that the overhead imposed by any performance gathering mechanism remains minimal, and its impact on the performance of the underlying MPI functionality is unaffected. To test how much overhead is introduced with these counters, I use the NetPIPE benchmark [52]. This benchmark performs a ping-pong throughput test and reports the bandwidth and latency for a variety of message sizes and repetitions for each message size. In this section I will focus on the latency numbers from NetPIPE because they provide more insight into the overhead of the different counters.

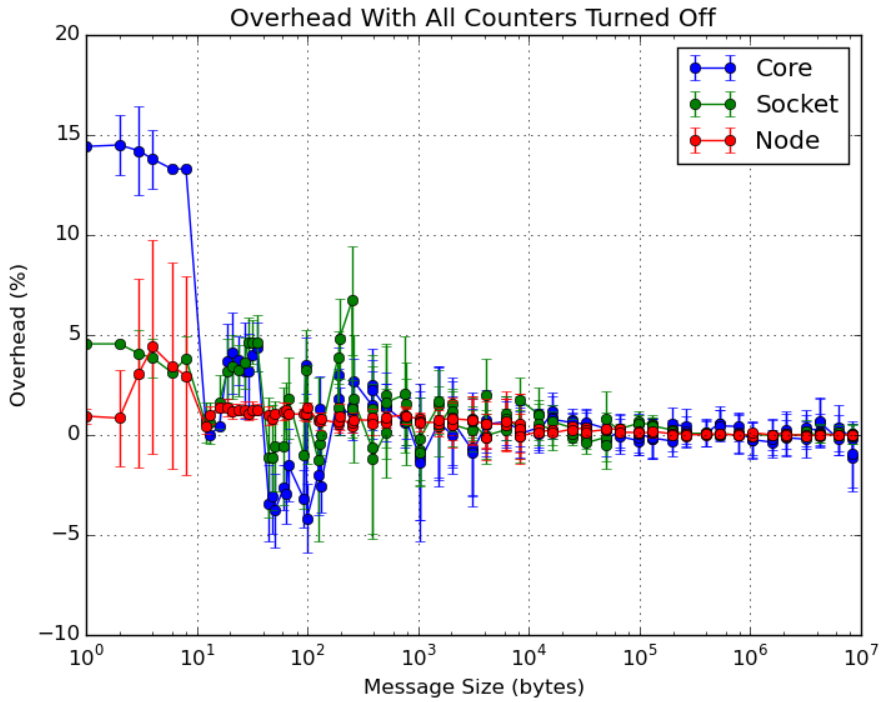


Figure 3.3: The overhead of adding SPCs to the code while leaving all of them turned off. Note: the error bars represent the standard deviation across the 10 test runs.

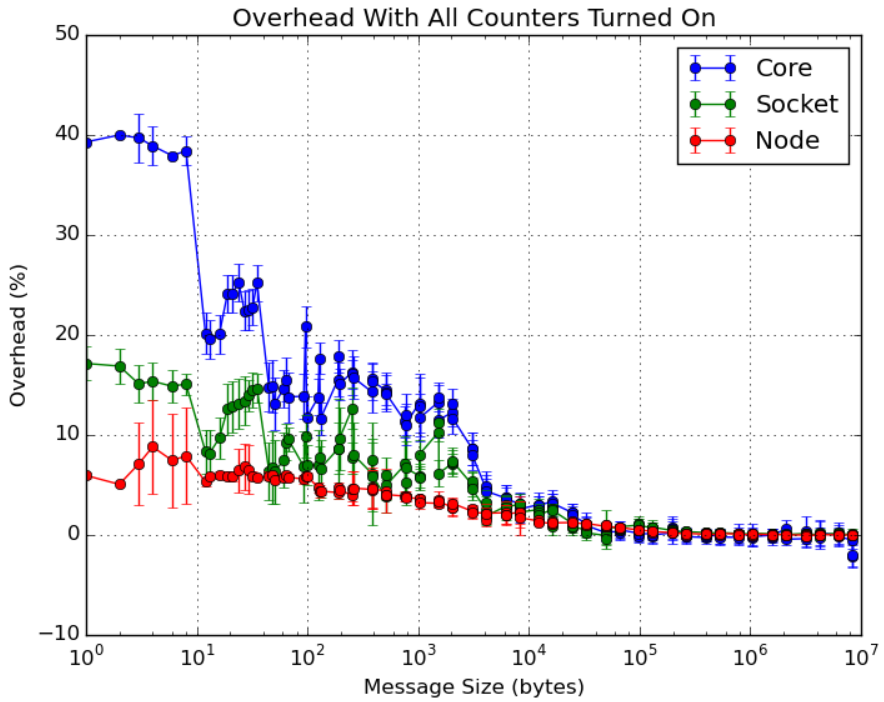


Figure 3.4: The overhead of adding SPCs to the code and turning all of them on. Note: the error bars represent the standard deviation across the 10 test runs.

These tests are performed on the *Arc* machine, the configuration for which can be found in Table 3.2. To test different usage cases, I performed the NetPIPE benchmark with three different configurations with varying degrees of expected impact. The first configuration, *Node*, focuses on inter-node communication over InfiniBand using the *openib* BTL. Here, *node* refers to an individual server within a distributed compute cluster. The next two configurations, *Socket* and *Core*, deal with intra-node communication over shared memory using *SysV* shared memory through the Open MPI *vader* BTL. With the *Socket* test, the MPI processes were bound to cores from different CPU sockets within the same node, and for the *Core* test, the MPI processes were bound to cores within the same socket. Here, *socket* refers to a CPU slot within a compute server. It is worth noting that these tests were conducted with an older version of SPCs which had fewer counters added to the Open MPI codebase, however the overhead is similar in the most recent version of SPCs.

For the baseline test, Open MPI is built with the same set of configuration parameters but without SPCs enabled, which turns all of the code associated with SPCs into no-ops. Next, I performed several tests with SPCs compiled in. The first two tests simply have all of the counters turned off or all of the counters turned on. The overhead of all counters being off shows the impact of the **if** statements added to the different paths in the code (including in some cases to the critical path). Having all of the counters turned on is the worst case for overhead, and the impact will be from both the **if** statements and instructions added to handle the counters (including in most cases atomic add operations). All of the overhead data points are the average of ten runs of NetPIPE to help account for noise in the network. I also present the standard deviation of the non-curated data points, to highlight the best and worst case scenarios.

Figure 3.3 shows the overhead incurred when SPCs are built, but all of them are turned off. This effectively shows the difference in performance if SPCs were to be included in the Open MPI build by default. The overall trend is that the overhead decreases as the message size increases. As expected, the inter-node overhead is the lowest with the overhead for most message sizes being around 1%. For messages between 3 and 8 bytes in length, there is an increase in latency on the *Arc* system. This spike in latency happened infrequently but was more likely to occur when the counters were turned on resulting in over 4% overhead on

Table 3.3: Counters used in the NetPIPE benchmark.

Counter Name
OMPI_SEND
OMPI_RECV
OMPI_BYTES_RECEIVED_USER
OMPI_BYTES_SENT_USER
OMPI_BYTES_GET
OMPI_UNEXPECTED
OMPI_MATCH_TIME

average. The maximum overhead for this test was ~14.5% for small messages sent between cores in the same socket. In most cases the overhead was less than 5%, and for the intra-node tests the latency was actually shorter on average for message sizes around 100 bytes when the counters were built.

Figure 3.4, shows the maximum overhead of using SPCs with NetPIPE, since all of the counters are turned on. There are similar patterns in the plots for the different test cases, simply with higher magnitudes. Again, the *Core* test shows the highest overhead with ~40.0%. The inter-node overhead remains around or below 5% overhead for most message sizes. This result shows that for the majority of cases, adding SPCs does not add a large amount of overhead.

To account for the sizable gap between the overheads of having all counters turned on and off, I performed tests with selected counters turned on. For the NetPIPE benchmark, seven different SPCs are encountered during the run. These counters are shown in Table 3.3. After testing with different counters turned on, I found that the OMPI_MATCH_TIME counter accounts for the majority of the overhead.

Figure 3.5 provides a comparison between having the counters turned on, turned off, only having OMPI_MATCH_TIME turned on, and only having the six counters needed by NetPIPE minus OMPI_MATCH_TIME, for the *Core* test. This figure shows that nearly all of the overhead increase from *all off* to *all on* can be attributed to the OMPI_MATCH_TIME counter. In order to update this counter, I use a timer function to get the start and end times of the matching process. Both starting and stopping the

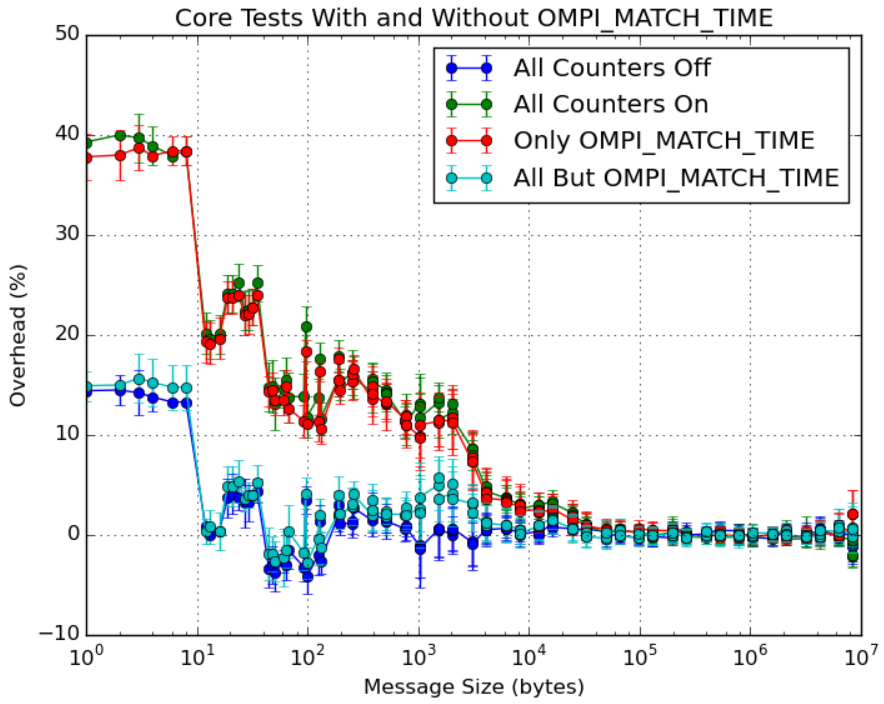


Figure 3.5: Comparing the intra-node overhead within a single socket with the counters all on, all off, only OMPI_MATCH_TIME turned on, or only the counters from Table 3.3 minus OMPI_MATCH_TIME. Note: the error bars represent the standard deviation across the 10 runs.

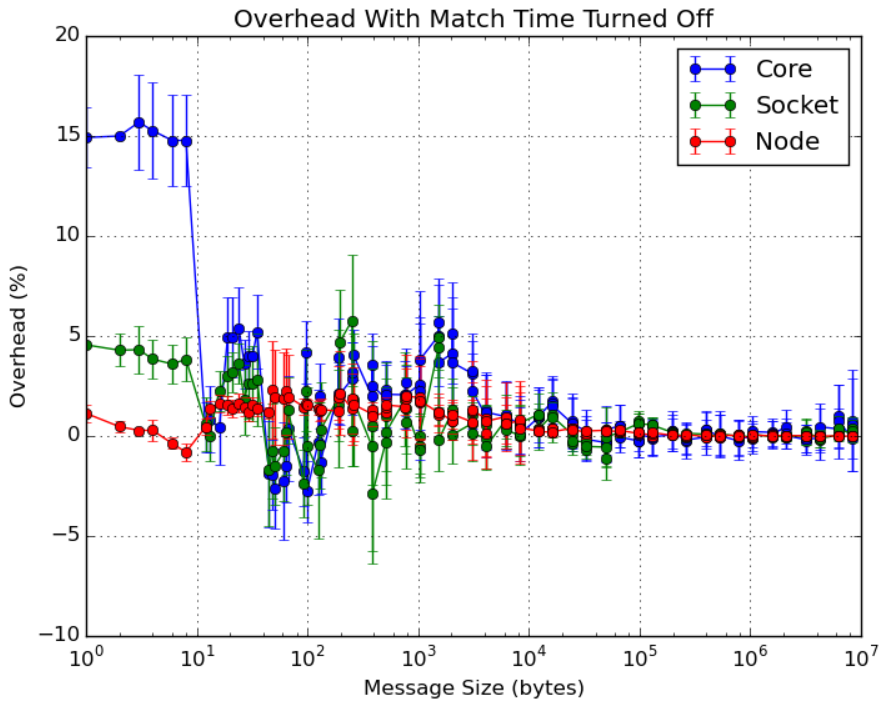


Figure 3.6: The overhead of adding SPCs to the code and turning on only the counters from Table 3.3 minus OMPI_MATCH_TIME. Note: the error bars represent the standard deviation across the 10 runs.

timer require *if* statements to ensure the OMPI_MATCH_TIME counter is turned on in addition to the *if* statement and atomic add operation of the counter update. In total, there are three *if* statements, two timer function calls, one subtraction operation (for calculating elapsed time), and one atomic add operation needed for each match. The other counters used in this test, by comparison, only require a single *if* statement and an atomic add. The OMPI_MATCH_TIME counter can also happen more often than many other counters because the matching process can happen multiple times for a single message if it is unsuccessful. The OMPI_BYTES_RECEIVED/SENT_USER counters can also happen multiple times per message if the message is broken into fragments, yet these counters do not add a significant amount of additional overhead. This suggests that the additional *if* statements and timer function calls are the cause of this increased overhead.

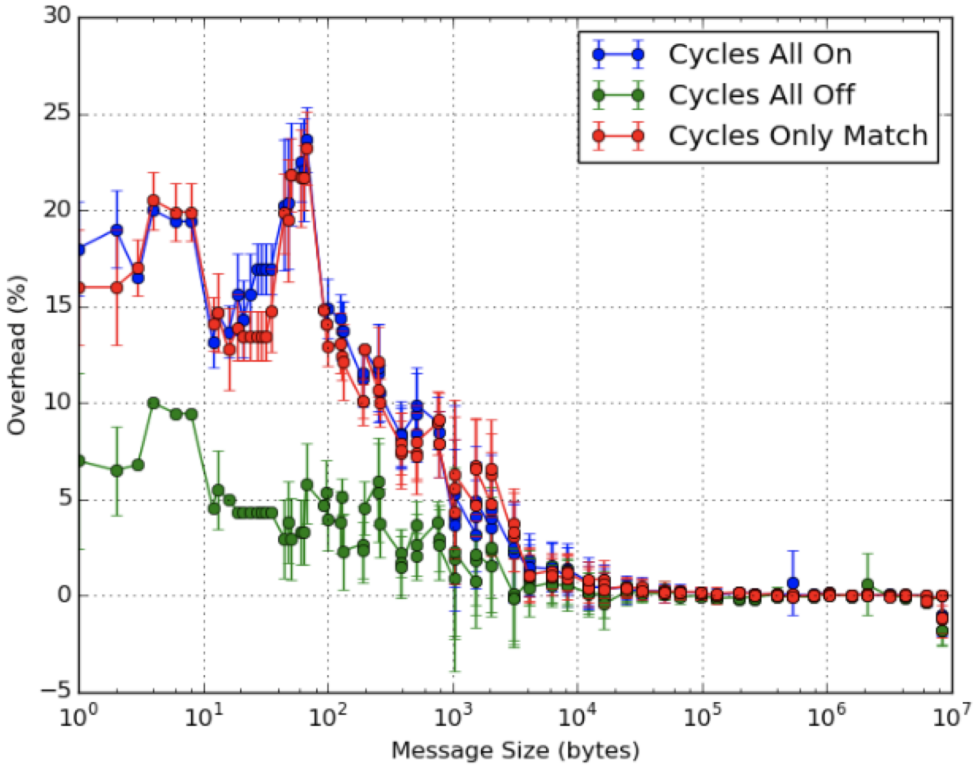


Figure 3.7: The overhead of adding SPCs to the code in the Core case when using cycles instead of converting to microseconds. This also looks only at the counters from Table 3.3 in all on, all off, and all on minus OMPI_MATCH_TIME. Note: the error bars represent the standard deviation across the 10 runs.

The timer function used for these test cases simply calculates the time in microseconds by dividing the monotonic number of cycles returned from the RDTSC instruction by the clock frequency in MHz. This division operation can add a large percentage of time when the latency is already low. For example, in the case where the overhead is 40%, the latency without the SPCs built is ˜200 nanoseconds and the latency with the SPCs built and turned on is ˜280 nanoseconds. This additional overhead of 80 nanoseconds equates to 184 cycles on the *Arc* machine. On this machine, the estimated length of a 32-bit division operation is 35-47 cycles according to the Intel manual, and there are two of them for each time the matching process happens, so for each match these division operations add 70-94 cycles of overhead [27]. To verify that removing the OMPI_MATCH_TIME counter reduces the overhead for all cases, I decided to redo the *Core*, *Socket*, and *Node* tests with this counter turned off. Figure 3.6 shows the overhead of the three tests if the OMPI_MATCH_TIME counter is turned off. For all of the use cases, the performance is nearly the same as having none of the counters turned on.

The bulk of the additional overhead of having counters turned on comes from the timer-based counters due to the more expensive operations in these counters. In order to alleviate this overhead, I reimplemented the timer counters to store the raw timer values in cycles rather than converting them to microseconds each time. This avoids the division operation in the timer counters, which can save a significant amount of overhead. Figure 3.7 shows that the maximum overhead of having all of the SPCs turned on drops from ˜40% to ˜25% in the *Core* test case, which provides the worst case scenario for overhead.

3.7.3 mpiP Overhead

To compare with a similar tool, I looked at the overhead of using the *mpiP* tool which uses the PMPI interface to instrument the code. *mpiP* adds timing information and counters to user-level MPI functions like MPI_Send and MPI_Recv and associates each function call with its calling location in the user code. The idea is to assess the number of times and length of time spent in each function from each location. This tool is similar to SPCs in that it keeps track of the number of times that MPI functions are called, however it provides additional information in the form of timing information for the function calls. Figure 3.8

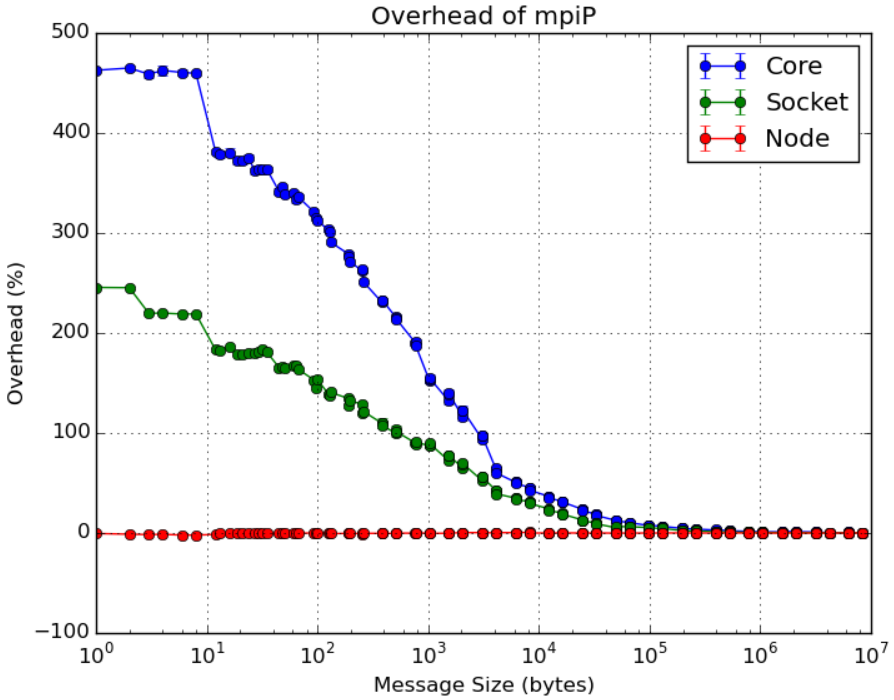


Figure 3.8: The overhead of using mpiP with NetPIPE. Note: the error bars represent the standard deviation across the 10 runs, however the deviation was extremely small so they appear nonexistent.

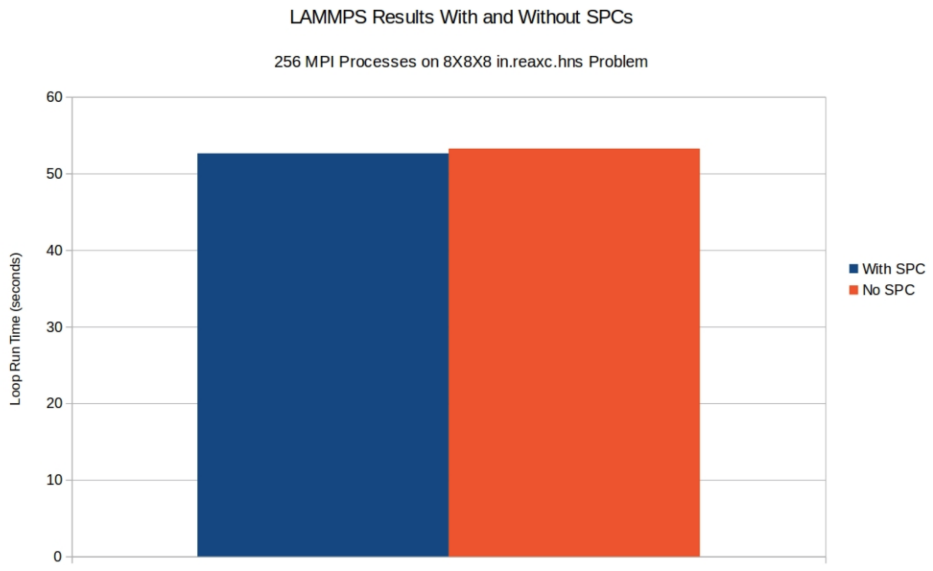


Figure 3.9: The overhead of having all SPCs turned on while running the LAMMPS proxy application.

shows the overhead of running NetPIPE with *mpiP*, using the default settings for *mpiP*. For the *Node* test, *mpiP* performed similarly to the baseline Open MPI test, but the shared memory tests tell a much different story. For the *Socket* test, the overhead of using *mpiP* was up to ~245.6% and for the *Core* test, the overhead was up to ~465%.

The *mpiP* build used PMPI_Wtime for its timing measurements, which redirects to the same low-level timer used for the SPC timer counters. Some of the additional overhead comes from determining the call sites of MPI functions from the call stack and from calling *mpiP* functions. This overhead is particularly apparent for the intra-node tests because the latency for these tests is already as low as hundreds of nanoseconds and simply adding timer functions can have a large impact as seen with the SPC timer-based counters. For the inter-node test, this overhead is largely hidden by the network latency since the baseline latency in these tests are in microseconds.

3.7.4 Proxy Application Overhead

I have shown that adding SPCs can add up to 25% overhead to each MPI call in the worst case and drops below 5% for message sizes over ~1000 bytes with larger messages having

negligible overhead. In order to determine how this added MPI overhead affects scientific applications, I ran a proxy application with all SPCs turned on and without SPCs enabled in Open MPI.

For this test, I used the *in.reaxc.hns* problem from the Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) [46]. I ran this test on a small x86 system with 16 cores per node and 16 nodes, for a total of 256 cores with an Infiniband interconnect. This test utilizes the entire system, with one MPI process per core arranged in an $8 \times 8 \times 4$ processor grid. I performed ten runs of each configuration and calculate the average runtime of the application. Figure 3.9 shows that there is a minimal difference in the application run time between having SPCs compiled in with all counters turned on, and running without SPCs compiled into Open MPI. In addition, the average overhead added to MPI operations throughout this application is less than 1% given that many of these communications are between nodes and are over 100 bytes in size.

3.8 SPC Reporting Methods

With all of this instrumentation added to Open MPI, there must be a way for users to access this information. I decided to provide a variety of methods for accessing SPCs, to give users options that suit a variety of use cases. These methods are: printing SPCs to *stdout* in MPI_Finalize, through the MPI_T interface, and through my custom *mmap* interface which also has a snapshot feature.

3.8.1 Printing to *stdout*

Essentially, the way this method works is that in MPI_Finalize, all of the SPCs in the MPI_COMM_WORLD communicator perform an MPI_Gather operation to put all of the data on rank 0, which then prints the data to *stdout* with a human readable format in rank order from rank 0 to the maximum rank. Printing the SPC values to *stdout* has the advantage that the user doesn't need to modify their code to get the counter values, however there are several drawbacks to this method. The most obvious drawback is that if there are a large number of processes, the output would print many lines and would be hard for a

person to read and make sense of it. For more systematic analysis, the user would need to capture *stdout* and then parse the text to get the information they are interested in. This is obviously not an ideal reporting method but can be useful for quick assessment of small application runs.

3.8.2 MPI_T Performance Variables

All of the SPCs are registered as MPI_T performance variables during the SPC initialization process. More specifically, SPCs are registered as performance variables with the *long long* type, the *counter* performance variable class, the *no object* binding, and are flagged as read only and continuous. This tells the MPI_T interface that these counters cannot be modified by the user, do not need to be bound to an MPI object, and their value does not not need to be modified by the MPI_T interface at all. It is important to provide these stipulations to allow for my own management of SPCs and to reduce potential overhead within MPI_T functions. It is worth noting that there is a special interaction for watermark counters when they are being read through the MPI_T interface: watermark counters are reset to the current value of their sentinel counter when they are read.

Using the MPI_T interface can be somewhat cumbersome with all of the functions and data structures required to find, initialize, and read MPI_T performance variables. This is a result of the design of the MPI_T interface being as general as possible and supporting a wide variety of use cases. See Section 3.3.3 for a more detailed discussion of the MPI_T interface.

3.8.3 SPC *mmap* Interface

The initial idea behind the *mmap* interface was to provide an alternative to the MPI_T interface that must be portable, have low overhead, and allow for both in-situ and post-mortem analysis. As the name implies, this interface relies on the *mmap* function, which is widely available across Linux systems. The *mmap* function creates a new mapping of pages in the virtual address space of the calling process [32]. This new mapping can optionally be attached to a file with a variety of different permissions and properties. I decided to map

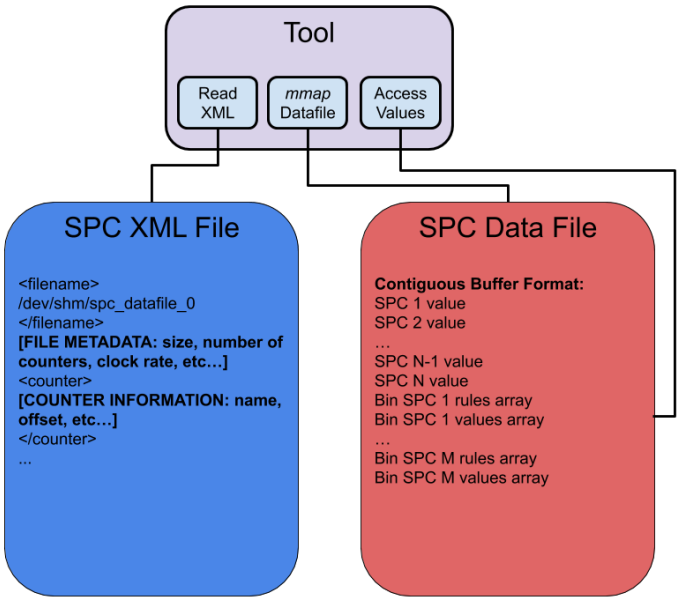


Figure 3.10: A diagram of the operation of the SPC *mmap* interface. Note: the "_0" in the data file name refers to the process rank (rank 0).

the new pages to a file, typically stored in a system location such as */dev/shm*, and make the mapping read only as well as shared. This means that any process that maps to the same memory region will be able to read the memory, and all updates to the memory will be written back to the file.

What this means for SPCs is that users and tool developers can get direct read only access to the counter data without having to perform any library calls after the initial mapping. Thus, from the user perspective, the overhead of using the *mmap* interface is essentially the same as any other memory load operation. Since I attach the new pages to a file, I can make that data file persist after the run for use in post-mortem analysis.

As explained in Section 3.6.1, the SPC values and related data is stored in one contiguous buffer. This buffer is what is mapped into the data file, so a user needs some way to determine where in that contiguous buffer the SPCs they are interested in are located. To facilitate this, I use an XML file that provides metadata that might be useful for performing analysis such as the name and path to the corresponding data file, the clock rate (for converting cycles to time), the number of counters, etc... There is also an entry for each SPC in the

XML file that provides offsets into the data file for the counter's value, and potentially the rules and bin values if the counter is a bin counter. So, an example usage would be for a tool to open the XML file, store the data file's location, read the number of counters, search through the list of SPCs for the metrics they are interested in, store the offsets to those events, use *mmap* to attach to the SPC data pages, and then use the offsets to read the values of interest. A simplified version of this example is shown in Figure 3.10.

One major difference for reading counters with the *mmap* interface is with watermark counters. With the MPI_T interface, watermark counters are reset to their sentinel value when read. This is not possible with the *mmap* interface, because there is no simple and fast way to tell when a counter value has been accessed.

3.8.4 SPC Snapshot Functionality

With the *mmap* interface introduced in Section 3.8.3, a user or tool can get a direct mapping into the SPC data to be used during runtime, or read the backing data file for post-mortem analysis. This provides for two of the more common use cases, however there is another case where the user wants regular snapshots of the counter values. I decided to implement an automatic snapshot feature to allow users and tools to specify a period of time after which a copy of the data file on a process will be made. The idea is that without modifying their code, a user could get access to the change in the counter values over time. This is implemented in the Open Run-Time Environment (ORTE) layer within Open MPI as an event added to an event loop. If the time between events exceeds the user-defined period of time, another copy of the data file is made. The data file copies append the timestamp to their name in order to provide context for that data. An example of what this process looks like can be found in Figure 3.11

3.9 SPC Use Case Results

There are many potential use cases for SPCs, but in this section I will be focusing on three use cases with separate areas of focus:

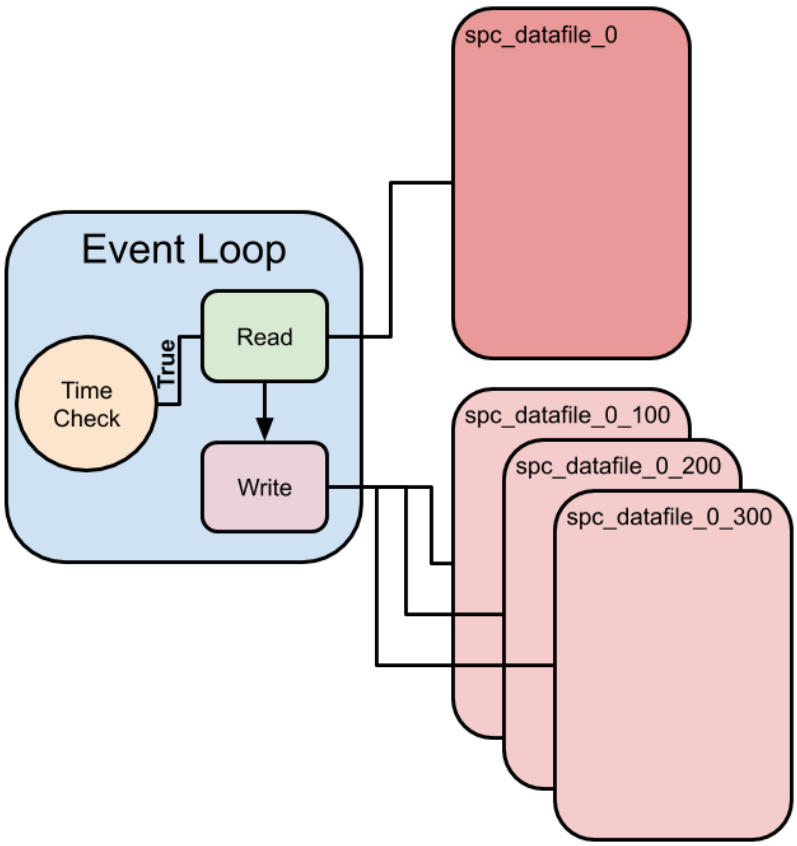


Figure 3.11: A diagram of the operation of the SPC snapshot feature using the *mmap* interface. Note: the write operation always creates a new file with the current timestamp appended to the end of the name, and the "_0" in the names refers to the process rank (rank 0).

1. Diagnosing issues in the MPI implementation
2. Application performance analysis
3. Machine/datacenter workload characterization

The first use case focuses on introspection of the MPI implementation to determine whether it is performing the way it was intended to. This could be looking at whether there is performance degradation due to new changes, or the choice of algorithm for a collective operation, or any number of other problems. The second use case focuses on analyzing a particular application and determining why it performed the way it did from an MPI perspective. This is not so much diving into why the implementation is doing what it is, but how the implementation affects the performance of a specific application. The third use case is the most general in that the information could be gathered across a variety of different applications and/or architectures and provide a general picture of what the MPI usage is like in these various scenarios.

3.9.1 Diagnose MPI Implementation Issues: Out of Sequence Messages Case Study

In this section, I will focus on the first use case of identifying issues with the MPI implementation. I performed a case study of one particular issue that I helped identify in the Open MPI implementation with the use of SPCs. The issue in question is the performance degradation of the multithreaded MPI implementation due to out of sequence (OOS) messages. I performed tests on two different sample applications to help identify this issue. The first application, called *multirate*, is a synthetic benchmark that is designed to measure message injection rate, extraction rate, latency, and bandwidth in a variety of different communication patterns and levels of multithreading [43]. The second application is a quantum chemistry application called *moldft* that was implemented with the Multiresolution Adaptive Numerical Environment for Scientific Simulation (MADNESS) [25]. Both of these applications have the option to run in a multithreaded environment using multithreaded MPI.

Table 3.4: Results of the multirate benchmark using the pairwise communication pattern with: 2, 4, and 8 threads, a window size of 256, message size of 64 bytes, and iteration count of 100. Note: The message rate and wall time do not include the warm-up phases, but the other values do. Without warm-up messages there are $256 \times 100 \times N_t$ messages sent where N_t is the number of threads.

Threads	Message Rate (msg/sec)	Receives	OOS Messages	% OOS	Wall Time (us)	Match Time (us)	OOS Match Time (us)
2	601,773.54	56,320	16,633	29.53%	85,598	9,634	9,875
4	476,174.73	112,640	47,216	41.92%	218,807	34,312	51,196
8	162,458.93	225,280	112,813	50.08%	1,260,863	96,465	729,187

Multirate Benchmark with the Pairwise Communication Pattern

The *multirate* benchmark is able to isolate the performance characteristics of various areas within an MPI implementation such as its capability to inject messages into the network or extract messages from the network. This is done through providing a variety of capabilities such as flexibility in usage of threads and processes for communication participants, and different communication patterns like many-to-one, one-to-many, and many-to-many. The communication pattern that provides the best-case scenario is the *pairwise* communication pattern, where every communication entity is paired with one other communication entity.

Since I am focusing on multithreaded MPI in this test, I initialize multithreading in MPI with MPI_THREAD_MULTIPLE, which allows for all threads to execute MPI functions concurrently. I have also set up the multirate test to have two MPI processes, one for sending (rank 0) and one for receiving (rank 1). Each of the MPI processes creates T threads and connects these threads pairwise between processes with all processes and threads sharing a single communicator, such that the first thread from rank 0 is paired with the first thread on rank 1 and so on. The paired threads only communicate with their partner thread. The benchmark performs a warm-up phase, and then calculates message rate by posting a window of W asynchronous sends of size S from each thread on process 0 to their respective partner threads on process 1, repeating this procedure I times and then dividing messages by time. In this case, I chose to use a relatively small message size of 64 bytes to focus on the stress to the MPI implementation rather than the communication hardware. I also used a window size of 256 and an iteration count of 100 to provide the potential for saturating

the communication infrastructure within the MPI implementation. I used the *vader* BTL for shared memory communication and placed the threads from each process on different CPU sockets to alleviate potential issues with cache interactions between threads from the separate processes. I then ran this experiment with 2, 4, and 8 threads per process.

Ideally, when the number of parallel entities increases, there would be an opportunity for higher message rate since messages are being injected and extracted from the network by several entities at once. This has been shown to be true when communicating exclusively with MPI processes without multithreading, however, there tends to be some performance degradation when enabling multithreading due to the overhead introduced for thread safety in multithreaded MPI. Theoretically, the performance of communicating between a series of pairs of threads should follow that of a series of pairs of processes with some factor of overhead added to the threads. At the very least, one would expect that the performance should increase up to a certain plateau as the number of threads increases.

The results of the multirate tests, shown in Table 3.4, tell a much different story. The message rate is significantly decreasing as the number of threads increases. To investigate this further, I looked at the SPC counter values, and I see that as the number of threads increases, so too does the number of out-of-sequence (OOS) messages. Messages that arrive out of the proper sequence cause a similar problem to messages that arrive unexpectedly in that they must be stored in an internal queue until they can be completed properly. In these particular tests, the number of OOS messages is becoming a significant portion of the total messages with 29.53% of the messages arriving as OOS messages with 2 threads and over 50% of the messages with 8 threads. With all of those messages being pushed into the OOS message queue, the time spent matching these messages increases with a much larger queue of OOS messages to search through. In these tests, the time spent matching OOS messages becomes a severe bottleneck for message rate with matching time reaching over half of the total wall time.

So OOS messages reduce the performance, but why are these messages arriving out of sequence in the first place? The answer lies in the fact that the multirate benchmark was configured to perform all the sends and receives between threads within a single communicator. With all of the threads in one communicator, they compete for acquiring

sequence numbers and memory locations to perform the data transfers since both of these operations are atomic. When multiple threads attempt these operations at the same time, the order in which they get what they need is nondeterministic. The OS can also deschedule the threads at any time which can influence the order in which they transfer the data. Effectively, when multiple threads attempt to send data at the same time, their order of acquiring sequence numbers, and the order in which they actually acquire a memory location to write to are not necessarily in the same order. The more threads there are, the more likely it is for these collisions to occur, which then increases the number of OOS messages.

Demonstrating the OOS Message Problem in the MADNESS Application

MADNESS comes packaged with several test benchmarks, one of which is called *moldft*. This benchmark takes a set of molecular geometry as input and performs a molecular dynamics simulation based on density-function theory. MADNESS operates in a multi-threaded MPI environment in which any thread can communicate with any other thread directly. As I have shown with the *multirate* benchmark, using multi-threaded MPI can potentially cause a large number of OOS messages when the thread counts are high. In a sense, the multirate tests show a worst-case scenario where many threads are putting a lot of stress on the MPI library all at the same time. On the other hand, the multirate tests only had pairs of threads communicating and did not look at what happens when any thread can communicate with any other thread. This test with MADNESS will serve to show a real-world case where any thread can communicate with any other thread, and those communications can cause delays to computation.

To test the impact of OOS messages on simulation performance, I decided to use three different BTLs to run the simulation: *vader*, *openib*, and *TCP* over InfiniBand. The *vader* shared memory BTL and the *openib* InfiniBand BTL can both potentially allow messages to be sent out of sequence, while the *TCP* BTL does not allow OOS messages. To provide a fair comparison between the different BTLs, I hold the number of threads constant at 18 with 9 on each node for *openib* and *TCP*. I decided to use a moderate sized problem within *moldft* that performs a simulation of five water molecules.

Table 3.5: The results of the MADNESS *moldft* tests using five water molecules. The counter values are the average of 10 runs with 18 threads per run of the simulation for each configuration. Note: the total time is the wall time reported by *moldft*.

Counter	*openib*	*vader*	*tcp*
Total Time (sec)	626.41	440.95	518.54
OMPI_RECV	12,995.6	13,024.7	12,710.6
OMPI_ISEND	3,252,957.0	3,253,238.0	3,143,029.9
OMPI_IRECV	3,291,284.3	3,291,596.8	3,180,212.8
OMPI_BCAST	4.0	4.0	4.0
OMPI_BYTES_RECEIVED_USER	1,898,491,237.9	879,724,185.9	23,428,171,947.7
OMPI_BYTES_RECEIVED_MPI	168.0	168.0	168.0
OMPI_BYTES_SENT_USER	1,980,636,856.5	968,525,668.5	23,675,080,020.9
OMPI_BYTES_SENT_MPI	168.0	168.0	280.0
OMPI_BYTES_PUT	0.0	0.0	129,295,502.3
OMPI_BYTES_GET	23,032,934,218.0	24,056,241,711.4	0.0
OMPI_UNEXPECTED	126,339.4	21,654.5	14,868.5
OMPI_OUT_OF_SEQUENCE	1,222,397.6	134,631.0	0.0
OMPI_MATCH_TIME	282,910.2	369,343.7	251,844.7
OMPI_OOS_MATCH_TIME	317,157.8	32,742.9	0.0

Table 3.5 shows the results of the MADNESS experiments. Under normal circumstances, one would expect that using the optimized InfiniBand BTL would outperform TCP, but this is not the case for these tests. The TCP over InfiniBand test ran ~20.8% faster than the pure InfiniBand test on average. As expected, OOS messages have a huge effect on performance here with ~37% of the messages in the *openib* tests being delivered out of sequence. With this many OOS messages, the time spent managing the OOS data structures and matching messages also increases. The shared memory test ran ~17.6% faster than TCP, and ~42% faster than *openib* and had much fewer out of sequence messages than *openib* with only ~4% of the messages arriving out out sequence.

These tests show that even in a situation where the communications are naturally spaced out across an application run and interspersed with computation there can be a significant portion of the messages delivered out of sequence in a multithreaded environment. This also shows that the *openib* BTL has even more of a problem with out of sequence messages than the *vader* BTL that was used for the *multirate* tests.

Improvements to Multithreaded MPI Performance

After identifying this issue of out-of-sequence messages in multithreaded MPI, some improvements were made by Open MPI developers to alleviate the number of out-of-sequence messages in the multithreaded case. Patinyasakdikul et. al. [42] show a number of improvements made to handling multithreading in MPI with particular focus to the progress engine and the matching process. They proposed a method for almost completely removing OOS messages by performing concurrent matching in addition to concurrent progress in Open MPI. They simulated this by performing all communications between pairs of communicating entities within separate communicators. Since matching in Open MPI is performed per pair of processes per communicator, isolating each communication pair in their own communicator effectively removes competition between threads as a factor with respect to sequence numbers. It is still possible to see OOS messages in a single-threaded case, but it is quite rare. While this method is somewhat extreme, it demonstrates a way to significantly improve the performance of multithreaded MPI.

3.9.2 Identify Application Bottlenecks

When it comes to identifying application bottlenecks, there are many potential issues that SPCs can highlight. They can be used to get higher precision bandwidth numbers than simply calculating based on timers placed around an MPI function, they can identify which algorithms are being used for collectives that may or may not suit the application, they can identify how well the matching process is being handled, etc... In this section, I want to focus on a real-world example where SPCs were used to identify a bottleneck with an application and provide a method for quantifying the improvements of the solution.

Identifying a Bottleneck in a Local Rollback Approach to Fault Tolerance

The study I will be looking at was conducted by Losada et. al. [34]. This study was focused on solving a few practical limitations of performing a local rollback of a checkpoint/restart scheme over User Level Failure Mitigation (ULFM) fault tolerant MPI. Local rollback is meant to only replay communications between the failed process and other processes affected

by the fault. One of the problems with this approach is that all of the communications from remote processes to the failed process will come flooding in at once and overwhelm the recovering process.

The authors decided to use SPCs to identify the fact that the number of unexpected messages significantly increases when local rollback is used. This is because the recovering process is not able to post the appropriate receives fast enough to be matched with all of the incoming messages from the remote processes that are replaying their messages. To solve this problem, the authors decided to use RMA operations for their local rollback so the recovering process can simply get the information it needs at the appropriate time without having the remote processes overwhelming it with messages. The authors were able to demonstrate with SPCs that their RMA-based local rollback approach drastically reduced the number of unexpected messages and showed a 59% reduction in recovery times across all of the applications they tested.

3.9.3 Workload Characterization

The SPC *mmap* interface with the *snapshot* feature lends itself well to performing workload characterization of individual applications or across many applications on a particular machine or across an institution. Since the mmap interface allows for storing data files with the counter values in them, it is easy to look back at the counter values for a particular application. With the snapshot feature, this can become more precise with several data files storing the counter values at various time slices throughout the application's lifetime.

To test this capability, I have created a simple test where I will run an example code called *indent* provided by the Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) [46]. The *indent* test is described as simulating a spherical indenter interacting with a two dimensional solid. I performed this test with two small CPU-based nodes each with two Intel Xeon processors with 10 cores on each processor. I ran the test with 40 MPI processes, one for each core in an $8 \times 5 \times 1$ processor grid. I enabled the *mmap* interface with snapshots turned on and set to record every 0.5 seconds. I enabled all of the SPCs, though I will primarily be focusing on seven counters:

Table 3.6: The MPI collective operations used in the LAMMPS *indent* test.

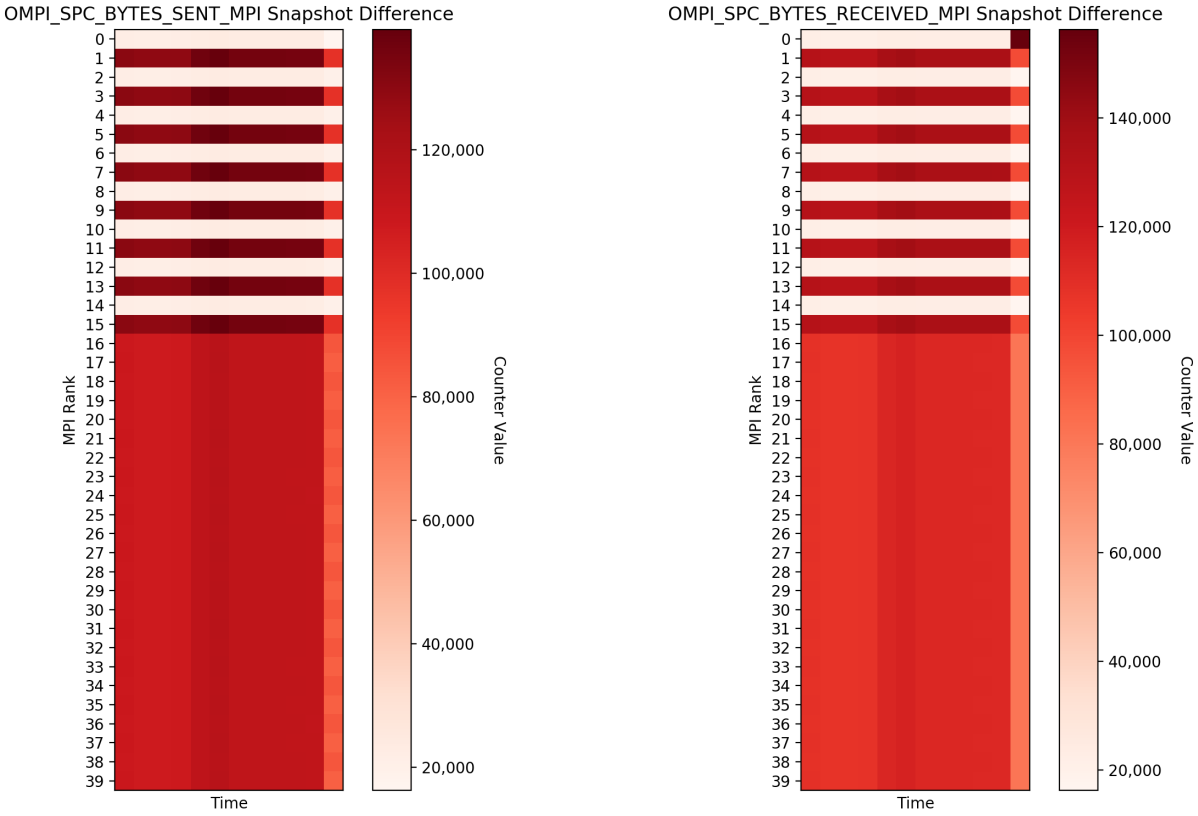
MPI Function	Algorithm	Approximate Calls/Process
MPI_Bcast	Binomial Tree	130
MPI_Reduce	Binomial Tree	6
MPI_Allreduce	Recursive Doubling	54,000

- OMPI_SPC_BYTES_SENT_MPI
- OMPI_SPC_BYTES_RECEIVED_MPI
- OMPI_SPC_BYTES_SENT_USER
- OMPI_SPC_BYTES_RECEIVED_USER
- OMPI_SPC_MATCH_TIME
- OMPI_SPC_MATCH_QUEUE_TIME
- OMPI_SPC_UNEXPECTED

To visualize the results of these tests, I created heatmap plots for each of the aforementioned counters across all MPI ranks and SPC snapshot data files. These plots will show the difference in the raw counter values between two snapshot files in each box. In this experiment, I want to primarily focus on what information can be provided from my counters and information that is provided at a high level by LAMMPS, without looking at the implementation of LAMMPS.

Collective Operations

The bytes sent/received by MPI counters primarily focus on data sent through MPI collective operations. In Figure 3.12, you can see that there is a very distinct pattern to the bytes transferred through collective operations in this test. Out of the first sixteen ranks, there is an alternation between light and heavy amounts of data transfers. To dig deeper into why



(a) Bytes sent by MPI, primarily through MPI collective operations.

(b) Bytes received by MPI, primarily through MPI collective operations.

Figure 3.12: Heatmaps of the bytes sent/received by MPI during a run of the LAMMPS *indent* test with 40 MPI processes across two nodes. Each box represents the counter values added in a 0.5 second time slice of the application run.

Table 3.7: The MPI point-to-point operations used in the LAMMPS *indent* test.

MPI Function	Approximate Calls/Process
MPI_Send	$350,000 \rightarrow 450,000$
MPI_Irecv	$350,000 \rightarrow 450,000$
MPI_Sendrecv	9,400

this is, I looked at which collectives were used. This particular example used the broadcast, reduce, and allreduce collectives as shown in Table 3.6.

The vast majority of the collectives performed were MPI_Allreduce operations using the recursive doubling algorithm. The recursive doubling algorithm requires a power of two processes to operate properly, however this run is using 40 processes which is not a power of two. Open MPI gets around this by first reducing the number of processes involved to the nearest lower power of two processes, which is 32 in this case, and then performing the standard recursive doubling algorithm. The full algorithm for this is shown in Figure 3.13. Essentially, if M processes need to be removed out of N total processes to get to a power of two, the following algorithm is used: (1) the first member of the first M pairs of processes sends its value to the second member of its pair to perform the op (sum, difference, etc...); (2) The remaining $N - M$ processes perform the recursive doubling algorithm; (3) The second member of the first M pairs of processes sends the final value to the first member of its pair. Thus, at the end there will be M processes with 1 send/receive, M processes with $\log_2 N + 1$ sends/receives, and $N - 2M$ processes with $\log_2 N$ sends/receives.

Knowing how Open MPI handles recursive doubling when the number of processes is not a power of two provides insight into why the heatmaps in Figure 3.12 look the way they do. Since there are 40 processes, 8 processes need to be removed to get to a power of two. So, we see that the first 8 pairs of processes exhibit the behavior of the first member of each pair being removed from the rest of the algorithm and the second member of each pair having extra communication to keep those removed processes updated.

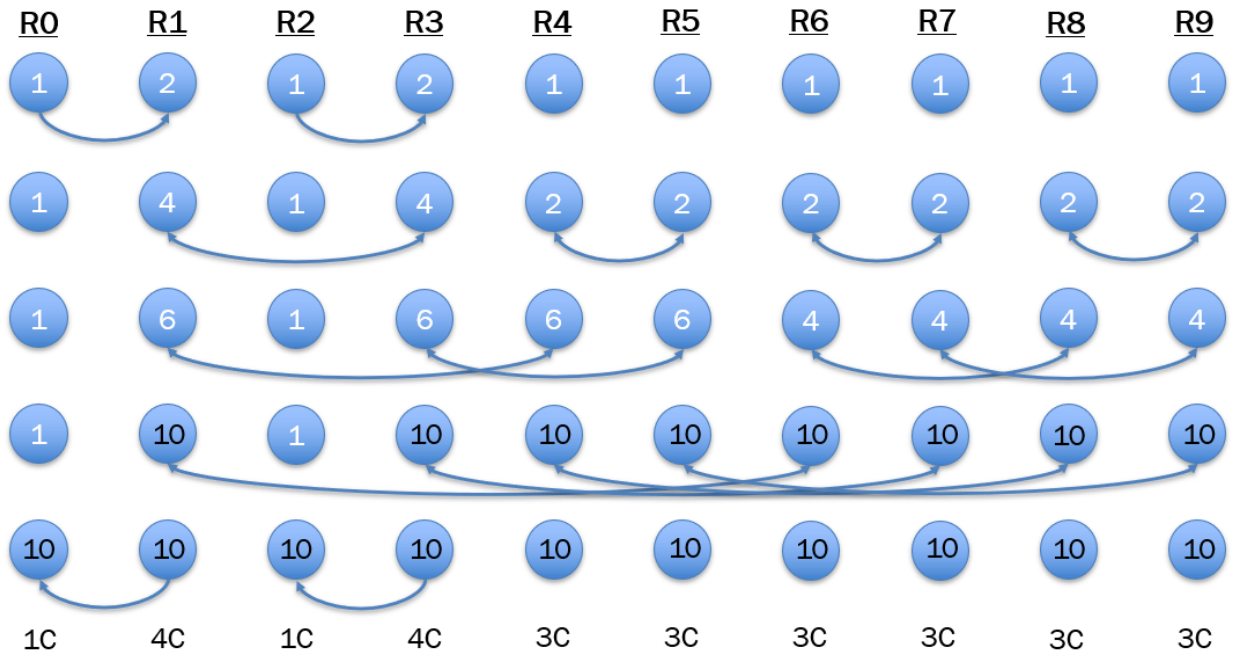
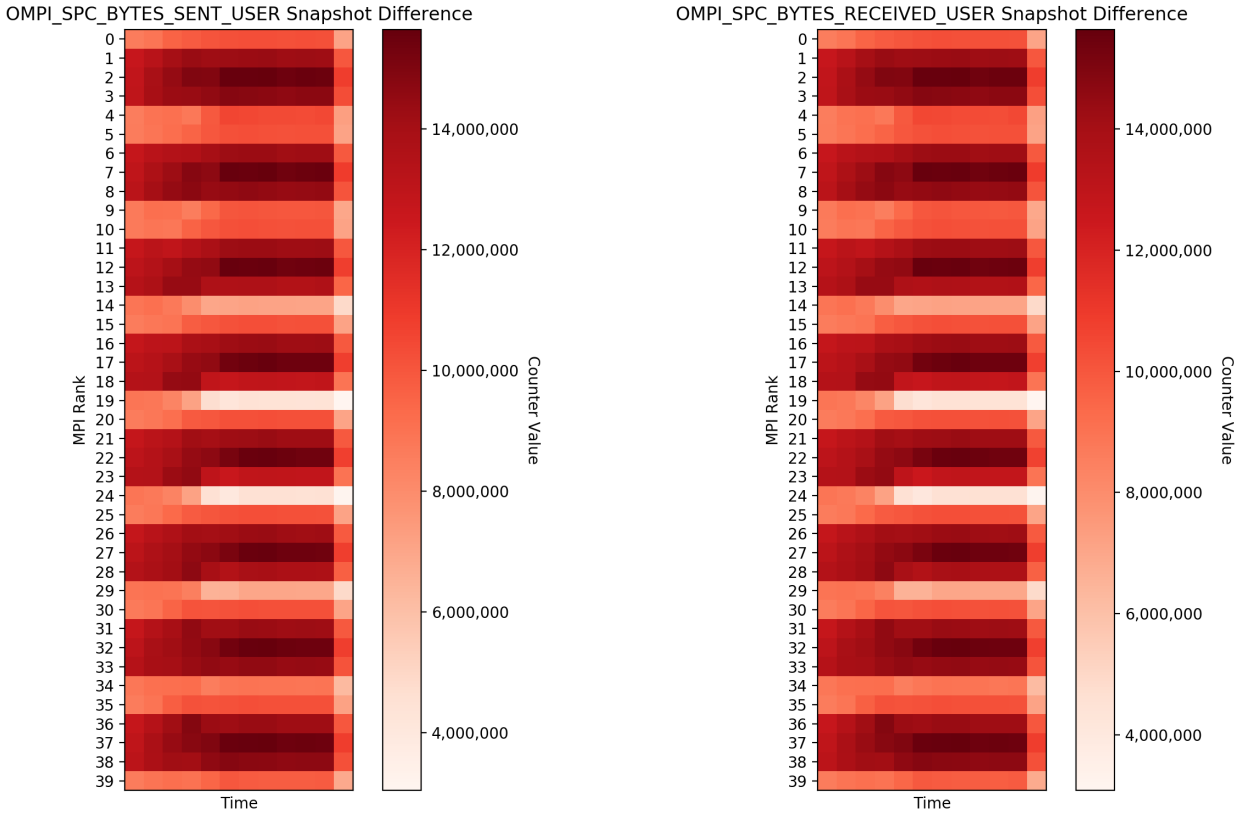


Figure 3.13: A visualization of the recursive doubling algorithm for an MPI_Allreduce as implemented in Open MPI. This assumes that there are 10 processes, each with a starting value of 1 with a summation operation. The 'R' labels indicate the MPI rank, and the 'C' labels indicate the number of round-trip communications performed on a given rank.



(a) Bytes sent by the user, primarily through MPI point-to-point operations.

(b) Bytes received by the user, primarily through MPI point-to-point operations.

Figure 3.14: Heatmaps of the bytes sent/received by the user during a run of the LAMMPS *indent* test with 40 MPI processes across two nodes. Each box represents the counter values added in a 0.5 second time slice of the application run.

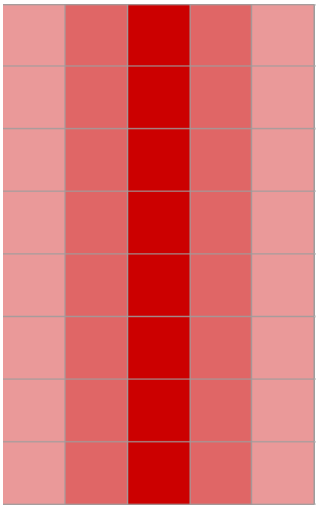


Figure 3.15: An approximate representation of the heatmaps in Figure 3.14 with the processes arranged in an 8×5 processor grid.

Point-to-Point Operations

When looking at the point-to-point messaging results for the LAMMPS test, things become a bit more difficult to parse. Figure 3.14 shows a pattern of hotspots much different from what we see with the collective communications. This pattern shows what appears to be a periodic behavior across processes with the bytes sent/received going from light to heavy and back again.

To get some more insight into this pattern, I looked at which MPI point-to-point operations were being used in this test. The three MPI functions used were: MPI_Send, MPI_Irecv, and MPI_Sendrecv as shown in Table 3.7. The MPI_Sendrecv function was used the same number of times per process, like with the collective operations. This is likely for performing *halo exchanges* of some ghost region data between processes. Unlike the collective operations, however, the MPI_Send and MPI_Irecv operations called per process vary wildly on the order of 350-450 thousand bytes.

To understand why this pattern emerges, it helps to look at what this application is simulating. The description is that there is a spherical indent into a two dimensional solid. I think of this solid as standing up with atoms stacked on top of each other, like grains of

sand in an ant farm. Then, a spherical indenter is pushed down from the top, displacing the atoms in its way.

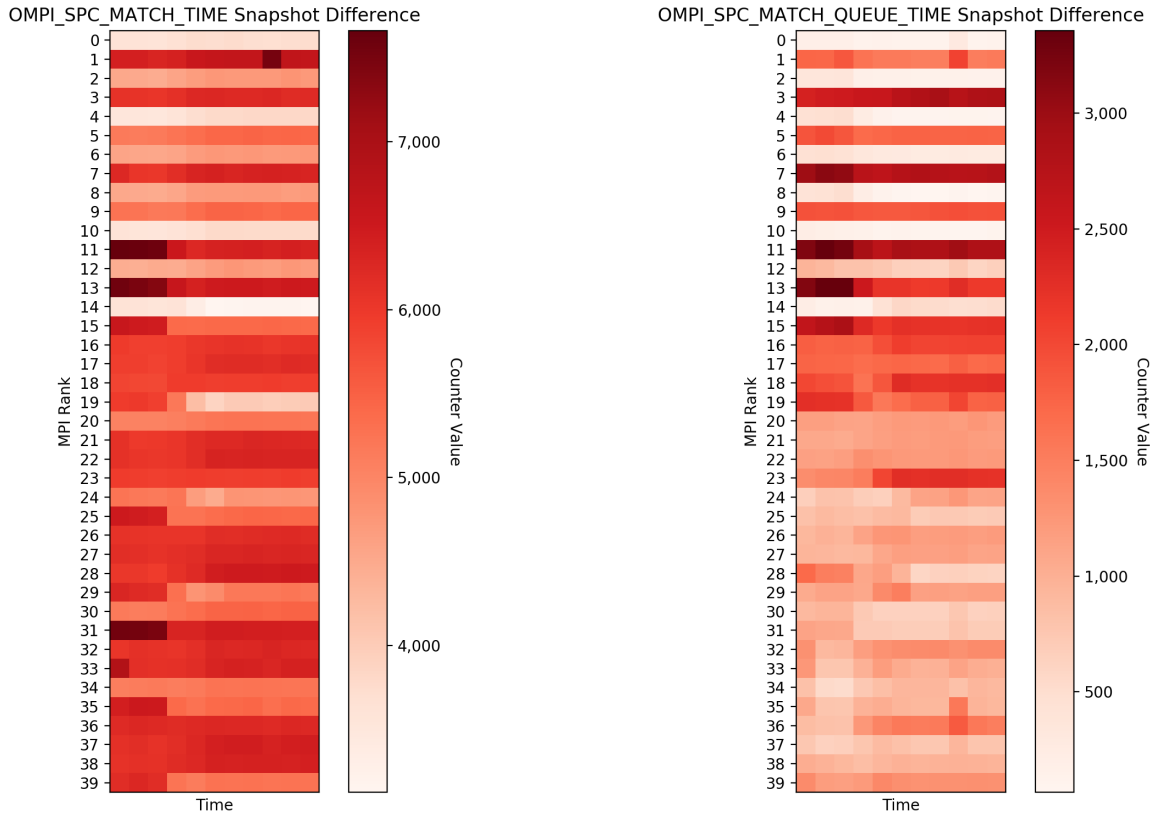
Figure 3.15 shows an approximation of what the bytes sent/received heatmap would look like if mapped onto the 8×5 processor grid. With this visualization, it is easy to see the path of the indenter going into the solid roughly in the middle. So, it seems that the processes with more communication are those which have more atoms displaced by the indenter.

Matching

When looking at message matching, there are two major aspects of matching that I want to focus on: attempting to match a message and inserting messages into the unexpected message queue. When matching messages, the number of posted receives in the posted receive queue can impact how long it takes to find whether there is a match. Once a match is found, if the match was delivered unexpectedly, it must be found in and removed from the unexpected message queue. As the size of the unexpected message queue grows, this process will take longer. In this particular experiment, the maximum number of messages in the unexpected message queue varied by process, but never exceeded 25 messages and was only above 15 for one process. So, for this particular experiment it would not take particularly long to find and remove an element from the unexpected message queue.

To compare these two matching areas, I look at the OMPI_SPC_MATCH_TIME and OMPI_SPC_MATCH_QUEUE_TIME SPCs. The MATCH_TIME counter keeps track of the amount of time spent attempting to find a match, and the MATCH_QUEUE_TIME counter keeps track of the amount of time spent inserting messages into the unexpected message queue, potentially including time spent allocating additional memory if necessary. In Figure 3.16 I show a comparison of the heatmaps for these two counters.

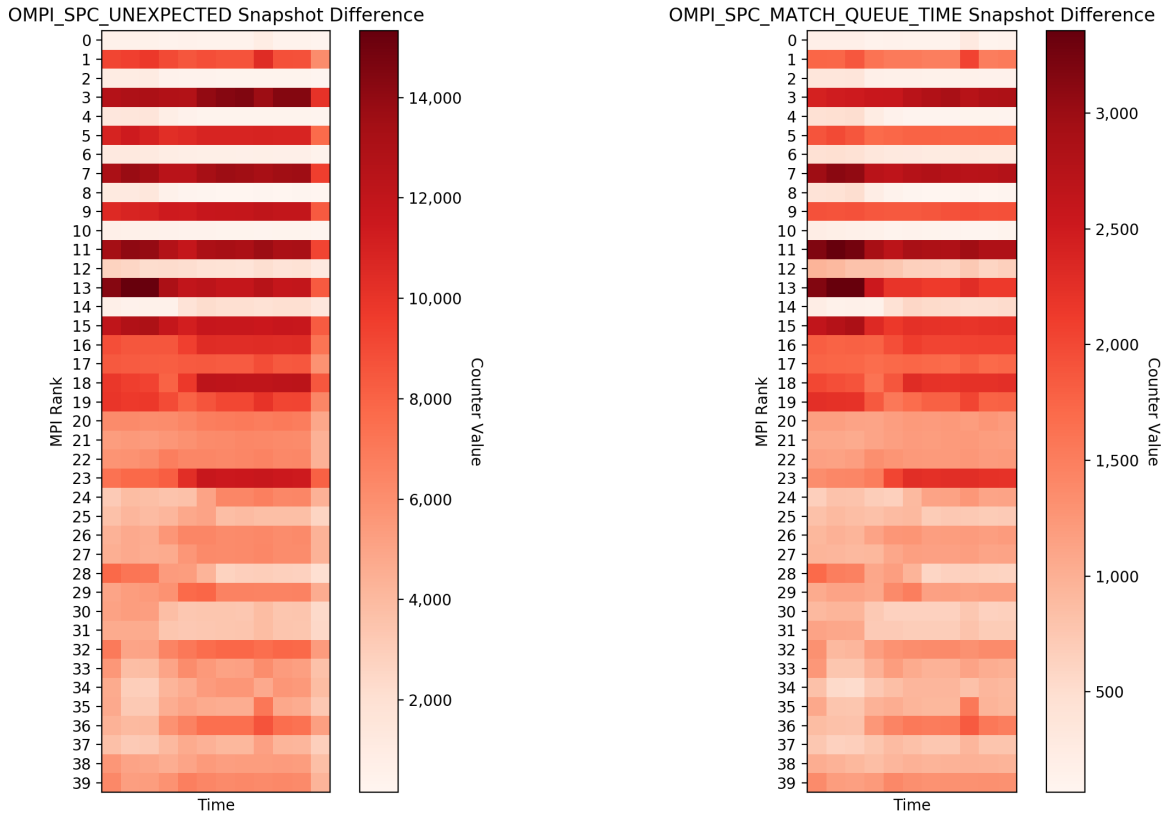
In Figure 3.16a, the pattern from the collective communications shows up for the first 16 processes, and the pattern for point-to-point messages is more prevalent in the later processes (though still visible to a certain degree in the earlier processes). Since matching must happen for each message, as the number of messages increases, so too will the time spent matching them. This time is increased when there are more unexpected messages, since the data must be found in and copied from the unexpected message queue. In Figure 3.16b, the pattern is



(a) Time spent attempting to match messages.

(b) Time spent inserting messages into the unexpected message queue.

Figure 3.16: Heatmaps of the time spent in the process of matching during a run of the LAMMPS *indent* test with 40 MPI processes across two nodes. Each box represents the counter values added in a 0.5 second time slice of the application run. Note: The timer counter values have been converted to microseconds.



(a) The number of unexpected messages encountered.

(b) Time spent inserting messages into the unexpected message queue.

Figure 3.17: Heatmaps of the number of unexpected messages and time spent inserting those messages into the queue during a run of the LAMMPS *indent* test with 40 MPI processes across two nodes. Each box represents the counter values added in a 0.5 second time slice of the application run. Note: The timer counter values have been converted to microseconds.

very similar to that of the collective communications with the first 16 processes alternating high and low values, while the remaining processes are relatively similar.

Since this application is using MPI_Irecv for the point-to-point communication, I suspect that many of the point-to-point messages are pre-posted messages, which means that they are unlikely to result in unexpected messages. On the other hand, MPI collective communications can cause increased unexpected messages due to their usage of the eager protocol for sending message fragments. In this case, most of the collectives are transmitting relatively small payloads so nearly all messages sent inside of these collectives will be sent with the eager protocol. Figure 3.17 shows that the there is a nearly perfect match between the number of unexpected messages and the time inserting those messages into the unexpected queue as one would expect. Comparing Figures 3.12 and 3.17 shows that processes with more communication as part of collectives tend to have more unexpected messages, and thus spend more time inserting those messages into the unexpected message queue.

3.10 Conclusion

In the interest of fulfilling the need for lower-level performance information for MPI performance analysis, this chapter assessed the current state of performance analysis in MPI with a particular focus in approaches available through the MPI Standard. I proposed the Software-based Performance Counters approach to address the lack of low-level performance metrics and implemented and evaluated my approach in the Open MPI implementation of the MPI Standard. I demonstrated that my approach can provide useful low-level metrics about the MPI implementation that can be used to perform detailed performance analysis of MPI applications with low overhead added to the MPI library. In summary, my contributions in this chapter are:

- **Software-based Performance Counters:** Performance counters reminiscent of PAPI counters that are added to an MPI implementation at key locations in order to provide context to how an MPI implementation is operating. I introduced several different types of counters such as *regular counters*, *timer counters*, *watermark counters*, *bin counters*, and *collective bin counters*. I added counters that keep track of metrics

such as: the number of invocations of high-level MPI functions such as MPI_Send; the algorithms used for MPI collective operations broken down by the size of the message and communicator; fine-grained data transfer information for both data sent and received by an MPI process; and detailed internal queue usage information.

- **Integration of SPCs with MPI_T:** I registered all of my SPCs as MPI_T performance variables so they can be accessed through the MPI_T interface. This allows tools to query, enable, and access my counters the same way they would any other performance variable exposed by Open MPI.

- **Custom Interface for Reporting SPCs:** I implemented an *mmap* interface that allows for SPCs to be stored in a shared data file that can be attached to by any process, and gives direct read-only access to the SPCs without having to spend the function overhead of using the MPI_T interface. These data files are also accessible after a program's execution for postmortem analysis and are associated with an XML file which provides the details of where each SPC is stored in the data file. This interface also supports creating automatic snapshots of the data files periodically, which can show the change in the counter values over time.

Chapter 4

GPU Load Imbalance

4.1 Chapter Overview and Acknowledgment

This chapter provides a discussion of my work on GPU load imbalance analysis, with a focus on using a synthetic GPU efficiency metric to analyze a proposed new load imbalance formula for GPU kernels. The work in this chapter was performed over the course of two summer internship programs at Lawrence Livermore National Laboratory (LLNL) under the supervision and mentorship of Dr. Olga Pearce (LLNL) and with significant contributions from Kewen Meng (University of Oregon) in the early code development, and Dr. David Boehme (LLNL) for his expertise in performance analysis, particularly with the *Caliper* tool.

Section 4.2 provides an introduction and motivation for this work. In Section 4.3, I will provide some relevant background information for the topics explored in this chapter. Section 4.4 details the design and implementation of my modifications to the CoMD proxy application as well as the metric driven load balancing approach. Section 4.5 describes the experimental setup and analysis of the results of those experiments. Finally, I will conclude this work in Section 4.6.

4.1.1 Auspices

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-TH-808649).

4.1.2 Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government.

Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights.

Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC.

The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

4.2 Introduction

Modern High-Performance Computing (HPC) systems have grown to meet the needs of state of the art scientific simulations for solving some of the world's hardest problems. The largest HPC systems use millions of independent processors, and with the increasing popularity of GPUs, the levels of concurrency on these systems has skyrocketed. In order to maximize the utilization of these machines, it is crucial to balance the work between processors to avoid wasting computational resources. With increased concurrency, this problem becomes more pronounced since the more processes you have, the more total compute time is spent waiting on the slowest process. Dynamic load balancing is one strategy for correcting imbalance that occurs as a simulation progresses, by migrating work between processes. The first step to performing dynamic load balancing is to accurately determine the current imbalance in the simulation. Assessing load imbalance in a GPU-based simulation is more nuanced than

in a CPU-based simulation due to the hierarchical parallelism inherent to GPUs and the increased concurrency that brings.

The aim of this work is to study the load imbalance of CPU-based vs. GPU-based simulations and assess whether the methods used to quantify load imbalance on CPU-based systems are sufficiently accurate when applied to GPU-based systems and if not, provide a new technique to quantify load imbalance on GPU-based systems. When assessing the viability of a technique for large-scale simulation codes, it is often the best practice to first demonstrate the technique on an appropriate smaller-scale proxy application. In this work, I extend the CoMD molecular dynamics proxy application to allow for easy switching between CPU-based and GPU-based computation using RAJA as a portability layer [47]. The CoMD proxy application was chosen because it has been shown to have a high correlation between simulation work units and application load and has been extended to provide a method to introduce controlled initial load imbalance in previous work [44].

My contributions in this chapter are:

- Extension of a proxy application (CoMD) with a well-understood correlation between simulation units and application workload, to be portable between the CPU and GPU using RAJA as a portability layer, for studying load imbalance
- Instrumentation of CoMD to provide insight into application load balance, and collection of GPU metrics using NVIDIA's *nvprof* tool to better understand internal GPU load imbalance
- Analysis and evaluation of a load imbalance metric for GPU-based HPC systems, which improves the correlation of measurements and application workload by up to 20.61%.

4.3 Background

4.3.1 The CUDA Programming Environment

The CUDA programming environment is NVIDIA's official programming environment for their GPUs [39]. The CUDA environment provides helper functions to allow for easy

programming of GPUs along with a runtime system for organizing how those functions operate and how code is executed on the GPU. When thinking about programming with CUDA, there is a clear distinction between code that is run on the CPU, or *host* side, and code that is run on the GPU, or *device* side.

Computational functions that are run on the GPU are referred to as *kernels* and are expressed in terms of *blocks*, which, in turn, are composed of *threads*. These blocks of threads are scheduled to run on the GPU by CUDA's runtime system using the three main levels of parallelism available in CUDA: *streaming multiprocessors* (SMs), *warps*, and *threads*. The CUDA runtime schedules the blocks across SMs depending on the resources available on each SM. Since there can be many threads within each block, those threads are grouped into teams of 32 threads called *warps*. The warps are scheduled to run within an SM. Within each warp, the individual threads are scheduled to run on the physical compute cores available on an SM.

4.3.2 Load Imbalance Between MPI Processes

On CPU-only architectures, load *imbalance* is defined as the scaled maximum load on any process minus the average [44]:

$$Imbalance = \left(\frac{L_{max} - L_{ave}}{L_{ave}} \right) \times 100\%. \tag{4.1}$$

The first consideration in calculating the load imbalance with Equation 4.1 is to determine a definition for what *load* is. For this study, *load* is defined as the time an MPI process spends doing some form of *work*. Since I am focusing on distributed applications using MPI, time spent performing MPI operations such as MPI_Barrier and MPI_Wait are not considered *work* from the application perspective, so I can further define load on a given process as:

$$Load_{process} = Time_{total} - Time_{MPI}. \tag{4.2}$$

where $Time_{MPI}$ refers to the time spent waiting at MPI synchronization points.

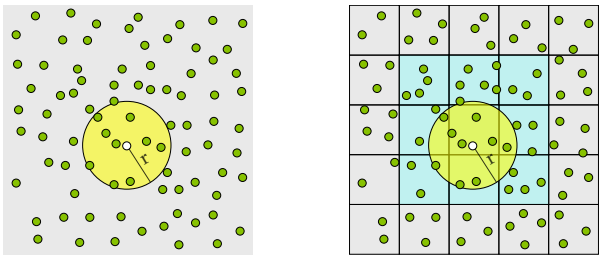
One of the main assumptions inherent to Equation 4.1 is that a process is the smallest level of parallelism in the system such that processes are not broken down into further levels of parallelism. What this translates to in a real-world example is that each MPI process would be bound to an individual processor core and only have one thread executing its workload. This is of course not always the case in modern HPC applications where one of the more popular paradigms, even on CPU-only systems, is to use an MPI+X approach where X is some form of multithreading layer such as OpenMP [40]. For simplicity's sake, when I refer to load balancing in CPU-based systems, I will be assuming that the traditional one MPI process per core approach is being used.

This is further complicated on a system where much of the work is offloaded to GPUs since there is a hierarchical nesting of parallelism inherent to the GPU architecture as discussed in Section 4.3.1. Effectively, a single GPU can be thought of as being similar to a CPU-only node in that it has several parallel elements within it. In a GPU-based system, the typical usage with MPI is to bind one MPI process to each GPU, and for simplicity's sake I will assume that this is the case throughout this study. Another typical usage scenario is that there are synchronization operations between the CPU and GPU to ensure that the GPU has finished its computation and written its result to memory before that memory is communicated by MPI. I have ensured that this is the case throughout the code used in this study, and I perform minimal operations on the CPU to keep synchronization times to a minimum.

4.3.3 CoMD Proxy Application

In order to provide a test bed for this load balancing study, I used the CoMD classical molecular dynamics proxy application that was introduced as a part of the ExMatEx project [18]. CoMD evaluates the forces that the atoms in a system exert on each other over time through two different energy potential models: Lennard-Jones (LJ) [21] and the Embedded Atom Method (EAM) [12, 13].

Rather than simulate an all-to-all interaction between the atoms, each atom only interacts with nearby atoms located within a certain radius defined by the potential function. In this case, interactions refer to the calculation of the forces that two atoms exert on each other and



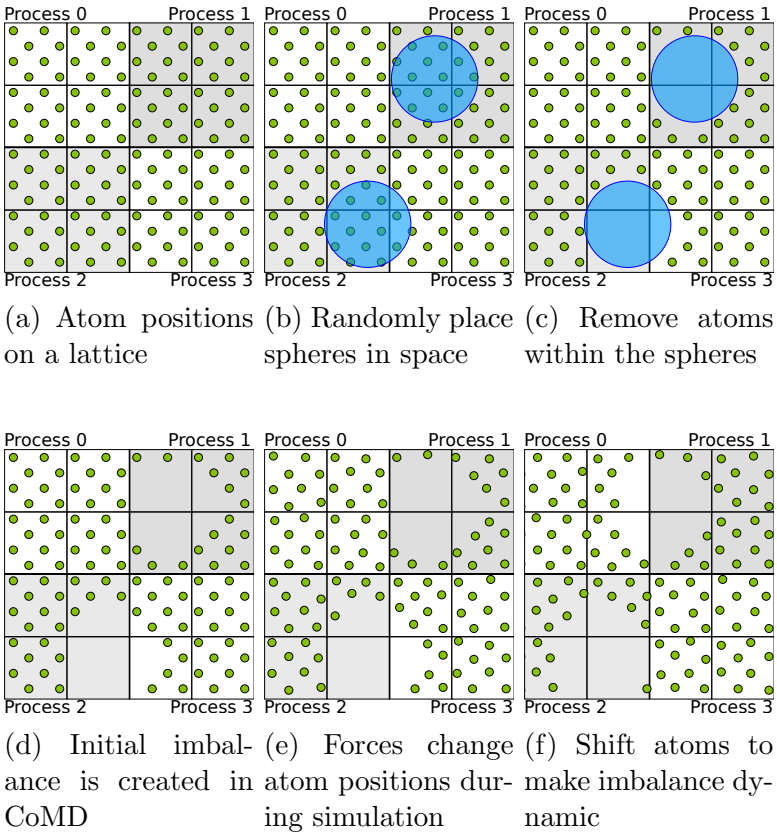
(a) Cutoff radius defines atom interactions to compute

(b) Regular grid defines proximity of atoms in space

Figure 4.1: Molecular dynamics definitions

how that would update their position, velocity, and energy. Figure 4.1 provides a simplified visualization for these interactions in two dimensions. Figure 4.1a shows an example of the radius around one particular atom. While only performing the interactions with atoms in a certain radius does reduce the number of computations, some method is needed to determine which atoms fall within the radius. To reduce this search space, the simulation area can be organized into cells that are no smaller than the cutoff radius as shown in Figure 4.1b. Thus, all atoms that could potentially be within the cutoff radius of a given atom are either in the same cell or in an immediately surrounding cell.

CoMD implements this by using a three-dimensional Cartesian spatial decomposition of atoms across processes. Each process computes the energy, velocity, and position of all atoms assigned to be within its local cells. As the positions of the atoms change, they can potentially cross the boundaries of the cells assigned to a given process. In this case, they would need to be transferred from one process to another. It is also important that cells that are on the border between one process and another have access to the border cells from the logically neighboring process in order to not ignore interactions between atoms in neighboring cells. To facilitate this, CoMD uses *ghost cells*, which are local copies of neighboring cells from a remote process. So, at the end of each time step all processes communicate which atoms moved from their local cells into the ghost region using a halo exchange, which is how ownership of atoms transfers between processes.



(a) Atom positions on a lattice

(b) Randomly place spheres in space

(c) Remove atoms within the spheres

(d) Initial imbalance is created in CoMD

(e) Forces change atom positions during simulation

(f) Shift atoms to make imbalance dynamic

Figure 4.2: Introducing load imbalance in CoMD

Other than the borders between processes, another special case in molecular dynamics simulations is the border of the simulation space itself. In the case of CoMD, all borders are periodic in that they wrap around to the opposite side of the simulation space. For example, in a simulation with a $3 \times 3 \times 3$ grid (X,Y,Z values of 0-2), atoms in cells with an X value of 0 would interact with atoms in cells on their 'right' with an X value of 1 and atoms in cells on their 'left' with an X value of 2 since the X values would wrap around to the other side of the simulation space.

The initial implementation of CoMD is designed to minimize load imbalance by default. This is done by dividing the simulation space as evenly as possible between MPI processes and having all of the atoms in the simulation space placed randomly but with a uniform density. In previous work from Pearce et. al. [44], CoMD was extended such that an initial load balance could be introduced by removing atoms from the simulation. They achieved this

by creating spherical voids in the simulation space within which all atoms would be deleted. The load imbalance could be adjusted by changing the diameter and placement of these spherical voids. Figure 4.2 shows a simplified version of what the progression of this cutout procedure looks like on a two dimensional grid for a problem with four processes with four cells each. The thin black lines denote cell boundaries, the thick black lines represent process boundaries, the green circles represent atoms, and the blue circles represent the spherical voids introduced to create a load imbalance. In this particular example, processes 1 and 2 would have significantly fewer atom interactions to calculate, while process 3 would have a lesser reduction of work and process 0 would still have the same number of local atoms. Due to the periodic nature of CoMD, all of the processes would be affected by the introduction of these spherical voids but some more than others as shown in Figure 4.2d. Pearce et. al. [44] provided several user-specified runtime parameters to allow users to manipulate how the spherical voids operate such as: 1) the spherical void size, 2) the sphere count, and 3) a random seed for generating the coordinates for the sphere center.

I was also able to ensure that the GPU processes were given enough work for all of the SMs to have at least some work to do by providing a floor to the atom removal due to the spherical cutouts. This floor value is a tunable parameter, and essentially, once the total number of atoms drops below a certain floor, no more atoms will be removed. This floor value is not a hard constraint since the removal stops once the number of atoms goes under the floor value and can be as much as 64 atoms under the floor.

4.4 Design and Implementation

4.4.1 Introducing RAJA

For this work, I updated CoMD to use the RAJA [47] portability layer. RAJA allows for implementation of C++ lambda functions that can use a variety of computational backends common on HPC systems such as OpenMP [40] and CUDA [39]. In this way, a user can write a computational kernel once and then run it on a variety of architectures depending on which backend is chosen at runtime.

In my implementation, I used several different RAJA policies for the various computational kernels in both the CPU and GPU policy sets. For CPU-based runs, I used the standard sequential policy for all outer loops, and the SIMD policy for the innermost loops in order to provide vectorization hints to the compiler for those inner loops. Things get a bit more complicated for the GPU kernels using the CUDA backend.

In the CUDA programming model, blocks and threads can be organized into one, two, or three-dimensional grids with blocks/threads having indices in the X, Y, and Z dimensions. Since most of the computational kernels used two nested loops, I spread the work of the outer loop across CUDA blocks in the X dimension, and the work of the inner loop across CUDA threads in the X dimension. Essentially, the loop iterations were assigned in a round-robin fashion to the blocks and threads. For some of the kernels, the outer loop didn't have enough iterations to saturate the GPU, so I fixed the block size to a smaller number of threads to allow for more parallelism at the cost of dense blocks. The increased parallelism improved the performance for these kernels.

The most computationally intensive kernel for CoMD is the one that calculates the force interactions between the atoms. This kernel consists of four nested loops, so initially, I spread the work of the first loop across blocks in the X direction, the work of the second loop across blocks in the Y direction, the work of the third loop across threads in the X direction, and the work of the fourth loop across threads in the Y direction. This initially sounds like a good idea but based on the way the CoMD data structures are set up, and how these loops are organized, this causes an extremely imbalanced implementation of this kernel.

4.4.2 Refactoring for More Optimal GPU Usage

Reorganizing the Force Kernel

The CoMD data structures are set up such that there is a grid of boxes with a fixed maximum number of atoms they can contain which is 64. Without removing atoms, the typical number of atoms per box is about 18 on average (though it can be as high as 32) and does not change much throughout the simulation. The size of these boxes are such that the atoms within a box could only potentially interact with atoms in immediate neighbor boxes due to the

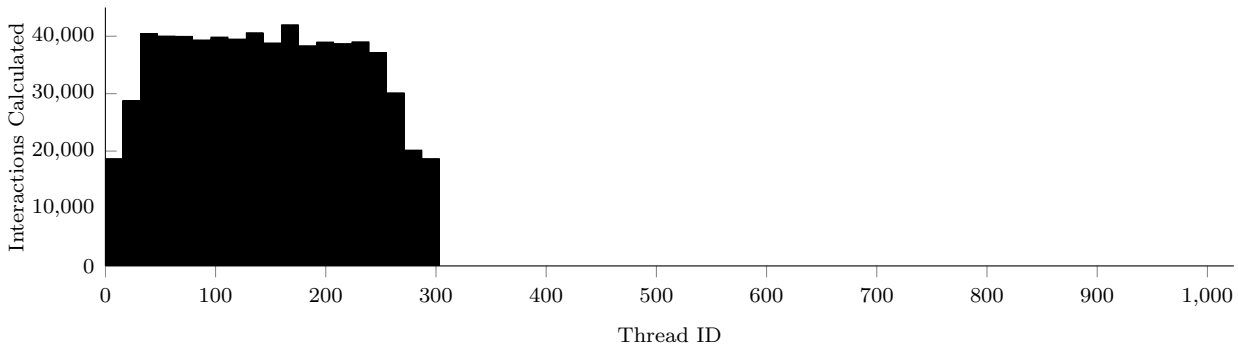


Figure 4.3: The average number of interactions per thread across all of the threablocks in the original implementation of the GPU-based CoMD.

radius of influence for force calculations. With that being said, the four loops for the force calculations iterate over: local boxes, neighbor boxes of the current local box, atom slots within the current local box's possible 64 atoms, atom slots within the current neighbor box's possible 64 atoms. The two inner-most loops must have a fixed iteration count in this particular instance due to a limitation with how loop indices can be written in RAJA policies and how the data structures are set up in CoMD.

In my initial implementation, the first two loops over local and neighbor boxes have their iterations spread out over CUDA blocks in the X and Y directions on the CUDA block grid, respectively. The local boxes loop has on the order of thousands of iterations, and the neighbor boxes loop always has a fixed number of iterations because each box has exactly 27 neighbor boxes. The third and fourth loops over local and neighbor atom slots would have their iterations spread out amongst the CUDA threads in the X and Y directions on the CUDA thread grid, respectively. Since each box has a fixed 64 atom slots, these two loops will always have 64 iterations. This creates a problem, because using 64 threads in the X and Y directions would result in $64 \times 64 = 4096$ threads, and the maximum number of threads in a block is 1024. Thus, the RAJA runtime reduces these dimensions to $64 \times 16 = 1024$ and unrolls the loops to allow for this configuration. Even though each loop iteration signifies an atom, the units of work here are atom interactions, where the first iteration would be the interaction between the first atom in the first local box and the first atom in the first neighbor box.

The boxes typically have less than 20 atoms in them and contain at most 32 atoms out of the fixed 64 atom slots. Since 64 threads are assigned to the interactions between atom slots in the local box, and there are never atoms in the second half of those slots, the threads with an X index of 32 or higher (half of the threads) will never be assigned any work. In addition, the way that RAJA unrolls the loops and the fact that most boxes have 24 or fewer atoms in them results in the work being further concentrated on the threads with lower thread IDs. Figure 4.3 shows that all of the work is assigned to approximately the first 300 threads and no work is given to any of the other threads. Since I am conducting an imbalance study on what is by default an extremely balanced application on the CPU, I decided to reorganize the way atoms are processed to be more balanced on the GPU.

To make this implementation more balanced, I first changed the unit of work in these loops from atom interactions to atoms, so one thread would perform all of the interactions between this atom and all other atoms within its radius. This has the added benefit of reducing the amount of potential collisions for atomic operations for updating shared data structures. Next, I packed all of the atoms from the local boxes on an MPI process into chunks of 1024 atoms each since I knew that the CUDA kernel would launch with 1024 threads per block. I then used a simplified bin packing algorithm to spread the chunks evenly across 80 bins, one for each SM on the V100 GPUs I used [29]. Since all of my chunks will be the same size except possibly the last one, I simply assigned chunks to bins in a loop from zero to the number of bins. I then changed the RAJA loop structure to have one outer loop for the bins and one inner loop for the chunks assigned to that bin, which results in the RAJA backend launching a total of 80 blocks with 1024 threads each. Since the chunks are distributed evenly, the worst case is that there are a factor of 80 plus one chunks and that singleton chunk has only one atom in it. This becomes less of an issue the more chunks there are, as the overhead of having one additional chunk is amortized by the total number of chunks in that bin.

Unified Memory

One of the early design decisions when porting CoMD to using GPUs through RAJA was to use *unified memory* for keeping track of memory that is used on the GPU [39]. Essentially,

unified memory pages are managed by the CUDA runtime, which moves them between the CPU and GPU as necessary. This means that if a page is currently resident on the CPU and is accessed from the GPU, this would cause a GPU page fault and the memory would need to be moved to the GPU. Using unified memory allows me to minimize the code modifications to CoMD outside of reimplementing the compute kernels using RAJA.

In order to minimize unified memory page faults, I made sure that shared data structures stored in unified memory were only accessed on the CPU when absolutely necessary. In the final implementation, the only time the the data structures are moved to the CPU is when data needs to be communicated between processes via MPI, and to speed up this process, I used packing and unpacking kernels to prepare the memory for transfer and extract the memory from the network respectively.

4.4.3 Instrumentation and Profiling

In order to gather accurate profiling information for the various tests, I introduced a few different forms of instrumentation to the CoMD code along with data from NVIDIA's *nvprof* tool. The first level of profiling I added was the Caliper tool from LLNL [8]. Caliper allows for low-overhead source code annotation and can provide information from third-party tools such as *nvprof*. In this case, I am using Caliper primarily for its timing capabilities and connection to *nvprof* and *cupPTI*. I collected all of the timing information through Caliper and provided hints to the NVIDIA profiling tools to focus the profiling only on the portions of code of interest.

I also added my own custom instrumentation to keep track of the exact number of atoms on each process. I did this by counting the number of atoms processed on each CUDA thread and writing that information out to a file. With this information, I can get a sense of where the atoms were placed across a run with a fine granularity. This instrumentation was used to inform the changes I made to the organization of the force kernel and to verify that the initial new placement of atoms was reasonably balanced to start with in the new implementation.

4.4.4 Metric-Driven Load Balancing

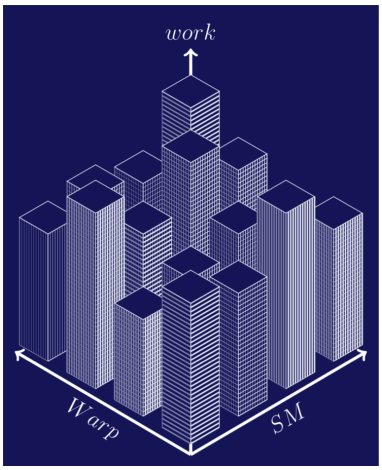


Figure 4.4: GPU efficiency: SM and warp usage.

GPU Parallelism

In this study, I am using NVIDIA GPUs through RAJA with the CUDA programming model on the back end, so it is important to understand how CUDA exposes the parallelism of the GPUs as discussed in Section 4.3.1. The various layers of parallelism within a GPU must be taken into account when reasoning about load balance between GPUs, which requires methods for identifying imbalance at each layer. The three primary levels of parallelism exposed through CUDA are the blocks, warps, and threads, and it is important to understand how these concepts translate to the GPU hardware.

The first level of parallelism is simple enough, the CUDA *blocks* are mapped onto SMs which are the first level of internal processing power in the GPU. Things get a bit more complicated when reasoning about how the warps and threads are mapped onto the hardware. For this study, I am using NVIDIA V100 GPUs which have 80 SMs, each with 32 FP64 cores, 64 FP32 cores, and 64 INT32 cores. CUDA threads are scheduled on these compute cores, and in NVIDIA GPUs the cores typically operate in lock-step to a certain extent, meaning that if there is divergence in the threads, each path is executed separately with threads

not involved in the current path masked out for these instructions. Since I have configured CoMD to be performing FP64 operations, for simplicity, I can assume that effectively only one warp (32 threads) can be actively processed on an SM at a time. So, the parallelism of the warps comes in the form of context switching between warps to hide stalls for things like memory load operations. Thus, from a hardware perspective, there are only two main levels of parallelism: SMs for blocks and compute cores for threads.

GPU Metrics

To better understand what is happening inside the GPU at runtime, I am using NVIDIA's *nvprof* tool which provides a number of metrics about the GPU. I have decided to use two particular metrics from nvprof: *sm_efficiency* and *warp_execution_efficiency*.

The *sm_efficiency* metric is defined as "the percentage of time at least one warp is active on a multiprocessor averaged over all multiprocessors on the GPU" in the CUDA documentation [39]. What this means is that this metric provides an estimate of the percentage of time that each SM was performing work on average. It is worth noting that as the warps complete their work and there are fewer warps to schedule on the SM, there will be fewer opportunities to hide stalls with context switches, so towards the end of a block's execution on an SM there could actually be more idle time due to stalls, which is not captured by this metric. In addition, this metric is an average across all SMs, so it can hide outliers and will not be able to provide a measure of load balance at this level of parallelism in the GPU.

The *warp_execution_efficiency* metric is defined as the "ratio of the average active threads per warp to the maximum number of threads per warp..." in the CUDA documentation [39]. This metric provides an estimate of the parallelism achieved in each warp, which essentially translates to the percentage of time work was being done within each warp on average. More recent CUDA architectures have relaxed the intra-warp lock-step constraints, but thread divergence can still cause a decrease in the average active threads per warp. Like *sm_efficiency*, this metric is an average so it can hide outliers and is unable to provide a measure of load balance at this level of parallelism.

These two metrics provide a coarse-grained estimate of the idle time at the two main levels of parallelism within the GPU hardware. Figure 4.4 visualizes what the distribution of work would look like within a GPU, and in the context of this figure, the metrics reveal the average difference between the intra-SM local maxima and the inter-SM global maximum bar height (*sm_efficiency*), and the average utilization within each bar (*warp_execution_efficiency*), but not the differences in the bar heights within each SM.

To estimate the percentage of time that the GPU is performing work, I can estimate the volume under the surface in Figure 4.4 as the following:

$$GPU_{eff} = SM_{eff} \times Warp_{eff} \tag{4.3}$$

where 100% efficiency would indicate full utilization of the GPU. This approximation is equivalent to the trapezoidal rule.

Adjusted Load Imbalance Formula

Dr. Olga Pearce of Lawrence Livermore National Laboratory has proposed applying the same principle from Equation 4.2 to the load within a GPU in order to remove the idle time within a GPU's internal computation. The idea is to use some metric of GPU efficiency, which I have provided in Equation 4.3, to adjust the wall time spent executing a kernel:

$$Time_{adj} = Time_{measured} \times GPU_{eff} \tag{4.4}$$

and apply this new measure of load to the original definition of load imbalance (Equation 4.1) which results in the following:

$$Imbalance = \left(\frac{Ladj_{max} - Ladj_{ave}}{Ladj_{ave}} \right) \times 100\%. \tag{4.5}$$

This load imbalance formula is designed to bring the *load* (time) imbalance more in line with the *work* imbalance of the application. Essentially, the idea is to remove noise added to the execution time via stalls and idle time due to the implementation of the application and the scheduling of blocks, warps, and threads in the CUDA runtime.

4.5 Results and Analysis

4.5.1 Experimental Setup

Test Machine

The following experiments were executed on *Lassen*, a 23-petaflop supercomputer at LLNL comprised of 795 nodes with two 22-core IBM Power9 CPUs, four NVIDIA Tesla V100 GPUs, and 256GB of DDR4 memory per node, with a Mellanox 100Gbps EDR InfiniBand interconnect. Each NUMA node is associated with two of the V100 GPUs, and I ensured that the MPI processes were bound to the correct CPU to maximize bandwidth and minimize latency between CPU and GPU.

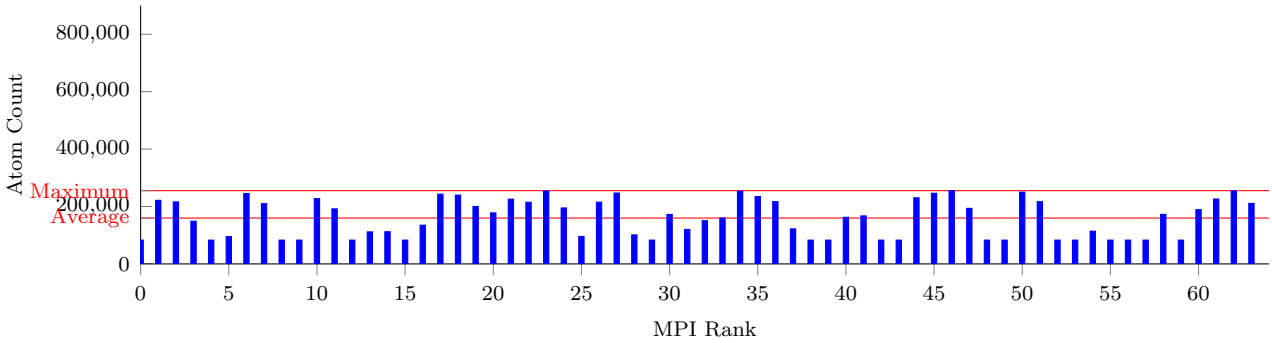
Instrumentation Setup

I added three different types of instrumentation to the CoMD implementation for these experiments: the Caliper [8] tool for precise timing information; my own custom instrumentation in the compute kernels to count atoms processed at the granularity of CPU cores and GPU threads; and NVIDIA's *nvprof* tool for GPU metrics. With consistent atom placement and removal across runs, I was able to perform separate runs for timing, *nvprof* instrumentation, and the custom instrumentation in order to minimize noise. For the timing information, I am only timing the most compute intensive kernel, the force calculation kernel, which makes up the majority of the execution time. So, when I discuss measured time throughout this section, I mean the time spent performing the force kernel on a given process.

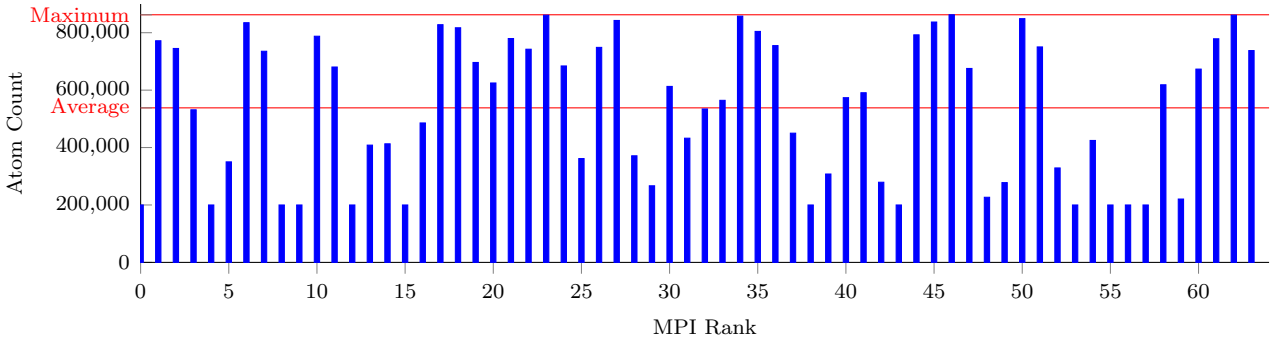
CoMD Problem Size

I ran my experiments with 64 MPI processes across 16 nodes with one MPI process per GPU bound to a single core of the appropriate CPU. In order to keep the MPI performance consistent between runs using the CPU vs. the GPU for computation, I use the same MPI process binding for both tests.

In previous work [44], it has been demonstrated that in CoMD the execution wall time closely correlates with the number of atoms. For this reason, I am using atoms as my metric



(a) Small problem size (256K ceiling and 80K floor atoms per process).



(b) Large problem size (864K ceiling and 200K floor atoms per process).

Figure 4.5: Atom removal histograms for 60% atom imbalance problems for both small and large amount of work per GPU.

for *workload* on a given MPI process. I chose to use two problem sizes for these experiments: one with a small amount of work per process and one with a large amount of work per process. For the smaller problem size, I set the minimum number of atoms per process to approximately 80,000, which results in the majority of CUDA threads having at least one atom to process, and the maximum number of atoms per process to approximately 256,000 atoms. I chose these bookends to allow for some threads to have nothing to do, but not many. For the larger problem size, I set the minimum number of atoms per process to approximately 200,000, and the maximum number of atoms per process to approximately 864,000 atoms. I chose these bookends to get to a point where the GPU is saturated for a significant portion of the execution.

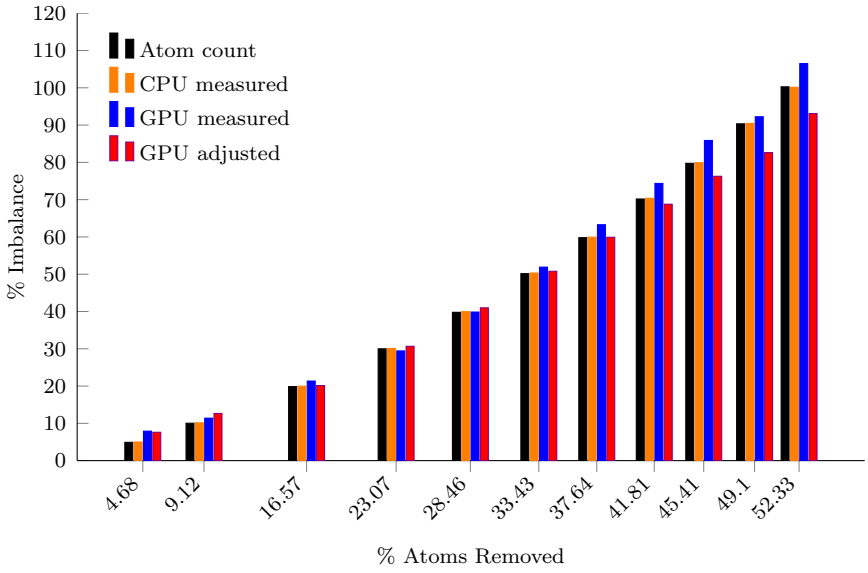


Figure 4.6: Imbalance percentage for each amount of atom removal for the small test case (256K ceiling and 80K floor atoms per process). Note: The *Atoms* values are the ground truth for work imbalance.

Initial Imbalance

I used the cutout feature from previous work by Pearce et. al. [44] to introduce a variety of different initial load imbalances in CoMD. I adjusted the sizes and placement of the spherical voids to create initial atom imbalances of approximately 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100 percent. These different initial imbalances produce similar results, so I will be focusing primarily on the 60% imbalance case for the two different problem sizes. Figure 4.5a shows the breakdown of the number of atoms on each MPI process with indicators for the maximum and average atom count for the small problem size, while Figure 4.5b shows the same but for the large problem size, both with an imbalance of 60%.

Datasets

In order to control for variation due to the random initialization of atoms by CoMD, I decided to use a fixed random seed for creating the two initial problems for the small and large problem size. I then created two separate datasets from each of these two initial problems by adjusting the parameters for the cutout voids to create 11 separate problems

from each of the initial two problems. Thus, there are a total of 22 different problems across two datasets. I always used exactly 10 spherical voids and centered them in exactly the same place for all problems. I only changed the radius of these spheres to change the initial load imbalance. This way, the processes with fewer atoms are always the same across problems, and the MPI communications and internal GPU imbalance are modified similarly across problems.

Figure 4.6 illustrates the 11 problems for the data set with a small amount of work per process. The x axis shows the percentage of atoms removed, and the y axis shows the *work* imbalance in remaining atoms, the *load* imbalance as measured when running the problems on the CPUs or GPUs, and the adjusted *load* imbalance metric for GPU-based computation. As more atoms are non-uniformly removed, the work imbalance in the problem increases.

Information Collected

Throughout this study I will be focusing on four main data points: atom count, CPU measured time, GPU measured time, and GPU adjusted time. The atom count is used for calculating the ground truth of the *work imbalance* since atoms are my measure of work. The measured wall time spent in CPU execution is meant to provide as a control group of a well understood case for *load imbalance* between MPI processes and serve as a verification of previous results. The measured wall time spent in GPU execution indicates the interprocess *load imbalance* of a GPU-based simulation without taking into consideration the intra-GPU load balance. The GPU adjusted time uses Equation 4.5 to update the load imbalance with some consideration of the intra-GPU load balance to alleviate the noise introduced by switching to a GPU-based execution.

4.5.2 Overview and CPU Verification

Overview

Table 4.1 provides an overview of the 22 separate problems across the two datasets with Table 4.1a showing the data for the small problem size and Table 4.1b showing the data for the large problem size. The interactions listed in these tables are the same interactions

Table 4.1: Detailed parameters and statistics for both the small and large problem sets.

(a) Small amount of work per process (floor 80K, ceiling 256K atoms).

Prob -lem	% atoms removed	# atoms remaining	# interactions simulated	Spherical voids num	Spherical voids radius	%imbal atoms	%imbal inters	%imbal CPU meas	%imbal GPU meas	%imbal GPU adj
1	4.7	15,617,860	4,132,620,520	10	60	4.91	5.46	5.00	7.91	7.61
2	9.1	14,889,652	3,926,773,538	10	75	10.04	10.98	10.15	11.40	12.65
3	16.6	13,669,392	3,584,852,799	10	92.5	19.86	21.57	19.96	21.35	20.13
4	23.1	12,604,115	3,289,583,127	10	104.5	29.99	32.48	30.07	29.43	30.67
5	28.5	11,720,591	3,039,418,540	10	114	39.79	43.38	39.96	39.86	40.99
6	33.4	10,906,225	2,807,628,009	10	123	50.15	54.98	50.32	51.91	50.79
7	37.6	10,216,908	2,615,513,545	10	131	59.81	65.61	59.94	63.28	59.91
8	41.8	9,533,211	2,422,236,329	10	139	70.18	77.39	70.36	74.35	68.78
9	45.4	8,944,387	2,255,092,428	10	146	79.75	88.49	79.90	85.88	76.27
10	49.1	8,339,459	2,090,266,717	10	153	90.34	100.42	90.45	92.24	82.62
11	52.3	7,810,298	1,943,529,301	10	159	100.27	111.23	100.15	106.49	93.09

(b) Large amount of work per process (floor 200K, ceiling 864K atoms).

Prob -lem	% atoms removed	# atoms remaining	# interactions simulated	Spherical voids num	Spherical voids radius	%imbal atoms	%imbal inters	%imbal CPU meas	%imbal GPU meas	%imbal GPU adj
1	4.7	52,710,379	12,698,722,571	10	90	4.91	5.27	4.95	7.58	8.00
2	9.0	50,319,214	12,097,045,030	10	112	9.89	10.50	9.90	12.94	12.95
3	16.8	46,008,721	11,019,695,672	10	139	20.19	21.31	20.08	24.15	22.89
4	23.0	42,559,982	10,164,195,089	10	156	29.92	31.52	29.73	30.35	31.04
5	28.5	39,549,154	9,422,710,951	10	169	39.82	41.87	39.44	40.31	40.66
6	33.4	36,840,543	8,742,594,337	10	180.5	50.08	52.83	49.58	50.40	50.13
7	37.7	34,448,771	8,142,891,813	10	191	60.24	63.69	59.55	61.92	60.77
8	41.5	32,363,490	7,622,638,462	10	200.5	69.97	74.12	69.12	67.76	67.03
9	45.0	30,390,629	7,125,268,667	10	209.5	80.03	85.10	78.95	80.06	74.44
10	48.2	28,623,431	6,690,383,093	10	217.5	89.85	95.63	88.53	86.38	82.24
11	51.3	26,956,716	6,281,384,208	10	225	99.92	106.48	98.29	98.44	92.35

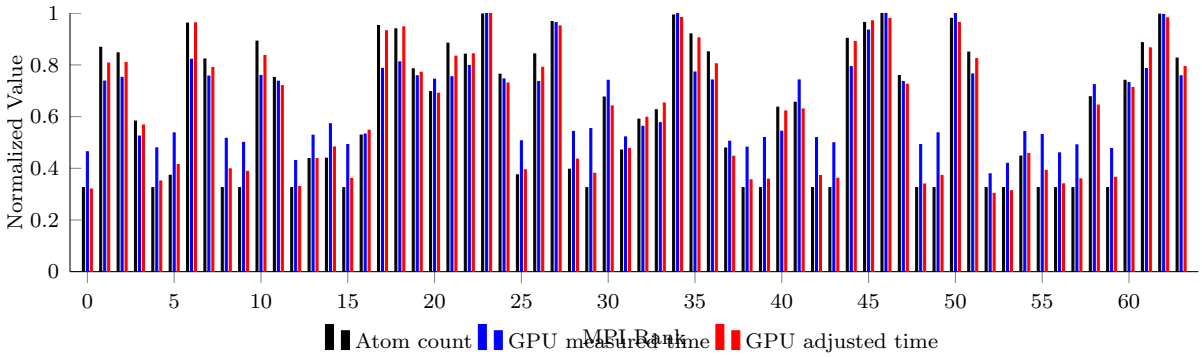
described in Section 4.3.3. It is worth noting that the initial work imbalance in the atoms is not exactly at the target test imbalances (5, 10, 20, etc...) for each problem, but is reasonably close. This is due to the fact that creating an exact desired imbalance with the spherical voids is non-trivial.

One of the major differences between the CPU and GPU to keep in mind is that of their optimal memory access patterns. On a CPU, sequential accesses are ideal because of how the caches are implemented, whereas on a GPU strided accesses with a step size of the number of threads in a warp and taking into account GPU memory banks are ideal since memory loads pull enough data for all threads in a warp or half warp meaning one memory load could service all threads in the ideal case. This means that the way this code is implemented with all of the interactions for a given atom handled by a single thread accessing all nearby atoms sequentially is more optimal for the CPU than the GPU and would cause a lot of stalls for memory operations in the GPU implementation.

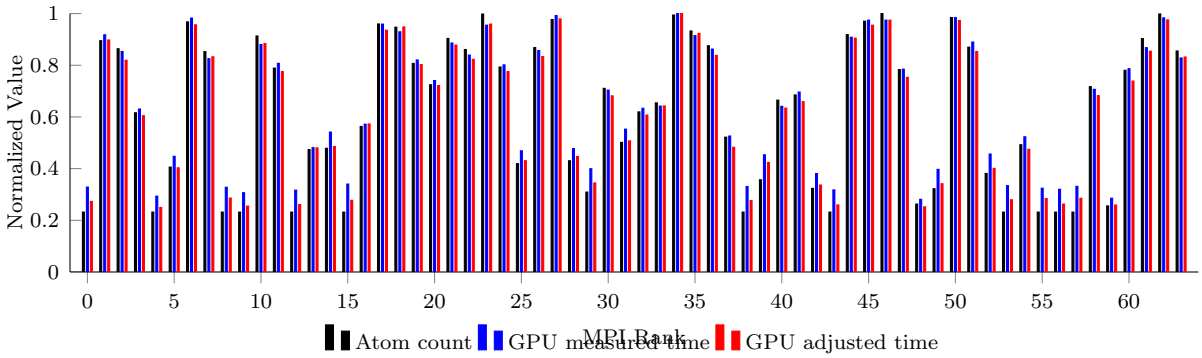
Table 4.1 verifies the premise of this study that the measured load imbalance of the GPU does not match the work imbalance. The load imbalance shown on the GPU shows the same general trend of the work imbalance, but is not nearly as accurate as the CPU load imbalance.

CPU Verification

In order to have a solid baseline, I wanted to verify the previous work [44] and show that the load imbalance as measured on the CPU-based implementation lines up with the work imbalance of the number of atoms per process. Table 4.1 shows that the measured load imbalance on the CPU matches the atom work imbalance very closely with most problems being within 1%. This verifies the notion that there is a strong correlation between the simulation workload (atoms) and the application run time. Thus, one would expect that the run time of the GPU should also match the simulation workload.



(a) Small test case (256K ceiling and 80K floor atoms per process).

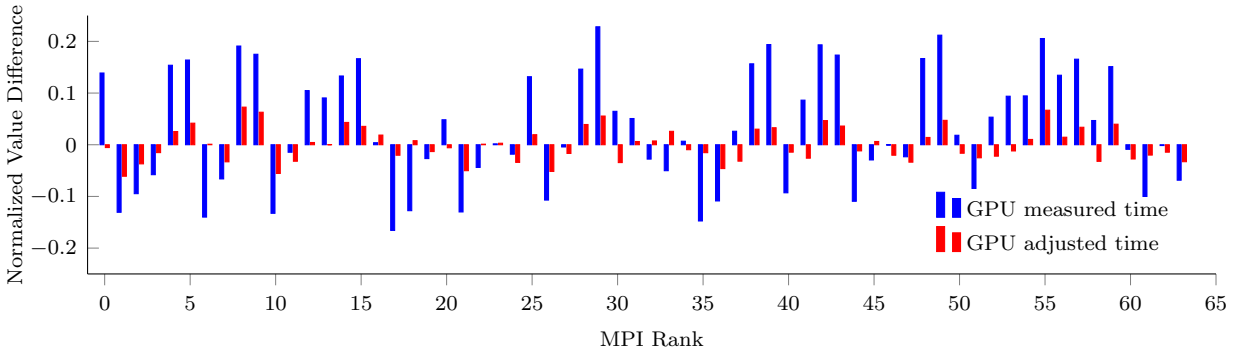


(b) Large test case (864K ceiling and 200K floor atoms per process).

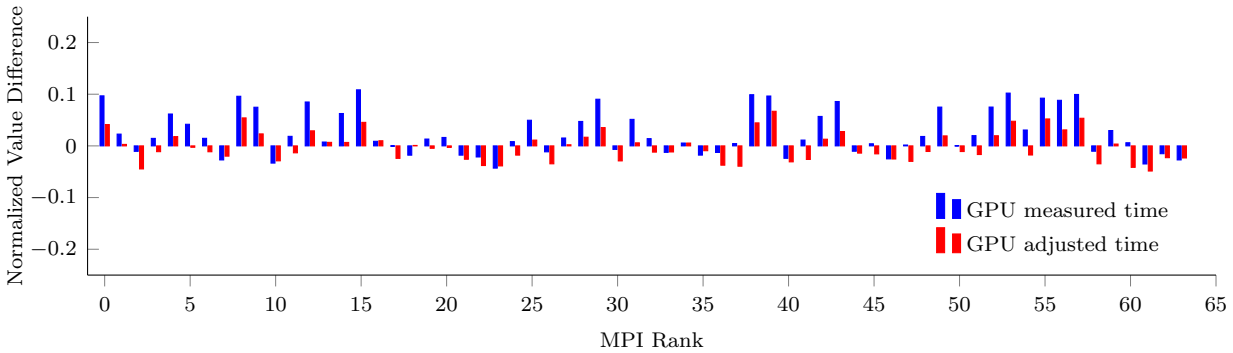
Figure 4.7: Measured and adjusted time vs atom values with 60% atom imbalance for both small and large test cases. All values are normalized by the maximum of their respective data sets.

4.5.3 Comparing Measured GPU Load and Adjusted GPU Load

The main issue at the heart of this study is that even though there is a near perfect match between application workload (atoms) and run time in the CPU-based implementation, the same cannot be said for the GPU-based implementation. One of the key differences between the load (measured time) values for the CPU and GPU is that the load on a CPU process only contains a single thread whereas a GPU process contains all of the nested parallelism inherent to a GPU. So, unless the GPU-based implementation is perfectly balanced and causes little to no interruption to computation due to stalls, there will be an internal load imbalance and additional noise from stalls on the GPU. So, in Section 4.4.4 I introduced some GPU metrics and a proposed update to the traditional load imbalance formula designed to



(a) Small test case (256K ceiling and 80K floor atoms per process).



(b) Large test case (864K ceiling and 200K floor atoms per process).

Figure 4.8: Measured and adjusted time value differences with atom values. Using 60% atom imbalance for both large and small test cases. All values are normalized by the maximum of their respective data sets.

adjust the load (run time) by an efficiency metric in order to better reflect the work (atoms) imbalance.

Measured vs. Adjusted Load Across Ranks

In Figure 4.7, I show histograms of the atom count, GPU measured execution time, and GPU adjusted execution time for the 60% imbalance versions of the small and large problem sizes with all values normalized to have unified units. In this figure, it is clear that both GPU metrics generally follow the atom count, however the adjusted metric tends to follow the atom count more closely. This is particularly true for the small problem size where there is more opportunity for internal load imbalance on the GPU since the GPU is not saturated with work and there is a larger proportional gap in the number of atoms per thread.

Figure 4.8 shows the same comparison, but plots the difference between the normalized values for the GPU data points (measured/adjusted time) and the normalized atom count. For the small problem size, Figure 4.8a illustrates that the adjusted load is indeed closer to the amount of work performed, or within 7%, while the measured load can be as far off as 23%. For the large problem size, the GPU has significantly more work overall which mitigates the internal GPU load imbalance to some extent. This is shown in Figure 4.8b, which still shows the same trend of adjusted load being closer to the amount of work performed, with adjusted load again within 7%, while the measured load can be as far off as 11%.

GPU Load Imbalance vs. Work Imbalance

I have demonstrated that the adjusted GPU load better matches the atom count values when normalized against their respective maximum values, but how does the adjusted time metric affect the closeness of the matching between the calculated work imbalance and load imbalance. If I refer back to Figure 4.6 and Table 4.1, neither the measured GPU load imbalance, nor the adjusted GPU imbalance closely match the work imbalance. In fact, the adjusted load imbalance appears to perform worse than the measured GPU load imbalance in several cases.

Correlation of Load and Work

For CoMD, the application work (atoms) should translate directly to application load (time), however this is not the case for the GPU-based approaches. Indeed, even when the GPU load is adjusted by an efficiency metric to help remove the noise introduced by internal GPU imbalance, the calculated load imbalance does not match the work imbalance well. In order to see how well the adjusted load reflects the application work, I decided to do an analysis of the Pearson correlation of the measured and adjusted GPU load values and the atom count work values. The Pearson correlation coefficient provides a metric of the strength of the linear relationship between two variables, and is important in this context because it shows whether the adjusted time metric correctly captures the relative loads per process [6]. The better a metric is at capturing the relative loads per process, the more accurately a load balancing scheme can correct the load imbalance based on this metric.

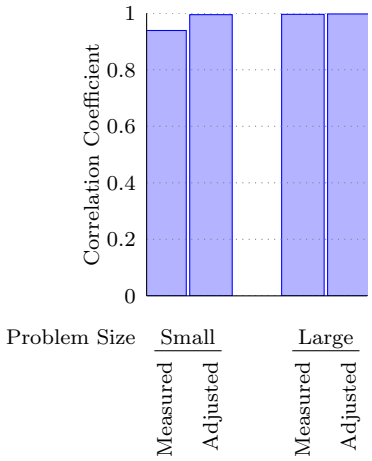
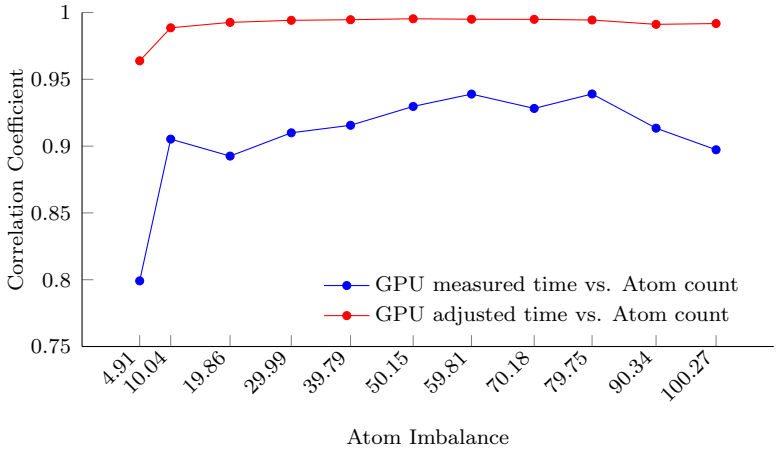


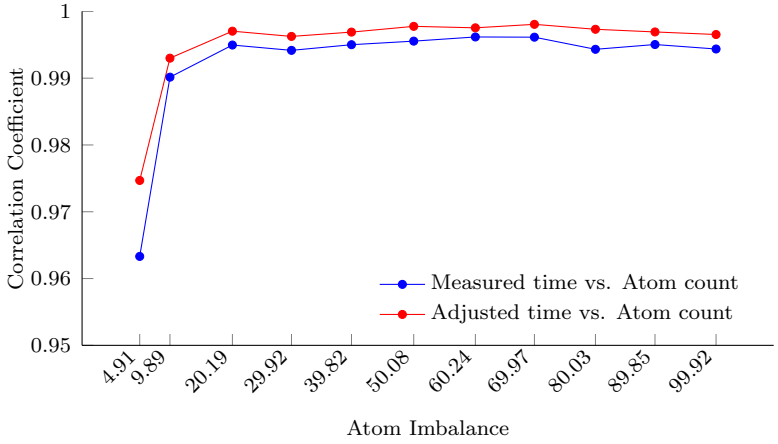
Figure 4.9: Correlation coefficients of small and large problem sizes for the 60% imbalance case.

Throughout this study, I have been focusing on the 60% imbalance case for both the small and large problem sizes. Figure 4.9 shows the Pearson correlation coefficients of measured and adjusted load when compared to the atom counts for the 60% imbalance problems. Both the measured and adjusted loads have a strong positive correlation with the atom count; however the adjusted metric has a higher correlation. The adjustment improves the correlation coefficient from 0.93895349 to 0.9949787 in the case where there is less work per process, and from 0.99616814 to 0.99753709 in the case of more work per process.

Figure 4.10 provides a more general look at the correlation coefficients across all of the different problems with Figure 4.10a focusing on the small problem size and Figure 4.10b focusing on the large problem size. This shows that across the board, the adjusted load metric provides a better correlation with the application workload than the measured load. For the small problem size, where there is more opportunity for load imbalance within the GPU, there is a 5.9% to 20.61% improvement in the correlation coefficient. Again, when the GPU is more saturated with work, the internal GPU load imbalance is mitigated better, yet there is still an improvement in correlation. The improvement is to a lesser degree, with only a little over a 1% improvement in the best case.



(a) Small test case (256K ceiling and 80K floor atoms per process).



(b) Large test case (864K ceiling and 200K floor atoms per process).

Figure 4.10: Correlation coefficients of measured/adjusted load (time) compared to workload (atom count) for the various initial atom imbalances for both small and large test cases.

4.6 Conclusions

I extended a proxy application with a well-characterized correlation of work to simulation run time with the RAJA portability layer, which enabled an evaluation of load imbalance on GPUs. I devised a metric for estimating the efficiency of a GPU's execution using two low-level GPU metrics from the *nvprof* tool and applied that to an extension to the standard load imbalance formula proposed by Dr. Olga Pearce. I was then able to evaluate the effectiveness of this new imbalance formula when applied to this extended proxy application and show an improvement of up to 20.61% correlation between application work in the form of atoms and load in the form of adjusted run time. This improvement is shown with a GPU-based implementation that is extremely balanced within the GPU to alleviate stalls due to synchronizations.

Chapter 5

Conclusions

5.1 Conclusions

Performance tools in the High-Performance Computing field rely on having access to low-level performance metrics in addition to timing information to conduct performance analysis of distributed applications. The MPI programming paradigm has been the de facto standard approach to programming in a distributed environment, and GPU accelerators have become more dominant as a computational platform recently, so both of these technologies are ideal candidates for showcasing the usefulness of low-level performance metrics. Thus, this study contributes to the performance analysis field by introducing new metrics and analysis approaches to the MPI and GPU paradigms. In summary, this study provides the following contributions to distributed performance analysis:

- **A Tool for Tracking Internal MPI Metrics:** The PMPI interface remains predominant approach to MPI performance analysis today, however it provides only a limited view of the highest level of MPI performance information by overloading MPI function calls. There have been efforts over the years to expose internal MPI performance information such as the Peruse interface [31], and recently the MPI_T interface was added to the MPI Standard to allow for exposing such internal information [20]. This study introduces a companion approach to the MPI_T interface called Software-based Performance Counters (SPCs) in the Open MPI implementation

of the MPI Standard in Chapter 3. With SPCs, I am able to track a wealth of internal MPI performance metrics across many different aspects of an MPI implementation's performance such as collective algorithm usage, fine-grained data transfer information, and detailed internal queue usage.

- **A Method for Exposing Internal MPI Metrics to Tools:** Collecting performance data is not terribly useful without a method for exposing this collected data to end users. In Chapter 3 I introduce several methods for accessing SPCs such as command line output, MPI_T performance variables, and the *mmap* interface. The *mmap* interface is my addition and adds a method for tools to gain direct read-only access to SPCs without having to pay the function overhead of using the MPI_T interface. With the *mmap* interface, the overhead of reading a counter is essentially the same as accessing an array element and does not have any effect on the collection of SPCs. The *mmap* interface is backed by a shared data file for storing counter data, and an XML file for describing the properties of the data file and providing offsets at which each counter is stored in the data file. These backing files persist after execution which allows for postmortem analysis of these counters, which is unavailable through the MPI_T interface. I have also provided a *snapshot* feature that allows for periodic copying of the data file in order to provide data for particular time slices of an application rather than just an overall summary.

- **A Demonstration of the Use Cases of Internal MPI Metrics:** I demonstrate several potential use cases for SPCs in Chapter 3 such as: identifying internal MPI implementation issues, MPI application analysis, and workload characterization. I demonstrated a problematic implementation of multithreaded MPI in Open MPI through a case study of out-of-sequence messages in a synthetic benchmark as well as in a multithreaded MPI application from MADNESS. SPCs were instrumental in identifying the issue of having far too many out-of-sequence messages when MPI was initialized with MPI_THREAD_MULTIPLE. I discussed a study in which SPCs were used to identify a bottleneck in an implementation of a local rollback algorithm for

fault tolerance. Finally, I conducted an example workload characterization study with the LAMMPS application and the *snapshot* feature of the *mmap* interface for SPCs.

- **A Testbed for Imbalance Studies:** In order to study the effects of load balancing algorithms or approaches to identify potential load imbalance, one must have an application to test out the new approach. There have been many load imbalance studies conducted on CPU-based applications, however there is a need for a test application with support for CPU-based and GPU-based computation. In Chapter 4 I introduce an extended version of the CoMD proxy application that uses RAJA as a portability layer for easy switching between CPU-based, GPU-based, and hybrid computation. This allows for future load imbalance studies to easily identify how their work is affected by changing between the CPU and GPU paradigms, or simply target a particular paradigm of interest.

- **An Efficiency Metric for Internal GPU Efficiency:** There are many metrics available through NVIDIA's *nvprof* tool, but none of them provides a complete view of the GPU efficiency during a CUDA kernel's lifetime. In Chapter 4 I combine two separate GPU efficiency metrics, *sm_efficiency* and *warp_execution_efficiency*, into a unified GPU efficiency metric. I do this by simply representing both of these metrics as a value between zero and one and multiplying them together. This is able to provide an estimate of the percentage of time that computation was being done throughout the kernel's execution.

- **An Analysis of a Proposed GPU Imbalance Formula:** With my GPU efficiency metric, I was able to evaluate a GPU load imbalance formula proposed by Dr. Olga Pearce. The key concept of this formula is to adjust the GPU execution time by an efficiency metric to create an adjusted time and use this adjusted time for load imbalance calculations. This is meant to take into account the internal imbalance of the GPU when calculating load imbalance across the distributed application. In Chapter 4, I show that this GPU imbalance formula allows for an improvement in the correlation between application work units and load in the form of time by up to 20.61%.

The initial version of the contributions from Chapter 3 have been published at EuroMPI/USA 2017. The current state of the work from Chapter 4 is in the process of being submitted to a Workshop at EuroPar 2020. I also contributed my expertise in performance analysis with SPCs to two different papers led by Dr. Thananon Patinyasakdikul: the introduction of the *Multirate* benchmark [43], and a study of the design of multithreading in MPI [42], accepted to the ExaMPI Workshop 2019 and CLUSTER 2019 respectively.

The initial implementation of SPCs that was introduced in [15] is incorporated into the Open MPI 4.0.0 release, and there is an active pull request to have the current version of SPCs included into the current master branch of the Open MPI repository for release in Open MPI 5.0.0. My RAJA implementation of CoMD, minus the bin packing optimization, has an active pull request to be incorporated into the RAJAProxies repository on GitHub.

5.2 Suggestions For Future Work

The work presented in this study addresses performance analysis of two major technologies in distributed applications development, MPI and GPUs. Both my MPI work in Chapter 3 and my GPU load imbalance work in Chapter 4 have the potential to provide for future performance analysis research in these areas.

5.2.1 Software-based Performance Counters

The work in Chapter 3 focuses on providing a baseline implementation of internal MPI metrics in Open MPI, including several methods for exposing those metrics to end users and tool developers. The implementation of SPCs has room for improvement and extension to provide better support for performance tool developers.

One of the areas where SPCs are currently lacking is in ease of use. The MPI_T interface can be cumbersome to use, and there is a lack of utility functions for facilitating use of the *mmap* and *snapshot* features of SPCs. Another area that would be ideal for further research is in incorporation of SPCs into the tool ecosystem through integration with existing performance analysis and data gathering tools.

SPC Utilities

Currently, the method for using SPCs exported through the *mmap* interface is to parse an XML file on each process which indicates some metadata about an SPC data file including offsets in the data file where counters are stored, then attach to that data file with the *mmap* function, and finally read the counters of interest. To assist with this, it would be useful to have a utility that can read and display counter values on the command line as well as a library to support parsing this data and delivering it to tools and application developers. Such utilities could potentially be extended to be able to handle a series of data files from the *snapshot* feature and convert them into a trace file which could be read by tools like Vampir.

Dynamic Open MPI Instrumentation with Dyninst

Currently, the SPC instrumentation in Open MPI is either enabled or disabled when Open MPI is compiled. This means that all of the added instrumentation must check whether a counter is enabled each time, even if the counter is disabled. This can add unnecessary overhead to the MPI library, though not much. One potential area for improvement would be to add the instrumentation dynamically with the Dyninst API. With this approach, the instrumentation could be added and removed throughout a program's execution, and only enabled counters would be added to the code.

PAPI Software-Defined Events Interface

The PAPI Software-Defined Events (SDE) interface would be a good candidate for adding SPCs to an existing tool. With some simple code additions to the SPC driver code, SPCs could be available through PAPI and would thus be immediately available to many performance tools due to PAPI's widespread adoption. I am currently working on providing SDE support inside my SPC driver code.

LDMS Integration

The Lightweight Distributed Metric System (LDMS) provides an interface for performing machine characterization across all of the major resources on the system. If SPCs were to be integrated with LDMS as a metric set, this internal MPI performance information could provide context to the other communication metrics across the system. I am currently discussing incorporating SPCs into LDMS in some capacity with the LDMS team at Sandia National Laboratory.

5.2.2 GPU Load Imbalance

The work in Chapter 4 primarily focuses on analyzing a metric-based extension to the traditional load imbalance formula to remove some of the noise introduced by the internal load balance of the GPU and the stalls inherent to how work is scheduled and run within a GPU. Using more precise internal load information, such as application-specific instrumentation, for calculating the internal GPU efficiency could further improve the extended load imbalance formula. Future work could also focus more on improving the accuracy of the load balance calculation with respect to the work imbalance after using the proposed load balance formula extension to help alleviate noise.

The concept of adjusting execution time by an efficiency metric could potentially be applied more generally to any form of hierarchical parallel application. For example, imagine an application is run at a large scale with tens of thousands of MPI processes, and each of those processes represents dozens of threads, and potentially some of those threads are associated with accelerators. In this instance, the load imbalance at a given level of this hierarchy could be influenced by the internal imbalance of all subsequent layers. Thus, an efficiency metric could be applied to each layer where parallelism is combined into one entity, such as the threads within a process, to estimate the internal load of this combined entity and factor that into the imbalance calculations.

Bibliography

[1] Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., and Tallent, N. R. (2010). Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701. 14, 21, 22

[2] Agelastos, A., Allan, B., Brandt, J., Cassella, P., Enos, J., Fullop, J., Gentile, A., Monk, S., Naksinehaboon, N., Ogden, J., et al. (2014). The lightweight distributed metric service: a scalable infrastructure for continuous monitoring of large scale computing systems and applications. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 154–165. IEEE. 14

[3] Armstrong, W., Christen, P., McCreath, E., and Rendell, A. P. (2006). Dynamic algorithm selection using reinforcement learning. In *2006 International Workshop on Integrating AI and Data Mining*, pages 18–25. 12

[4] Balaji, P., Buntinas, D., Goodell, D., Gropp, W., and Thakur, R. (2008). Toward efficient support for multithreaded mpi communication. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 120–129, Berlin, Heidelberg. Springer-Verlag. 32

[5] Benedict, S., Petkov, V., and Gerndt, M. (2010). Periscope: An online-based distributed performance analysis tool. In *Tools for High Performance Computing 2009*, pages 1–16. Springer. 13, 16

[6] Benesty, J., Chen, J., Huang, Y., and Cohen, I. (2009). Pearson correlation coefficient. In *Noise reduction in speech processing*, pages 1–4. Springer. 102

[7] Bernholdt, D. E., Boehm, S., Bosilca, G., Gorentla Venkata, M., Grant, R. E., Naughton, T., Pritchard, H. P., Schulz, M., and Vallee, G. R. (2020). A survey of mpi usage in the us exascale computing project. *Concurrency and Computation: Practice and Experience*, 32(3):e4851. e4851 cpe.4851. vi

[8] Boehme, D., Gamblin, T., Beckingsale, D., Bremer, P.-T., Gimenez, A., LeGendre, M., Pearce, O., and Schulz, M. (2016). Caliper: Performance Introspection for HPC Software

Stacks. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'16)*, Salt Lake City, Utah, USA. 4, 14, 90, 94

[9] Brunst, H., Winkler, M., Nagel, W. E., and Hoppe, H.-C. (2001). *Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach*, pages 751–760. Springer Berlin Heidelberg, Berlin, Heidelberg. 2, 4, 13, 15

[10] Chabbi, M., Murthy, K., Fagan, M., and Mellor-Crummey, J. (2013). Effective sampling-driven performance tools for GPU-accelerated supercomputers. In *SC'13*. 22

[11] Danalis, A., Jagode, H., Herault, T., Luszczek, P., and Dongarra, J. (2019). Software-defined events through papi. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 363–372. IEEE. 33

[12] Daw, M. S. and Baskes, M. I. (1984). Embedded-Atom Method: Derivation and application to impurities, surfaces, and other defects in metals. *Phys. Rev. B*, 29:6443–6453. 83

[13] Daw, M. S., Foiles, S. M., and Baskes, M. I. (1993). The Embedded-Atom Method: A review of theory and applications. *Materials Science Reports*, 9(7):251 – 310. 83

[14] Dennard, R. H., Gaensslen, F. H., Rideout, V. L., Bassous, E., and LeBlanc, A. R. (1974). Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268. 24

[15] Eberius, D., Patinyasakdikul, T., and Bosilca, G. (2017). Using software-based performance counters to expose low-level open mpi performance information. In *Proceedings of the 24th European MPI Users' Group Meeting*, EuroMPI '17. 7, 109

[16] Elis, B., Yang, D., and Schulz, M. (2019). Qmpi: a next generation mpi profiling interface for modern hpc platforms. In *Proceedings of the 26th European MPI Users' Group Meeting*, pages 1–10. 18, 26

[17] Exascale Computing Project (2020). HPCToolkit Documentation. `http://hpctoolkit.org/`. 14, 17

[18] ExMatEx Project (2012). Exascale Co-design Center for Materials in Extreme Environments (ExMatEx). 83

[19] Farooqui, N., Kerr, A., Diamos, G., Yalamanchili, S., and Schwan, K. (2011). A framework for dynamically instrumenting gpu compute applications within gpu ocelot. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 9:1–9:9, New York, NY, USA. ACM. 22

[20] Forum, M. P. I. (2015). *MPI: A Message-Passing Interface Standard Version 3.1*. `http://mpi-forum.org/`. 1, 2, 3, 17, 18, 24, 26, 27, 30, 39, 106

[21] Frenkel, D. and Smit, B. (2001). *Understanding Molecular Simulation*. Academic Press, Inc., Orlando, FL, USA, 2nd edition. 83

[22] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. (2004). *Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation*, pages 97–104. Springer Berlin Heidelberg, Berlin, Heidelberg. 3, 5, 31

[23] Geimer, M., Wolf, F., Wylie, B. J. N., Ábrahám, E., Becker, D., and Mohr, B. (2010). The scalasca performance toolset architecture. *Concurr. Comput. : Pract. Exper.*, 22(6):702–719. 13, 21

[24] Haidar, A., Jagode, H., Vaccaro, P., YarKhan, A., Tomov, S., and Dongarra, J. (2018). Investigating power capping toward energy-efficient scientific applications. *Concurrency Computation: Practice and Experience*, 2018:1–14. 11

[25] Harrison, R. J., Beylkin, G., Bischoff, F. A., Calvin, J. A., Fann, G. I., Fosso-Tande, J., Galindo, D., Hammond, J. R., Hartman-Baker, R., Hill, J. C., Jia, J., Kottmann, J. S., Ou, M.-J. Y., Pei, J., Ratcliff, L. E., Reuter, M. G., Richie-Halford, A. C., Romero, N. A., Sekino, H., Shelton, W. A., Sundahl, B. E., Thornton, W. S., Valeev, E. F., Álvaro Vázquez-Mayagoitia, Vence, N., Yanai, T., and Yokoi, Y. (2016). Madness: A

multiresolution, adaptive numerical environment for scientific simulation. *SIAM Journal on Scientific Computing*, 38(5):S123–S142. 61

[26] Hermanns, M.-A., Hjelm, N. T., Knobloch, M., Mohror, K., and Schulz, M. (2019). The mpi_t events interface: An early evaluation and overview of the interface. *Parallel Computing*, 85:119–130. 19

[27] Intel Corporation (2016). *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel. 53

[28] Jagode, H., Danalis, A., Anzt, H., and Dongarra, J. (2019). Papi software-defined events for in-depth performance analysis. *The International Journal of High Performance Computing Applications*, 33:1113–1127. 11

[29] Johnson, D. S. (1973). *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology. 89

[30] Jost, G., Jin, H., Labarta, J., Gimenez, J., and Caubet, J. (2003). Performance analysis of multilevel parallel applications on shared memory architectures. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 80.2–, Washington, DC, USA. IEEE Computer Society. 2, 15

[31] Keller, R., Bosilca, G., Fagg, G., Resch, M., and Dongarra, J. J. (2006). Implementation and usage of the peruse-interface in open mpi. In *Proceedings of the 13th European PVM/MPI User's Group Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, EuroPVM/MPI'06, pages 347–355, Berlin, Heidelberg. Springer-Verlag. xii, 2, 19, 106

[32] Kerrisk, M. and Brouwer, A. (2020). *mmap(2) Linux Programmer's Manual*. http://man7.org/linux/man-pages/man2/mmap.2.html. 57

[33] Knüpfer, A., Rössel, C., Mey, D. a., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., Nagel, W. E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S., Tschüter, R., Wagner, M., Wesarg, B., and

Wolf, F. (2012). *Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir*, pages 79–91. Springer Berlin Heidelberg, Berlin, Heidelberg. 13

[34] Losada, N., Bouteiller, A., and Bosilca, G. (2019). Asynchronous receiver-driven replay for local rollback of mpi applications. In *Fault Tolerance for HPC at eXtreme Scale (FTXS) Workshop at The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'19)*. 66

[35] Malony, A. D., Biersdorff, S., Shende, S., Jagode, H., Tomov, S., Juckeland, G., Dietrich, R., Poole, D., and Lamb, C. (2011). Parallel performance measurement of heterogeneous parallel systems with gpus. In *International Conference on Parallel Processing (ICPP'11)*, Taipei, Taiwan. ACM, ACM. 11

[36] Miller, B. P., Callaghan, M. D., Cargille, J. M., Hollingsworth, J. K., Irvin, R. B., Karavanic, K. L., Kunchithapadam, K., and Newhall, T. (1995). The paradyn parallel performance measurement tool. *Computer*, 28(11):37–46. 11, 12

[37] Moore, G. E. et al. (1965). Cramming more components onto integrated circuits. 24

[38] Mucci, P. J., Browne, S., Deane, C., and Ho, G. (1999). Papi: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10. 4, 11

[39] NVIDIA Corporation (2019). *CUDA Toolkit Documentation v10.2.89*. https://docs.nvidia.com/cuda/index.html. 1, 4, 20, 81, 86, 89, 92

[40] OpenMP Architecture Review Board (2018). *OpenMP Application Programming Interface Version 5.0*. https://www.openmp.org/. 1, 83, 86

[41] Paradyn Project (2020). Dyninst documentation. https://www.dyninst.org/. 12

[42] Patinyasakdikul, T., Eberius, D., Bosilca, G., and Hjelm, N. (2019). Give mpi threading a fair chance: A study of multithreaded mpi designs. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–11. IEEE. 66, 109

[43] Patinyasakdikul, T., Luo, X., Eberius, D., and Bosilca, G. (2019). Multirate: A flexible mpi benchmark for fast assessment of multithreaded communication performance. In *2019 IEEE/ACM Workshop on Exascale MPI (ExaMPI)*, pages 1–11. 61, 109

[44] Pearce, O., Ahmed, H., Larsen, R. W., Pirkelbauer, P., and Richards, D. F. (2018). Exploring Dynamic Load Imbalance Solutions with the CoMD Proxy Application. In *Future Generation Computer Systems*. LLNL-JRNL-725317. 7, 81, 82, 85, 86, 94, 96, 99

[45] Perarnau, S., Thakur, R., Iskra, K., Raffenetti, K., Cappello, F., Gupta, R., Beckman, P., Snir, M., Hoffmann, H., Schulz, M., and Rountree, B. (2015). Distributed monitoring and management of exascale systems in the argo project. In Bessani, A. and Bouchenak, S., editors, *Distributed Applications and Interoperable Systems*, pages 173–178, Cham. Springer International Publishing. 18

[46] Plimpton, S. (1995). Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117:1–19. Available at `http://lammps.sandia.gov`. 56, 67

[47] R. D. Hornung and J. A. Keasler (2014). The RAJA Portability Layer: Overview and Status. Technical Report LLNL-TR-661403, Lawrence Livermore National Laboratory. 81, 86

[48] Ramesh, S., Mahéo, A., Shende, S., Malony, A. D., Subramoni, H., Ruhela, A., and Panda, D. K. D. (2018). Mpi performance engineering with the mpi tool interface: the integration of mvapich and tau. *Parallel Computing*, 77:19–37. 18

[49] Scalasca Project (2020). Scalasca documentation. `https://www.scalasca.org/scalasca/about/about.html`. 13, 16

[50] Schmitt, F., Stolle, J., and Dietrich, R. (2014). Casita: A tool for identifying critical optimization targets in distributed heterogeneous applications. In *2014 43rd International Conference on Parallel Processing Workshops*, pages 186–195. 22

[51] Shende, S. S. and Malony, A. D. (2006). The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311. 2, 4, 12, 13, 18

[52] Snell, Q. O., Mikler, A. R., Gustafson, J. L., and Archives, T. P. S. U. C. (1996). Netpipe: A network protocol independent performance evaluator. 45

[53] Strohmaier, E., Dongarra, J., Simon, H., and Meuer, M. (2019). Top500 november 2019 list. 3, 24

[54] Tallent, N. R., Adhianto, L., and Mellor-Crummey, J. M. (2010a). Scalable identification of load imbalance in parallel executions using call path profiles. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE. 21

[55] Tallent, N. R., Mellor-Crummey, J. M., and Porterfield, A. (2010b). Analyzing lock contention in multithreaded applications. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 269–280. 21

[56] TAU Project (2020). Tau documentation. https://www.cs.uoregon.edu/research/tau/home.php. 12, 16

[57] Terpstra, D., Jagode, H., You, H., and Dongarra, J. (2010). Collecting performance data with papi-c. *Tools for High Performance Computing 2009*, pages 157–173. 11

[58] Underwood, K. D. and Brightwell, R. (2004). The impact of mpi queue usage on message latency. In *International Conference on Parallel Processing, 2004. ICPP 2004.*, pages 152–160 vol.1. 32, 39

[59] Vampir Team (2020). Vampir Documentation. https://vampir.eu/. 15

[60] Vetter, J. and Chambreau, C. (2005). mpip: Lightweight, scalable mpi profiling. 2, 18

[61] Woodall, T. S., Graham, R. L., Castain, R. H., Daniel, D. J., Sukalski, M. W., Fagg, G. E., Gabriel, E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., and Lumsdaine, A. (2004). *Open MPI's TEG Point-to-Point Communications Methodology: Comparison to Existing Implementations*, pages 105–111. Springer Berlin Heidelberg, Berlin, Heidelberg. 3, 31

Appendices

A List of SPCs

Table 1: A list of the currently available SPCs in my pull request to the Open MPI development repository. Note: '*' represents 'OMPI_SPC'. Table 1 is continued on to pages 122-133.

Level	SPC Name	SPC Description
MPI	*_SEND	The number of times MPI_Send was called.
MPI	*_BSEND	The number of times MPI_Bsend was called.
MPI	*_RSEND	The number of times MPI_Rsend was called.
MPI	*_SSEND	The number of times MPI_Ssend was called.
MPI	*_RECV	The number of times MPI_Recv was called.
MPI	*_MRECV	The number of times MPI_Mrecv was called.
MPI	*_ISEND	The number of times MPI_Isend was called.
MPI	*_IBSEND	The number of times MPI_Ibsend was called.
MPI	*_IRSEND	The number of times MPI_Irsend was called.
MPI	*_ISSEND	The number of times MPI_Issend was called.
MPI	*_IRECV	The number of times MPI_Irecv was called.
MPI	*_SENDRECV	The number of times MPI_Sendrecv was called.

Table 1 (continued)

MPI	*_SENDRECV_REPLACE	The number of times MPI_Sendrecv_replace was called.
MPI	*_PUT	The number of times MPI_Put was called.
MPI	*_RPUT	The number of times MPI_Rput was called.
MPI	*_GET	The number of times MPI_Get was called.
MPI	*_RGET	The number of times MPI_Rget was called.
MPI	*_PROBE	The number of times MPI_Probe was called.
MPI	*_IPROBE	The number of times MPI_Iprobe was called.
MPI	*_BCAST	The number of times MPI_Bcast was called.
MPI	*_IBCAST	The number of times MPI_Ibcast was called.
MPI	*_BCAST_INIT	The number of times MPI_Bcast_init was called.
MPI	*_REDUCE	The number of times MPI_Reduce was called.
MPI	*_REDUCE_SCATTER	The number of times MPI_Reduce_scatter was called.
MPI	*_REDUCE_SCATTER_BLOCK	The number of times MPI_Reduce_scatter_block was called.
MPI	*_IREDUCE	The number of times MPI_Ireduce was called.
MPI	*_IREDUCE_SCATTER	The number of times MPI_Ireduce_scatter was called.

Table 1 (continued)

MPI	*_IREDUCE_SCATTER_BLOCK	The number of times MPI_Ireduce_scatter_block was called.
MPI	*_REDUCE_INIT	The number of times MPI_Reduce_init was called.
MPI	*_REDUCE_SCATTER_INIT	The number of times MPI_Reduce_scatter_init was called.
MPI	*_REDUCE_SCATTER_BLOCK_INIT	The number of times MPI_Reduce_scatter_block_init was called.
MPI	*_ALLREDUCE	The number of times MPI_Allreduce was called.
MPI	*_IALLREDUCE	The number of times MPI_Iallreduce was called.
MPI	*_ALLREDUCE_INIT	The number of times MPI_Allreduce_init was called.
MPI	*_SCAN	The number of times MPI_Scan was called.
MPI	*_EXSCAN	The number of times MPI_Exscan was called.
MPI	*_ISCAN	The number of times MPI_Iscan was called.
MPI	*_IEXSCAN	The number of times MPI_Iexscan was called.
MPI	*_SCAN_INIT	The number of times MPI_Scan_init was called.
MPI	*_EXSCAN_INIT	The number of times MPI_Exscan_init was called.

Table 1 (continued)

MPI	*_SCATTER	The number of times MPI_Scatter was called.
MPI	*_SCATTERV	The number of times MPI_Scatterv was called.
MPI	*_ISCATTER	The number of times MPI_Iscatter was called.
MPI	*_ISCATTERV	The number of times MPI_Iscatterv was called.
MPI	*_SCATTER_INIT	The number of times MPI_Scatter_init was called.
MPI	*_SCATTERV_INIT	The number of times MPI_Scatterv_init was called.
MPI	*_GATHER	The number of times MPI_Gather was called.
MPI	*_GATHERV	The number of times MPI_Gatherv was called.
MPI	*_IGATHER	The number of times MPI_Igather was called.
MPI	*_IGATHERV	The number of times MPI_Igatherv was called.
MPI	*_GATHER_INIT	The number of times MPI_Gather_init was called.
MPI	*_GATHERV_INIT	The number of times MPI_Gatherv_init was called.
MPI	*_ALLTOALL	The number of times MPI_Alltoall was called.
MPI	*_ALLTOALLV	The number of times MPI_Alltoallv was called.

Table 1 (continued)

MPI	*_ALLTOALLW	The number of times MPI_Alltoallw was called.
MPI	*_IALLTOALL	The number of times MPI_Ialltoall was called.
MPI	*_IALLTOALLV	The number of times MPI_Ialltoallv was called.
MPI	*_IALLTOALLW	The number of times MPI_Ialltoallw was called.
MPI	*_ALLTOALL_INIT	The number of times MPI_Alltoall_init was called.
MPI	*_ALLTOALLV_INIT	The number of times MPI_Alltoallv_init was called.
MPI	*_ALLTOALLW_INIT	The number of times MPI_Alltoallw_init was called.
MPI	*_NEIGHBOR_ALLTOALL	The number of times MPI_Neighbor_alltoall was called.
MPI	*_NEIGHBOR_ALLTOALLV	The number of times MPI_Neighbor_alltoallv was called.
MPI	*_NEIGHBOR_ALLTOALLW	The number of times MPI_Neighbor_alltoallw was called.
MPI	*_INEIGHBOR_ALLTOALL	The number of times MPI_Ineighbor_alltoall was called.
MPI	*_INEIGHBOR_ALLTOALLV	The number of times MPI_Ineighbor_alltoallv was called.
MPI	*_INEIGHBOR_ALLTOALLW	The number of times MPI_Ineighbor_alltoallw was called.
MPI	*_NEIGHBOR_ALLTOALL_INIT	The number of times MPI_Neighbor_alltoall_init was called.

Table 1 (continued)

MPI	*_NEIGHBOR_ALLTOALLV_INIT	The number of times MPI_Neighbor_alltoallv_init was called.
MPI	*_NEIGHBOR_ALLTOALLW_INIT	The number of times MPI_Neighbor_alltoallw_init was called.
MPI	*_ALLGATHER	The number of times MPI_Allgather was called.
MPI	*_ALLGATHERV	The number of times MPI_Allgatherv was called.
MPI	*_IALLGATHER	The number of times MPI_Iallgather was called.
MPI	*_IALLGATHERV	The number of times MPI_Iallgatherv was called.
MPI	*_ALLGATHER_INIT	The number of times MPI_Allgather_init was called.
MPI	*_ALLGATHERV_INIT	The number of times MPI_Allgatherv_init was called.
MPI	*_NEIGHBOR_ALLGATHER	The number of times MPI_Neighbor_allgather was called.
MPI	*_NEIGHBOR_ALLGATHERV	The number of times MPI_Neighbor_allgatherv was called.
MPI	*_INEIGHBOR_ALLGATHER	The number of times MPI_Ineighbor_allgather was called.
MPI	*_INEIGHBOR_ALLGATHERV	The number of times MPI_Ineighbor_allgatherv was called.
MPI	*_NEIGHBOR_ALLGATHER_INIT	The number of times MPI_Neighbor_allgather_init was called.
MPI	*_NEIGHBOR_ALLGATHERV_INIT	The number of times MPI_Neighbor_allgatherv_init was called.

Table 1 (continued)

MPI	*_TEST	The number of times MPI_Test was called.
MPI	*_TESTALL	The number of times MPI_Testall was called.
MPI	*_TESTANY	The number of times MPI_Testany was called.
MPI	*_TESTSOME	The number of times MPI_Testsome was called.
MPI	*_WAIT	The number of times MPI_Wait was called.
MPI	*_WAITALL	The number of times MPI_Waitall was called.
MPI	*_WAITANY	The number of times MPI_Waitany was called.
MPI	*_WAITSOME	The number of times MPI_Waitsome was called.
MPI	*_BARRIER	The number of times MPI_Barrier was called.
MPI	*_IBARRIER	The number of times MPI_Ibarrier was called.
MPI	*_BARRIER_INIT	The number of times MPI_Barrier_init was called.
MPI	*_WTIME	The number of times MPI_Wtime was called.
MPI	*_CANCEL	The number of times MPI_Cancel was called.

Table 1 (continued)

PML	*_BYTES_RECEIVED_USER	The number of bytes received by the user through point-to-point communications. Note: Includes bytes transferred using internal RMA operations.
PML	*_BYTES_RECEIVED_MPI	The number of bytes received by MPI through collective
PML	*_BYTES_SENT_USER	The number of bytes sent by the user through point-to-point communications. Note: Includes bytes transferred using internal RMA operations.
PML	*_BYTES_SENT_MPI	The number of bytes sent by MPI through collective
PML	*_BYTES_PUT	The number of bytes sent/received using RMA Put operations both through user-level Put functions and internal Put functions.
PML	*_BYTES_GET	The number of bytes sent/received using RMA Get operations both through user-level Get functions and internal Get functions.
PML	*_UNEXPECTED	The number of messages that arrived as unexpected messages.
PML	*_OUT_OF_SEQUENCE	The number of messages that arrived out of the proper sequence.
PML	*_MATCH_TIME	The number of microseconds spent matching unexpected messages. Note: The timer used on the back end is in cycles

Table 1 (continued)

PML	*_MATCH_QUEUE_TIME	The number of microseconds spent inserting unexpected messages into the unexpected message queue. Note: The timer used on the back end is in cycles
PML	*_UNEXPECTED_IN_QUEUE	The number of messages that are currently in the unexpected message queue
PML	*_OOS_IN_QUEUE	The number of messages that are currently in the out of sequence message queue
PML	*_MAX_UNEXPECTED_IN_QUEUE	The maximum number of messages that the unexpected message queue
PML	*_MAX_OOS_IN_QUEUE	The maximum number of messages that the out of sequence message queue
MPI	*_BASE_BCAST_LINEAR	The number of times the base broadcast used the linear algorithm.
MPI	*_BASE_BCAST_CHAIN	The number of times the base broadcast used the chain algorithm.
MPI	*_BASE_BCAST_PIPELINE	The number of times the base broadcast used the pipeline algorithm.
MPI	*_BASE_BCAST_SPLIT_BINTREE	The number of times the base broadcast used the split binary tree algorithm.
MPI	*_BASE_BCAST_BINTREE	The number of times the base broadcast used the binary tree algorithm.
MPI	*_BASE_BCAST_BINOMIAL	The number of times the base broadcast used the binomial algorithm.
MPI	*_BASE_REDUCE_CHAIN	The number of times the base reduce used the chain algorithm.

Table 1 (continued)

MPI	*_BASE_REDUCE_PIPELINE	The number of times the base reduce used the pipeline algorithm.
MPI	*_BASE_REDUCE_BINARY	The number of times the base reduce used the binary tree algorithm.
MPI	*_BASE_REDUCE_BINOMIAL	The number of times the base reduce used the binomial tree algorithm.
MPI	*_BASE_REDUCE_IN_ORDER_BINTREE	The number of times the base reduce used the in order binary tree algorithm.
MPI	*_BASE_REDUCE_LINEAR	The number of times the base reduce used the basic linear algorithm.
MPI	*_BASE_REDUCE_SCATTER_NONOVERLAPPING	The number of times the base reduce scatter used the nonoverlapping algorithm.
MPI	*_BASE_REDUCE_SCATTER_RECURSIVE_HALVING	The number of times the base reduce scatter used the recursive halving algorithm.
MPI	*_BASE_REDUCE_SCATTER_RING	The number of times the base reduce scatter used the ring algorithm.
MPI	*_BASE_ALLREDUCE_NONOVERLAPPING	The number of times the base allreduce used the nonoverlapping algorithm.
MPI	*_BASE_ALLREDUCE_ RECURSIVE_DOUBLING	The number of times the base allreduce used the recursive doubling algorithm.
MPI	*_BASE_ALLREDUCE_RING	The number of times the base allreduce used the ring algorithm.
MPI	*_BASE_ALLREDUCE_RING_SEGMENTED	The number of times the base allreduce used the segmented ring algorithm.
MPI	*_BASE_ALLREDUCE_LINEAR	The number of times the base allreduce used the linear algorithm.
MPI	*_BASE_SCATTER_BINOMIAL	The number of times the base scatter used the binomial tree algorithm.

Table 1 (continued)

MPI	*_BASE_SCATTER_LINEAR	The number of times the base scatter used the linear algorithm.
MPI	*_BASE_GATHER_BINOMIAL	The number of times the base gather used the binomial tree algorithm.
MPI	*_BASE_GATHER_LINEAR_SYNC	The number of times the base gather used the synchronous linear algorithm.
MPI	*_BASE_GATHER_LINEAR	The number of times the base gather used the linear algorithm.
MPI	*_BASE_ALLTOALL_INPLACE	The number of times the base alltoall used the in-place algorithm.
MPI	*_BASE_ALLTOALL_PAIRWISE	The number of times the base alltoall used the pairwise algorithm.
MPI	*_BASE_ALLTOALL_BRUCK	The number of times the base alltoall used the bruck algorithm.
MPI	*_BASE_ALLTOALL_LINEAR_SYNC	The number of times the base alltoall used the synchronous linear algorithm.
MPI	*_BASE_ALLTOALL_TWO_PROCS	The number of times the base alltoall used the two process algorithm.
MPI	*_BASE_ALLTOALL_LINEAR	The number of times the base alltoall used the linear algorithm.
MPI	*_BASE_ALLGATHER_BRUCK	The number of times the base allgather used the bruck algorithm.
MPI	*_BASE_ALLGATHER_ RECURSIVE_DOUBLING	The number of times the base allgather used the recursive doubling algorithm.
MPI	*_BASE_ALLGATHER_RING	The number of times the base allgather used the ring algorithm.
MPI	*_BASE_ALLGATHER_ NEIGHBOR_EXCHANGE	The number of times the base allgather used the neighbor exchange algorithm.

Table 1 (continued)

MPI	*_BASE_ALLGATHER_TWO_PROCS	The number of times the base allgather used the two process algorithm.
MPI	*_BASE_ALLGATHER_LINEAR	The number of times the base allgather used the linear algorithm.
MPI	*_BASE_BARRIER_DOUBLE_RING	The number of times the base barrier used the double ring algorithm.
MPI	*_BASE_BARRIER_RECURSIVE_DOUBLING	The number of times the base barrier used the recursive doubling algorithm.
MPI	*_BASE_BARRIER_BRUCK	The number of times the base barrier used the bruck algorithm.
MPI	*_BASE_BARRIER_TWO_PROCS	The number of times the base barrier used the two process algorithm.
MPI	*_BASE_BARRIER_LINEAR	The number of times the base barrier used the linear algorithm.
MPI	*_BASE_BARRIER_TREE	The number of times the base barrier used the tree algorithm.
PML	*_P2P_MESSAGE_SIZE	This is a bin counter with two subcounters. The first is messages that are less than or equal to mpi_spc_p2p_message_boundary bytes and the second is those that are larger than mpi_spc_p2p_message_boundary bytes.
PML	*_EAGER_MESSAGES	The number of messages that fall within the eager size.
PML	*_NOT_EAGER_MESSAGES	The number of messages that do not fall within the eager size.

Table 1 (continued)

PML	*_QUEUE_ALLOCATION	The amount of memory allocated after runtime currently in use for temporary message queues like the unexpected message queue and the out of sequence message queue.
PML	*_MAX_QUEUE_ALLOCATION	The maximum amount of memory allocated after runtime at one point for temporary message queues like the unexpected message queue and the out of sequence message queue. Note: The *_QUEUE_ALLOCATION counter must also be activated.
PML	*_UNEXPECTED_QUEUE_DATA	The amount of memory currently in use for the unexpected message queue.
PML	*_MAX_UNEXPECTED_QUEUE_DATA	The maximum amount of memory in use for the unexpected message queue. Note: The *_UNEXPECTED_QUEUE_DATA counter must also be activated.
PML	*_OOS_QUEUE_DATA	The amount of memory currently in use for the out-of-sequence message queue.
PML	*_MAX_OOS_QUEUE_DATA	The maximum amount of memory in use for the out-of-sequence message queue. Note: The *_OOS_QUEUE_DATA counter must also be activated.

B SPC Example Code

```python

import sys
import glob
import operator
import struct

import numpy as np
import matplotlib
matplotlib.use('Agg') # For use with headless systems
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import matplotlib.ticker as ticker

def combine(filename, data):
    f = open(filename, 'rb')
    for i in range(0,num_counters):
        temp = struct.unpack('l', f.read(8))[0]
        if 'TIME' in names[i]:
            temp /= freq_mhz
        data[i].append(temp)

def fmt(x, pos):
    return '{:,.0f}'.format(x)

# Make sure the proper number of arguments have been supplied
if len(sys.argv) < 3:
    print("Usage: ./parse.py [/path/to/data/files] [datafile_label]")
    exit()

path = sys.argv[1]
```

```
label = sys.argv[2]

xml_filename = ''
# Lists for storing the snapshot data files from each rank
copies = []
ends = []
# Populate the lists with the appropriate data files
for filename in glob.glob(path + "/spc_data*"):
    if label in filename:
        if xml_filename == '' and '.xml' in filename:
            xml_filename = filename
        if '.xml' not in filename:
            temp = filename.split('/')[-1].split('.')
            if len(temp) < 5:
                temp[-1] = int(temp[-1])
                ends.append(temp)
            else:
                temp[-1] = int(temp[-1])
                temp[-2] = int(temp[-2])
                copies.append(temp)

# Sort the lists
ends = sorted(ends, key = operator.itemgetter(-1))
for i in range(0,len(ends)):
    ends[i][-1] = str(ends[i][-1])
copies = sorted(copies, key = operator.itemgetter(-2,-1))
for i in range(0,len(copies)):
    copies[i][-1] = str(copies[i][-1])
    copies[i][-2] = str(copies[i][-2])

sep = '.'

```

```
xml_file = open(xml_filename, 'r')
num_counters = 0
freq_mhz = 0
names = []
base = []
# Parse the XML file (same for all data files)
for line in xml_file:
    if 'num_counters' in line:
        num_counters = int(line.split('>')[1].split('<')[0])
    if 'freq_mhz' in line:
        freq_mhz = int(line.split('>')[1].split('<')[0])
    if '<name>' in line:
        names.append(line.split('>')[1].split('<')[0])
        value = [names[-1]]
        base.append(value)

prev = copies[0]
i = 0
ranks = []
values = []
times = []
time = []

# Populate the data lists
for n in range(0,len(base)):
    values.append([0, names[n]])
for c in copies:
    if c[-2] != prev[-2]:
        filename = path + "/" + sep.join(ends[i])
        combine(filename, values)

        ranks.append(values)
```

```
95          times.append(time)
96          for j in range(0, len(names)):
97              temp = [ranks[0][j][0]]
98
99          values = []
100         time = []
101         for n in range(0,len(base)):
102             values.append([i+1, names[n]])
103         i += 1
104
105     filename = path + "/" + sep.join(c)
106     time.append(int(filename.split('.')[-1]))
107     combine(filename, values)
108     prev = c
109
110 filename = path + "/" + sep.join(ends[i])
111 combine(filename, values)
112 ranks.append(values)
113 times.append(time)
114
115 for i in range(0, len(names)):
116     fig = plt.figure(num=None, figsize=(7, 9), dpi=200, facecolor='w',
            edgecolor='k')
117
118     plot = False
119     # Only plot the SPCs of interest
120     if names[i] == 'OMPI_SPC_BYTES_SENT_USER':
121         plot = True
122
123     map_data = []
124     avg_x = []
125
```

```
126     for j in range(0, len(ranks)):
127         if avg_x == None:
128             avg_x = np.zeros(len(times[j])-1)
129         empty = True
130         for k in range(2,len(ranks[j][i])):
131             if ranks[j][i][k] != 0:
132                 empty = False
133                 break
134         if not empty:
135             if plot:
136                 xvals = []
137                 yvals = []
138                 for l in range(1, len(times[j])):
139                     if ranks[j][i][l+2] - ranks[j][i][l+1] < 0:
140                         break
141                     xvals.append(times[j][l] - times[j][0])
142                     yvals.append(ranks[j][i][l+2] - ranks[j][i][l+1])
143
144                 map_data.append(yvals)
145                 for v in range(0,len(avg_x)):
146                     avg_x[v] += xvals[v]
147     if plot:
148         for v in range(0,len(avg_x)):
149             avg_x[v] /= float(len(ranks))
150
151         ax = plt.gca()
152         im = ax.imshow(map_data, cmap='Reds', interpolation='nearest')
153
154         cbar = ax.figure.colorbar(im, ax=ax, format=ticker.FuncFormatter(fmt))
155         cbar.ax.set_ylabel("Counter Value", rotation=-90, va="bottom")
156
157         plt.title(names[i] + ' Snapshot Difference')
```

```

        plt.xlabel('Time')
        plt.ylabel('MPI Rank')

        ax.set_xticks(np.arange(len(avg_x)))
        ax.set_yticks(np.arange(len(map_data)))
        ax.set_xticklabels(avg_x)

        plt.show()
        fig.savefig(names[i] + '.png')
```

Listing 1: An example Python script for parsing SPC snapshot data files and creating a heatmap of their value differences over time.

Vita

David William Eberius was born on April 28th, 1992 in Baltimore, Maryland. He attended Patterson Mill High School and graduated in 2010. He received a Bachelor of Science degree in Computer Science with a minor in Mathematics from Salisbury University in 2014. Also in 2014, he enrolled at the University of Tennessee as a Master's Degree candidate in Computer Science, and later transferred into Ph.D. candidacy in Computer Science in 2016, both under the advisement of Dr. Jack Dongarra. Throughout his time at The University of Tennessee, David served as a Graduate Research Assistant in the Innovative Computing Laboratory under Dr. Jack Dongarra, supervised by Dr. George Bosilca. His research interests are in High-Performance Computing with a particular focus in performance analysis and tools. During his Ph.D. studies, David completed several internships at United States Government institutions such as Oak Ridge National Laboratory in 2015, The Army Research Laboratory in 2016, and Lawrence Livermore National Laboratory in 2018 and 2019. He also competed in the World Whistlers Convention in 2016, and the Masters of Musical Whistling Festival in 2019 receiving second place in the Stage 2 Popular category. David is expected to receive his Doctor of Philosophy degree in Computer Science in August of 2020. After graduation, he will be pursuing the next step in his career with a position as a Postdoctoral Appointee at Oak Ridge National Laboratory.