Spring 5-1-2017

Characterizing and Improving Power and Performance in HPC Networks

Taylor L. Groves
University of New Mexico

Taylor Groves

Candidate

Computer Science

Department

This dissertation is approved, and it is acceptable in quality and form for publication:

Approved by the Dissertation Committee:

Dr. Dorian Arnold, Chair

Dr. Dave Ackley, Member

Dr. Patrick Bridges, Member

Dr. Wei Wennie Shu, Member

Characterizing and Improving Power and Performance of HPC Networks

by

Taylor Groves
tgroves@cs.unm.edu

B.S., Computer Science, Texas State University, 2009
M.S., Computer Science, University of New Mexico, 2012

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

May 2017

tgroves@cs.unm.edu

Dedication

To Haley, The Unburnt, Queen of the Andals, the Rhoynar, and of the First Men, Queen of Meereen, Khaleesi of the Great Grass Sea, Breaker of Chains, Mother of Dragons, Hank and Simon.

"Just remember, if you hang in there long enough, good things can happen in this world."

Tom Smykowski

Acknowledgments

I thank my family for all of their support throughout the years. Their love and encouragement laid the foundation to which I owe my success. No matter what kind of day it's been, my children (Hank and Simon) are always able to bring a smile to my face. The other day, my three year old Hank said "we might be moving soon", and I asked where. He said, "Not you. You can come when you are done being a doctor." So, it's important that I finish this doctor-thing now before they move away without me.

My wife, Haley always pushes me to achieve my best and has made many sacrifices that allow me the time to pursue a doctorate. I have known Haley since high school and she has been a huge part of my life for more than a decade. Through all of this time her love and dedication has been absolute and something I could always depend on. None of this would have been possible without her support.

My mother and father have always provided love, support and given me the confidence (maybe too much) to believe I could achieve great things. Sadly, my father was unable to see the completion of this endeavor, but I will always remember his kindness and his passion for engineering and exploration. Beyond the immediate family, my brother and sisters, grandparents, in-laws, aunts, uncles and cousins have all been there for me and I appreciate it.

There have been many individuals throughout my time in academia and industry who have had a hand in shaping my research. Starting in my undergraduate degree, there were several professors who helped spark an interest in graduate studies, specifically, Xiao Chen, Byron Gao, Mina Guirguis, Carol Hazelwood and Mark McKenney. I remember attending the network research group at Texas State with Dr. Guirguis and Dr. Chen. It was here that I got my first taste of graduate research as people discussed papers and presented their work. Later on I would get my first publication with Dr. Chen, writing a simulator for mobile networks.

At UNM I have to thank the entire Scalable Systems Lab for listening to all of those boring talks and presentations I gave throughout the years and providing feedback and expertise. Special thanks to Dorian Arnold, Patrick Bridges, Zhenjie Chen, Matthew Dosanjh, Evan Dye, Noah Evans, Josh Goehner, Aaron Gonzales, Sam Gutierrez, Nathan Hjelm, Dewan Ibtesham, Beverly Klemme, Scott Levy, Oscar Mondragon, Donour Sizemore, Whit Schonbein, Philip Soltero, and Hans Weeks. I had great times playing Civilization and racing cars with Donour before he decided to work hard and graduate. Scott is always a great person to bounce ideas off of, rig CSGSA elections for, and to ask for legal advice when your dog bites the mailman. At various points I shared an office with Dewan and Matt, for which I apologize. Outside of the SSL, UNM has been an amazing place to develop my knowledge and understanding. There are other groups and students that played a large part in making this time

a great experience: Ben Edwards, Roya Ensafi, Sunny Fugate, Ben Gordon, Jeff Knockel, Drew Levin, David Mohr, Eric Schulte and George Stelle. I think it must have been an anomaly that we had so many great people in the graduate program all at one time. I have many amazing memories of hanging out and celebrating with these people and our families.

During my stay, I spent some time exploring opportunities in industry which led to an internship at Yahoo!. I had a great time in the Bay Area and learned a tremendous amount about data-center networks from people like Yihua He and Jeremy Dahl. I miss the Bay-Area weather, the free food, and the great people I met out there.

When I finished working at Yahoo!, I got to meet another group of great people as I interned at Sandia National Laboratories. Sandia played a large role shaping me into the researcher I am today. What amazed me the most about Sandia is the volume of expertise concentrated in CSRI and CERL. There are always new and interesting problems to work on and good people to work with. I received immense amounts of time and support from individuals in 1422 and 1423: Ron Brightwell, Kurt Ferreira, Si Hammond, Scott Hemmert, Michael Levenhagen, Stephen Olivier, Kevin Pedretti, Patrick Widener, and most importantly my mentor of two years Ryan Grant.

Ryan's expertise and patience were crucial in the shaping of much of this research. Unofficially, Ryan was a second advisor to me. He was indespensible in the research process. From brainstorming ideas to the late night editing sessions, I could always count on Ryan's dedication and expertise.

Lastly, I would like to thank my proposal and dissertation committee members and advisor: David Ackley, Patrick Bridges, Melanie Moses, Wennie Shu, and Dorian Arnold.

I convinced Dorian to take me on as one of his students when class was canceled due to snow and I ran into him at a McDonald's. This is probably not the best way to find an advisor, but I've been incredibly lucky with how it worked out. Dorian took me under his wing and has had a huge impact on my research, my writing, and my outlook on the field. He has exposed me to a breadth of individuals – spanning national laboratories, universities and industry. Additionally, I consider myself lucky to find an advisor who was patient enough to invest in me for the long-haul – through coursework, comprehensive exams, deaths, births and everything else life brought my way. When I told Dorian that my father was diagnosed with cancer and that I would be shifting priorities away from school and research, making frequent trips to Texas, he continued to support me. This really meant a lot to me. I greatly appreciate everything he has taught me and the time he has invested in my success.

Thank you all.

Characterizing and Improving Power and Performance of HPC Networks

by

Taylor Groves
tgroves@cs.unm.edu

B.S., Computer Science, Texas State University, 2009
M.S., Computer Science, University of New Mexico, 2012

Ph.D., Computer Science, University of New Mexico, 2017

Abstract

Networks are the backbone of modern HPC systems. They serve as a critical piece of infrastructure, tying together applications, analytics, storage and visualization. Despite this importance, we have not fully explored how evolving communication paradigms and network design will impact scientific workloads. As networks expand in the race towards Exascale (1×10^{18} floating point operations a second), we need to reexamine this relationship so that the HPC community better understands (1) characteristics and trends in HPC communication; (2) how to best design HPC networks to save power or enhance the performance; (3) how to facilitate scalable, informed, and dynamic decisions within the network. *My thesis is that one can improve application performance and system power usage by gaining a detailed understanding of HPC communication on both the network endpoints and fabric; specifically, I address the problem of network-induced memory contention, quantify the power/performance tradeoffs for dragonfly topologies in HPC networks, and increase the scalability/responsiveness of large-scale network monitoring.* This dissertation

highlights opportunities for improving network performance and power efficiency, while uncovering pitfalls and mitigation strategies brought about by shifting trends in HPC communication and fabric design. We begin by examining the communication characteristics of the network endpoints. We show how one-sided communication techniques can lead to contention in the memory subsystem with (3X increases to runtime) and how this can be avoided. Then, we move onto a macro level study of the network fabric, where we demonstrate the tradeoffs between power and performance when designing HPC network topology. Lastly, in order to facilitate dynamic and responsive solutions, we provide new methods for scalable network monitoring and improved models of data aggregation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

High performance computing (HPC) systems continue to grow in size and complexity. In 2008 the Roadrunner supercomputer at Los Alamos National Laboratory had just broken the Petaflop[1] barrier and consumed 2.35 MW of power. In less than ten years, computing performance has increased by 125X and the power consumption by 6.4X. Currently, the largest system (Sunway TaihuLight) computes at 125 Pflop/s, using over 10 million cores [121]. The power consumption of this system is a massive 15MW. To put this in perspective, Sunway TaihuLight consumes more power than 7,000 U.S. homes [27]. The motivation for building such a large system comes from numerous research domains. Weather forecasting, climate modeling, nuclear test simulations and molecular dynamics are all complex problems that demand massive amounts of computational power to solve with high precision. For example, when NOAA recently upgraded its supercomputers Luna and Surge, Kathryn Sullivan, Ph.D., NOAA's administrator said [92],

[1] 1×10^{15} floating point operations per second

> The faster runs and better spatial and temporal resolution that Luna and Surge provide will allow NOAA to improve our environmental intelligence dramatically, giving the public faster and better predictions of weather, water and climate change. This enhanced environmental intelligence is vital to supporting the nation's physical safety and economic security.

And while the current systems are big, there is a push to reach Exascale (1×10^{18} floating point operations per second) computation. Both power and performance savings need to be extracted from every level of the system, in order to reach such scale. Network performance and power is the focus of this dissertation.

Networks are the backbone of modern HPC systems. They serve as critical infrastructure, tying together applications, analytics, storage and visualization. Despite this importance, we have not fully explored how evolving communication paradigms and network design will impact scientific workloads. As networks expand in the race towards Exascale, we must reexamine this relationship so that the HPC community better understands (1) characteristics and trends in HPC communication; (2) how to best design HPC networks to save power or enhance the performance; and (3) how monitoring facilitates scalable, informed, and dynamic decisions within the network. This understanding directly impacts our ability to create efficient Exascale platforms.

Challenges for HPC communication: The first challenge is understanding how to best leverage next-generation communication techniques. Traditionally, HPC has followed a bulk synchronous model of communication. That is, large portions of computation followed by mass communication/synchronization. Now, HPC is transitioning towards asynchronous task based paradigms that overlap communication and computation, with fine grain distribution of work across a large number of cores. This overlap between communication and computation can only occur if we can isolate incoming network traffic, so that it does not interrupt the processor responsible for

computation. In this respect, Remote direct memory access (RDMA) provides a convenient mechanism. RDMA allows nodes to largely bypass the CPU of a remote (target) node and read or write directly to a target's memory, which facilitates computation/communication overlap. This creates new opportunities throughout the HPC ecosystem in areas as diverse as resilience, system monitoring, analytics and visualization. These shifts in communication paradigms require us to reexamine the traditional models of communication and fully explore the impact one-sided communication has on the system.

Challenges for the network fabric: Communication paradigms are just a part of the challenges Exascale networks face. Additionally, we must consider the network fabric (the links, routers, switches and network cards). Modern HPC systems require low latency and high bandwidth networks. This demand for performance has forced a steady growth in bandwidth offerings as networks have moved from 54Gb/s in 2011 to a proposed 600Gb/s bandwidth in 2017 [65]. This volume of bandwidth and low latency is necessary due to the bursty traffic patterns common in HPC workloads. For many workloads, it is not uncommon for links to be idle for more than 84% of the runtime [43]. If a network is poorly provisioned, these traffic bursts may sporadically create bottlenecks on the fabric that propagate at scale, creating far reaching performance penalties known as system noise [101]. Such a performance penalty is unacceptable, so networks in HPC systems tend to be overprovisioned. This overprovisioning results in increased complexity and monetary costs to the system, and leaves potential for improvement. Furthermore, as we increase the scale of future systems, there is a concern that these systems will be constrained by power, such that we will not be able to power 100% of the system at any given time. HPC networks make up a significant portion of total system power. It takes around 1.7 MW to power the links and network cards in a 100,000 node system [43]. Therefore, we need to look at how we might temporarily shut down portions of the network to save power

when necessary.

Challenges for monitoring: Any strategy for scaling a network up or down dynamically must be aware of the workload and the power constraints. This awareness typically involves the use of large scale system monitoring. Yet, many of the larger HPC systems now span 100,000 nodes or more. When you consider the hundreds of thousands of switches, NICs, ports and cables operating at microsecond latencies, monitoring becomes a daunting task. Many of the traditional techniques for monitoring large systems (e.g. SNMP, OpenSM) are centralized, pull-based approaches that do not allow for scale or high granularity of collection. Furthermore, because of the propagation of noise, any monitoring technique must ensure that it does not unduly perturb the system. Even after all of the raw data has been collected, we have the additional challenge of transforming it into meaningful insights.

1.2 Dissertation Overview

This dissertation explores three topics in HPC networks (1) performance of one-sided communication and its interaction with the memory subsystem; (2) performance and power tradeoffs for different topologies of Exascale class networks; (3) best practices in monitoring large-scale networks.

1.2.1 Thesis statement

My thesis is that one can improve application performance and system power usage by gaining a detailed understanding of HPC communication on both the network endpoints and fabric; specifically, I address the problem of network-induced memory contention, quantify the power/performance tradeoffs for dragonfly topologies in HPC networks, and increase the scalability/responsiveness of large-scale network monitoring.

This dissertation highlights opportunities for improving network performance and power efficiency, while uncovering pitfalls and mitigation strategies brought about by shifting trends in HPC communication and fabric design.

At this point we provide a brief overview and contributions of the bodies of work that address these three topics. Each of these overviews are expanded fully in the succeeding chapters.

Evaluated the Performance of Multi-threaded One-sided Communication:[30]
On top of lower level communication layers sits libraries such as MPI. Remote Memory Access (RMA) operations have been incorporated as a part of the MPI-3 specification and it's important we evaluate their performance, so that we have a complete understanding of one-sided operations throughout the communication stack. To this end, we have developed and released a set of benchmarks to evaluate the performance of multi-threaded RMA operations in MPI. These benchmarks are used to both evaluate and debug Multi-threaded RMA performance within OpenMPI and MVAPICH implementations of MPI.

Characterized, Detected and Eliminated Network-induced Memory Contention:[42]
Communication paradigms are moving towards asynchronous communication mecha-

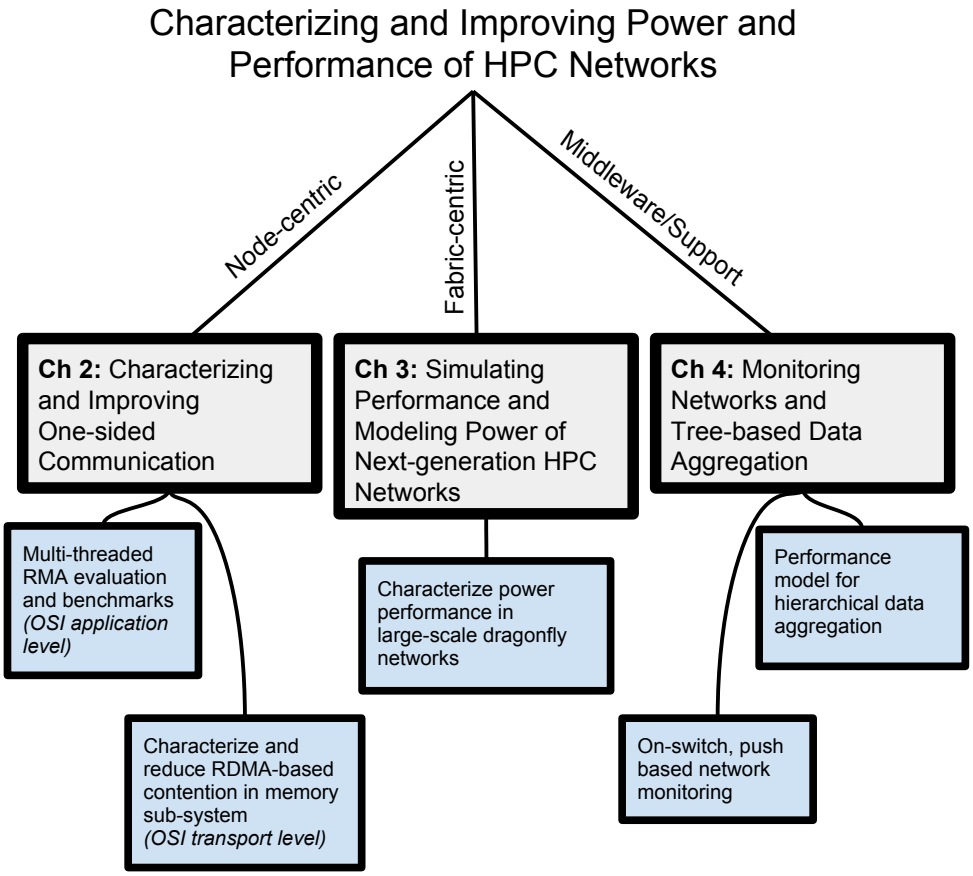

Figure 1.1: An illustration showing the relationship between the contributions of this work. We evaluated the network from the endpoints (node-centric), within the fabric (fabric-centric), and how we can enhance the scalability and responsiveness of network monitoring (middleware/support). Within each topic, the underlying goal is to improve the power and performance of the network, by removing bottlenecks, exposing underutilized resources and enhancing responsiveness.

nisms that provide opportunities for overlapping communication and computation by bypassing the CPU. We analyzed the impact this has on creating contention on the memory subsystem and potential solutions to reduce or avoid the associated performance penalty. Specifically, our results show Network-induced Memory Contention (NiMC) can increase application runtime by 3X at scales of 8,192 processes. Our proposed solutions may be enabled either through changes to hardware or changes in

software and can reduce the runtime penalty to a 0-6% increase. Furthermore, we explored how we might leverage machine learning to detect the occurrence of NiMC and predict the degree of performance degradation so that we may dynamically apply the best solution. Understanding this relationship between RDMA and application performance will be crucial to developing accurate models of network performance in future systems.

Analyzed Large-scale Networks and Characterized Power and Performance:[43]

The HPC community's understanding of current and emerging workloads on on Exascale network topologies is limited. Left ignored, this naivete will translate into missed opportunities to (1) increase application performance and (2) decrease both power and monetary costs in next generation systems. We used simulation to characterize a variety of relevant workloads on dragonfly topologies of 110,592 nodes. We examined tradeoffs in network design between execution time, power, bandwidth, and the number of global links. Our simulations report stalled, active and idle time on a per-port level of the fabric and we introduce a new method for visualizing network performance. The findings in this work will help shape network design decisions in future large-scale networks.

Enabled Scalable Network Monitoring:[41, 44]

Dynamic approaches for saving power and improving performance are reliant on streams of information regarding the current state of the system. For example, these data streams can contain valuable information about the power draw of different components or the utilization of a network link. Understanding the best practices and limitations of system monitoring is an important part of enabling dynamic solutions. With this in mind, we develop scalable push-based approaches for in-

network monitoring and introduce a new models for hierarchical data aggregation. We show that we can (1) improve the responsiveness of the monitoring system; (2) accurately predict the cost of data collection with a simple extension to canonical models of parallel computation.

1.2.2 Organization

We begin by exploring the performance of one-sided communication at the application level. This includes the development of a multi-threaded RMA benchmark suite, used to evaluate the performance of OpenMPI communication libraries. After presenting an benchmarks and evaluation for multi-threaded RMA performance, We characterize one-sided communication at the transport layer, specifically with a study of RDMA communication and its potential to create contention in memory (Chapter 2). Following this work, we expand the scope of our studies to include the network fabric (Chapter 3). We introduce a new approach to visualize networks at fine-grained, port-level detail. This visualization provides greater insight about network performance, that traditional metrics fail to capture. Using empirical measurements and analytical models of power, we evaluate how topology design impacts the power/performance tradeoff for large-scale dragonfly networks. We detail potential power savings for a power proportional network and different static configurations of the network topology. Furthermore, we estimate the amount of power saved by dynamically configuring the network. Dynamically configured networks should make adjustments according to the environment they operate in. This knowledge of the environment is commonly obtained through system monitoring. With this in mind, we explore limitations in scalable monitoring and develop in-network push based monitoring for large scale networks (Chapter 4). In this same chapter we extend a canonical model of parallel computation, to better represent hierarchical data aggregation that is central to scalable monitoring. Finally, we conclude with an overview of this dissertation's

contributions and reemphasize their relationships to each other (Chapter 5).

Chapter 2

Characterizing and Improving Performance of One-sided Communication

On today's high-performance computing (HPC) systems, standard communication mechanisms are synchronous, requiring active sender and receiver participation in message transmission – expending resources and time on both ends.

On emerging many-core Exascale-class systems, synchronous operation is expected to become a prohibitive bottleneck [35]. As systems and application researchers investigate novel approaches to reduce application synchronization, asynchronous communication is widely considered to be a part of the Exascale system design solution [35]. We expect that this emerging trend toward less synchronous computational paradigms and services, along with improvements to one-sided communications in MPI-3 [88] and new network technologies will lead to increased popularity for asynchronous communication and shared memory abstractions.

In this chapter we study one-sided communication at both the application layer

(MPI-3) and the transport layer (IB verbs). One-sided communication separates data movement from data synchronization so that data transfer can occur with only the involvement of a single (origin) process. While MPI provides abstractions for one-sided communication, the underlying implementation may not efficiently utilize passive data transfer. In order to evaluate the performance of one-sided MPI, we develop a suite of bandwidth, latency, message rate and application benchmarks. These are the first publicly available benchmarks to evaluate the combination of multi-threaded and one-sided performance in MPI.

After completing our study of the application layer, we strip away higher level abstractions of MPI and transition to the lower level transport layer, where we can leverage truly passive data transfer through the use of Remote Direct Memory Access (RDMA). In this second study, we characterize contention in the memory subsystem as a result of communication/computation overlap. We quantify the performance impact of NiMC on a variety of hardware architectures and explore how machine learning can detect the presence of NiMC and predict the increase to application runtime. We show that **NiMC can reduce single node memory bandwidth by 56%.** Furthermore, for an **application that is performance-resilient to system noise, we show NiMC can increase runtime by up to a factor of three** (at our tested scales). We then ask how NiMC can be eliminated, evaluating three candidate hardware and software solutions. Together, the two studies in this chapter provide valuable insights about how we can improve the effectiveness of one-sided communication on the network endpoints/nodes.

2.1 Evaluation of Multi-threaded RMA Performance in MPI

Two advancements that are expected to impact Exascale technology are one-sided communication and multi-threading. As the number of cores increases, there is an additional need to leverage parallism in communication. One-sided communication is a natural fit for a multi-threaded environment since this reduces the need for unnecessary synchronization and locks. By decoupling synchronzation and data transfer, threads are not contending for the same locks, provided they operate on independent regions of memory. While these two mechanisms will likely be part of the Exascale solution, the combination of multi-threaded and one-sided communication had not been explored previously.

In this section we will evaluate two MPI implementations of Multi-threaded Remote Memory Access (RMA). While both RDMA and RMA are considered one-sided communication, RMA is not RDMA. When discussed at higher layer in the network stack, such as MPI or SHMEM one-sided communication is referred to as RMA. An RMA programming interface provides a shared memory abstraction that may or may not be implemented on top of RDMA. To evaluate RMA-MT performance, we developed the first publicly available suite of RMA-MT micro-benchmarks and mini-applications [30].

This suite includes benchmarks of latency, bandwidth and message rate for each of the four different MPI-3 RMA synchronization methods. Additionally, both *put* and *get* data transfer operations are explored as well as varying thread count. The bandwidth and latency benchmarks are derived from `MPI_THREAD_MULTIPLE` benchmarks from Thakur and Gropp [118]. Message rate benchmarks were adapted from the Sandia Microbenchmarks (SMB) [28]. The modifications to each of these benchmarks required that they be redesigned, with two-sided MPI calls replaced

with the one-sided equivalent. Checksums were added to the bandwidth and latency benchmarks which allowed us to evaluate the correctness of the MPI implementations as well. As our results show, correctness of these RMA-MT MPI implementations was lacking, particularly as scale was increased. Our benchmarks were able to expose these issues to the MPI development team.

A final contribution of this work was the conversion of three mini-applications (HPCCG, MiniFE and MiniMD) from two-sided to one-sided. Each of these mini-applications represent computation and communication kernels important to the HPC community. The conversion of these mini-applications allowed us to evaluate the scalability of RMA-MT on clusters at Sandia of 512 processes. This scaling was crucial to uncovering some of the failure points in the evaluated MPI implementations.

In Section 2.1.3 we discuss the miniapps and micro-benchmarks in detail, providing informations about their development and design. Section 2.1.4 provides details about our experimental setup and evaluation. Sections 2.1.5 and 2.1.6 show the results of testing with the micro-benchmarks for a range of synchronization methods and thread counts. In Section 2.1.7 we highlight results and provide a discussion of the work with an assessment of three mini-apps (HPCCG, miniFE and miniMD). In Section 2.1.8 we outline the contributions of this work.

The work of developing RMA-MT benchmarks, running experiments and writing the paper was done collaboratively and shared between Matthew Dosanjh, Ryan Grant and myself [30]. I cannot take full credit for the contributions in this section of the dissertation. Specifically, I was responsible for developing the set of latency and bandwidth benchmarks, while Matthew developed the message rate benchmarks. Mini-app benchmarks were split between Matthew and Ryan. The work of running the experiments and analysis was divided between myself and Matthew, where I was in charge of the microbenchmarks and Matthew was responsible for mini-apps and message rate. As writing goes, both Matthew and I added text, figures and results

to the paper for the experiments we each developed and ran. Specifically, I authored text regarding latency and bandwidth benchmarks in Sections 2.1.3, 2.1.5, and 2.1.6. "Background" and "Conclusions" text was provided by Ryan Grant with "Experimental Setup" text provided by Matthew. For each of these three sections I had an editing role. The "Introduction" text for this chapter of the dissertation was written by myself, and differs from the original paper. For the sake of completeness, I include descriptions and results of the entire benchmark suite.

2.1.1 Background

MPI RMA

MPI first provided a one-sided communication interface in the MPI 2.0 specification [63]. It provided three synchronization methods for RMA: `MPI_Win_fence`, `MPI_Win_lock`, and "Post/Start/Complete/Wait" (PSCW). MPI RMA works by allowing one-sided put, get, and accumulate operations on shared windows of memory during an exposed time period, or epoch. Synchronization must occur between different epochs to ensure that the memory window is in a determinant state. This synchronization is key to enable reasoning about the current content of memory and its use in an application. The three methods provide active and passive target synchronization. `MPI_Win_fence` and PSCW both require the target of the RMA operations to participate in the operations through a call to fence or Post/Wait calls, and are therefore active target synchronization methods. `MPI_Win_lock` does not require that the target call lock, only the origin, and therefore is a passive target synchronization method.

To extend the capabilities of MPI 2.0 RMA, the one-sided (RMA) interface was updated in the MPI-3 specification [87]. `MPI_Win_lock_all` was introduced to solve two problems. First, MPI-2 required that when using locks only one target could

be locked at any given time. This was resolved with `MPI_Win_lock_all` as it allows an origin to obtain a *shared* lock on multiple targets at the same time. Second, fine grained data movement synchronization was provided through the `MPI_Win_flush` call that allows for assurance that remote operations on the window were complete. Also added in MPI 3.0 were request generating RMA operations (`MPI_RPut` etc.), new memory models (unified vs. separate), and new window types. The new window types enabled windows that can attached to memory after creation, windows with memory allocated by MPI, and shared memory windows. This work focuses on the basic operation of RMA in MPI with multi-threading and as such does not explore memory models or window types in great depth. The interested reader is referred to [25] for more information on these RMA features and a history of their development.

MPI multi-threading

MPI provides several threading modes. The default threading mode, `MPI_THREAD_SINGLE`, requires the guarantee that each process has only one execution thread at all times. `MPI_THREAD_FUNNELED` relaxes the single thread requirements by allowing multiple execution threads but requires that only one thread, specifically the one that called `MPI_Init_thread`, be the only thread that can make MPI calls. `MPI_THREAD_SERIALIZED` further relaxes guarantees by allowing multi-threaded processes and allowing any thread to call MPI but guaranteeing that only one thread at a time can make MPI calls (serialization is assured outside the MPI library). Finally, MPI provides `MPI_THREAD_MULTIPLE` which allows for multi-threaded processes and any thread may call MPI and they may do so concurrently. Some MPI implementations opt to treat the single, funneled, and serialized threading models similarly, as they all guarantee that only a single thread is in the MPI library at any given time.

MPI has provided `MPI_THREAD_MULTIPLE`, beginning in MPI-2.0 [63]. `MPI_THREAD_MULTIPLE` is not widely adopted, and consequently the multi-threaded

mode of MPI is not yet heavily tuned in MPI implementations. MPI RMA with multi-threading is not widely used due to a lack of benchmarks and application code utilizing the combination of methods. However, with new proposals to allow for more exposure of threads to MPI, such as the MPI Endpoints work [26, 113], the use of multi-threaded MPI may increase. RMA is a promising communication mechanism for future extreme scale systems; therefore, it is reasonable to predict that multi-threaded RMA may be used in future MPI programs. This work seeks to provide, to the best of the authors' knowledge, the first publicly available MPI RMA multi-threaded micro-benchmarks. These micro-benchmarks will provide the foundation to begin optimizing RMA thread multiple as application developers explore alternative MPI communication and threading models in the future.

2.1.2 Related work

Several MPI benchmark suites have been enhanced to support measuring MPI-3 RMA performance. The OSU Benchmark Suite from Ohio State University [93] supports several different measurements associated with MPI-3 RMA operations, including different window creation and synchronization methods. It also supports several benchmarks for the OpenSHMEM one-sided operations. However, it does not currently measure operations in the context of multiple threads. Likewise, the Intel MPI Benchmark suite [67] also has several benchmarks for measuring MPI-3 RMA performance and allows for measuring the impact of the different MPI thread levels, but does not currently measure performance involving multiple threads within an MPI rank.

Understanding the relationship between threads and the performance of communication operations has also been the subject of previous research. A test suite specifically for measuring the performance of MPI communication for multi-threaded processes was presented in [118]. This suite was used to measure the performance of MPI

point-to-point and collective communication functions in open source and vendor MPI implementations on three different platforms. More recently, the emergence of many-core processors has motivated closer examination of the interaction of threading, one-sided operations, and the need for achieving more concurrency from the network. Proposals have been made to better support thread safety and performance optimizations for threaded programs in OpenSHMEM [117], and a proposal for endpoints in MPI [113] seeks to offer enhanced network performance for multi-threaded MPI applications. Similar examinations are occurring for low-level one-sided communication layers as well, including extensions to the GASNet [52] networking programming interface. Several of the issues with extracting more concurrency from the networking hardware and software stack were explored in [64].

2.1.3 Benchmarks and mini-apps

This work introduces four micro-benchmarks and three mini-applications to evaluate different aspects of RMA performance and how it affects application performance. In this section, we describe the various elements of the resulting suite.

Benchmarks

The RMA-MT test suite includes four micro-benchmarks:

- latency,
- bandwidth,
- single direction message rate, and
- halo exchange message rate.

The goal of these micro-benchmarks is to measure the performance difference between multi-threaded RMA operations using the default locking scheme of MPI and a

reduced locking scheme. For each benchmark, four different synchronization methods (fence, PSCW, lock/unlock, and lock_all/unlock_all) and two RMA operations (Put and Get) are explored. Additionally, these micro-benchmarks allow evaluation of the effectiveness of multi-threaded RMA operations for a varying thread count and message size.

The RMA synchronization methods used in the micro-benchmarks cover all four common methods: fence, lock/unlock, lockall/unlockall, and post/start/complete/wait. It is important to note that RMA synchronization cannot be called on the same window from multiple threads. This is because the RMA synchronization is done at per-rank level. To avoid calling these synchronization methods multiple times per rank, thread-level synchronization is used. When each thread is launched, it updates a simple counter and waits for a broadcast from the parent thread, which signifies that all threads have been created and are ready to begin message transfer. Once each Put/Get thread is waiting, the original thread begins the timer, broadcasts to the Put/Get threads to continue, and runs the RMA synchronization. This method of timing is used to avoid measuring the extra time involved in creating and starting threads. While this is more idealized than would be expected in real applications, the overall thread creation overhead should be relatively small when using a thread pool for performing communication.

Latency RMA operations are not ideal for latency sensitive communications because of the high overhead of synchronization. Our latency tests measure the round trip time of a single message, including the cost of synchronization. Despite these shortcomings, a simple multi-threaded latency test is included in the RMA-MT benchmark suite, which provides some insight into the impact multiple threads has on message latency.

For this benchmark, each thread is launched and waits for a broadcast from the parent thread before beginning a single data transfer. This removes any artifacts

of initializing the threads. After the call to the non-blocking data transfer, each thread waits for an additional broadcast. Receipt of the second broadcast signifies that all child threads have completed a data transfer and that the initial thread has completed the RMA synchronization.

Bandwidth The bandwidth micro-benchmarks evaluate the potential bandwidth for different RMA synchronization schemes with a varying number of threads. The tests perform a large number of put or get calls between synchronization calls and bandwidth is measured over all iterations. For RMA operations, bandwidth is important to typical use cases because multiple data transfers can utilize a single RMA synchronization, amortizing the cost. The RMA-MT bandwidth micro-benchmarks do not "warm-up" the caches before commencing. Therefore, the resulting average bandwidth reflects this warm-up penalty.

Similar to the latency test, each thread launched by the parent thread waits for an initial broadcast, which signifies that RMA synchronization has occurred. This also signals that all data transfer threads have been launched and are ready to transfer data. Once receiving the broadcast, each data transfer thread performs multiple iterations of put or get to the shared target buffer offset by its thread ID. Following the completion of the data transfer operations, each thread waits for a second broadcast, signifying the completion of a closing RMA synchronization.

Message rate Message Rate is a subset of the RMA-MT micro-benchmarks, based on the Sandia Micro-benchmarks [28]. Single threaded, two-sided versions of the SMB's have been used in past work [9, 10]. These tests look at different message sizes, peer counts, and two different communication patterns. For this work, applicable communication patterns were extended to evaluate RMA synchronization methods, RMA transfer methods and multiple threads. Communication patterns dealing with two-sided specific communication were not relevant to RMA communication and were

not extended. The synchronization methods in these tests are fence, lock, PSCW, and lockall. The lockall implementation calls flush after every transfer operation, to provide multi-threaded progress.

Message rate (single direction) The single direction communication pattern looks much like the bandwidth test. It starts up sender ranks that communicate with a paired receiver rank. The primary difference between the two is that single direction uses larger number of ranks. It uses these ranks to test the communication of group of nodes, rather than being limited a single pair.

Message rate (halo exchange) The halo exchange test emulates application behavior by implementing a commonly used communication pattern. This test has every rank transfer data to a number of neighbors. In the default case, the benchmark communicates with six neighbors. Due to it's prominence in HPC applications, three of the four two-sided Sandia Micro-benchmarks use this communication pattern, with different variations in the manner and order in which sends and receives are posted. For the RMA-MT versions of the halo exchange tests, only one version was needed to map to RMA, since RMA does not have an unexpected message equivalent.

Mini-apps

This subsection presents the modifications made to a subset the Mantevo Suite [55]. We focused on three Miniapps: HPCCG, MiniFE, and MiniMD. These were selected to stress the diversity of problems that can use RMA and to stress the RMA components of an MPI implementation in different ways. HPCCG was implemented using the Lock_all/Unlock_all to test the most recent synchronization method. This was added in MPI 3.0 to support passive target RMA. MiniFE and MiniMD both use Fence as it fits well with the design of those miniapps and was performant in the microbenchmarks

tests, especially for MVAPICH.

HPCCG HPCCG is a conjugate gradient code focusing on the sparse iterative solver. It is designed to be very scalable and is approximately 3100 lines of code. It uses a halo exchange communication on a 27-point stencil. Therefore, this communication pattern is somewhat similar to the message rate halo-exchange micro-benchmark, however this mini-app performs calculation and only communicates at message sizes that are relevant to the computation. In order to adapt HPCCG to use RMA data transfers with multi-threading, each process spawns a communication thread per neighbor in the halo-exchange. Each process creates a thread for each of the neighbors it needs to communicate with and then uses that thread to drive the traffic solely to that neighbor. This means the number of threads created is not an independent variable for the results shown for this mini-app. The message size used in the `MPI_Put` is the same as those used in the `MPI_Isend` in the two-sided version of the mini-app. HPCCG uses lock_all/unlock_all synchronization semantics.

MiniFE MiniFE is a finite elements code and is similar to HPCCG because it focuses on a similar problem, but it has substantially more features. The code is around 8000 lines. Its main loop, much like HPCCG, is a conjugate gradient solver. Again, `MPI_Put` calls replace the `MPI_Isend`. MiniMD uses fence synchronization semantics.

MiniMD MiniMD is a molecular dynamics code focused on recreating the behavior of LAMMPS. The code is under 3000 lines. It's limited to Leonard Jones pair interactions. MiniMD uses two communication phases per iteration. The first being a forward communication, and the second being a reverse communication. In both cases, we have replaced the `MPI_Isend` call with `MPI_Put`. MiniMD like MiniFE uses fence synchronization semantics.

2.1.4 Experimental methodology

All of the tests were run on a Skybridge production cluster, which consists of 1,848 dual-socket nodes (totaling 29,568 cores). Each node contains 2X Intel E5-2670, 2.60GHz, 8-core processors with 64 GiB (32 per socket) of DDR3-1600, memory. Each node is connected by a Qlogic onload 4X QDR IB interconnect across a Fat Tree topology. The fabric utilizes three 648-port core switches and 108 36-port edge switches (both Qlogic).

There are two versions of MPI used for the tests. For MVAPICH, we used the 2.1 release downloaded from the official website. For Open MPI, we used a copy of the v2.x branch of the ompi-release candidate development repository on GitHub pulled on October 20th 2015. Both were compiled using Intels 15.0.4 compilers, and unless noted otherwise, were compiled with THREAD_MULTIPLE support. MVAPICH was compiled using the ch3:psm netmod, while Open MPI was given the runtime flags to specify the use of the openib Byte Transfer Layer (BTL), due to development issues with the PSM Message Transfer Layer (MTL).

All versions of the micro-benchmarks perform 100 iterations within each test. Each test was run 10 times. The results presented are the average of those 10 runs in each figure in Sections 2.1.5 and 2.1.6. All figures shown include error bars, although some may not be visible due to small standard deviations. For the miniapp runs, 3 runs were performed. All results are the average of those 3 runs, with applicable error bars for the standard deviation. While thread creation could be expected to introduce variance that would result in larger standard deviations than presented in the following sections, the overheads of thread creation and joining are not included in the performance results of the micro-benchmarks. As the overheads due to thread creation/destruction can be highly variable depending on the approach to threading used. For example, thread pools have lower overheads than on-demand creation/destruction of threads.

2.1.5 One-sided vs two-sided

Single threaded comparisons between one-sided and two-sided communication in MPI have been explored before [25]. We primarily include the results of Figure 2.1-Figure 2.4 to inform the reader of the baseline performance values for the system under test.

In these figures, we reduce the amount of information displayed by only presenting the best performing single-thread synchronization methods for one-sided communication. When reviewing the bandwidth performance of MVAPICH, the best synchronization is Lock_All, which has 2.1% greater throughput on average across all message sizes than the next best performing synchronization method (Lock). When comparing maximum throughput, Lock_All has a 4.0% increase over the next best method (Lock). In the case of Open MPI, the best performing synchronization in terms of bandwidth is PSCW, which is 1.3% greater on average across all message sizes than the next best method (Lock). The maximum throughput of PSCW is 1.4% greater than the next best synchronization method (Lock).

While the differences between one-sided synchronization methods are small for single threaded communication, we see more significant differences when comparing one-sided versus two-sided bandwidth. There are sudden dips in bandwidth for all the series, with the exception of Open MPI PSCW, as different eager/rendezvous thresholds are activated. The Open MPI revision used for these tests implemented RMA using two sided network calls and has since been updated. Of all the single thread techniques evaluated, Open MPI achieves the best peak bandwidth using one-sided PSCW, just surpassing 3 GiB/s.

In the case of the single threaded latency results, again, we only display the best performing synchronization method. For MVAPICH this is PSCW, which has 2.6% decrease to latency on average than the next best performing synchronization method

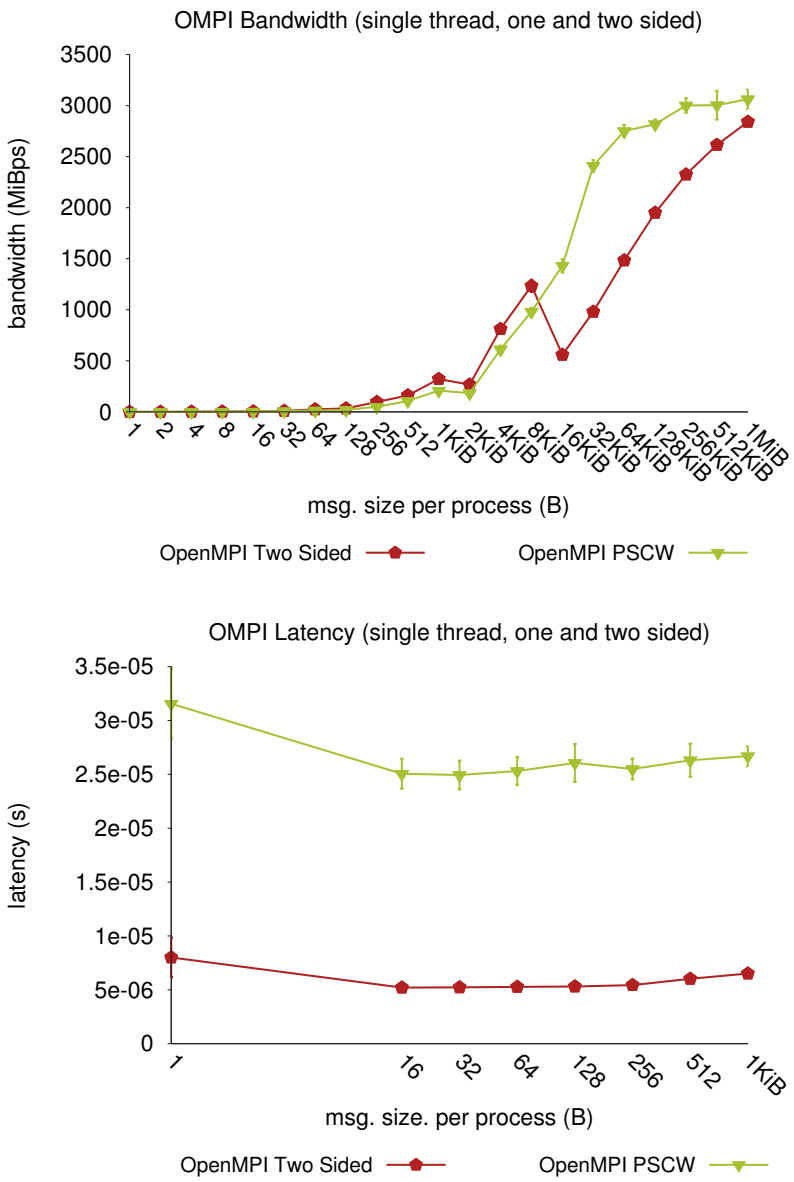

Figure 2.1: Single threaded one sided and two sided bandwidth and latency of Open MPI for varying message size.

(Fence). When comparing minimum latency, PSCW has a 5.2% decrease over the next best method (Fence). In the case of Open MPI, the best performing synchronization is also PSCW, which saw a decrease to latency of 12.4%, averaging across all message

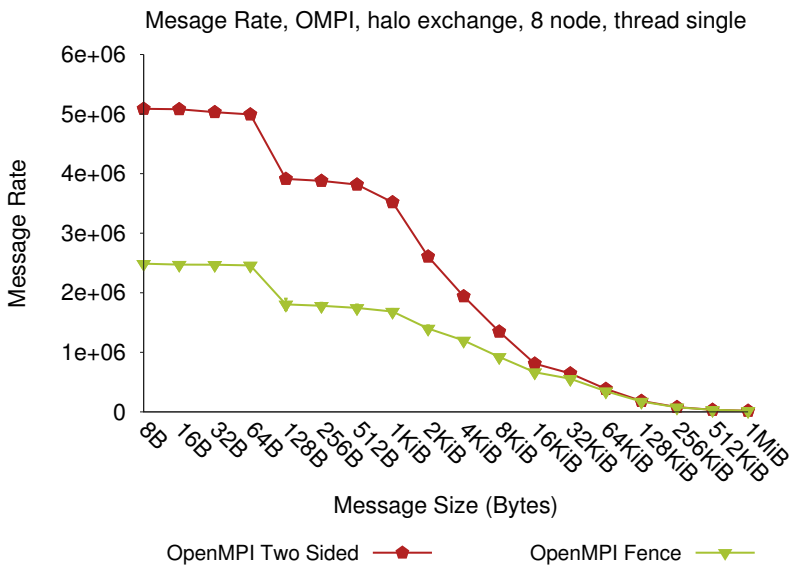

Figure 2.2: Single threaded one sided and two sided message rate of Open MPI for varying message size.

sizes than the next best method (Fence). The minimum latency of PSCW is 23% less than the next best synchronization method (Fence).

While our benchmarks sample message sizes at powers of two, we match the sampling of the original multi-threaded MPI benchmarks [118], which lack samples between 1 and 16 Bytes. In both cases of MVAPICH and Open MPI, the latency of one-sided operations is significantly worse than the latency of two-sided operations, which is somewhat expected as the maturity of the one-sided code in MPI implementations is significantly less than that of the two-sided communication code path. In these experiments the best observed latency was seen in two-sided MVAPICH (1.8μs).

For the message rate halo exchange results, we also only display the best synchronization method. For Open MPI, the best was Fence, which did an average of 1.7% better than PSCW and 12.9% better than Lock. For MVAPICH, Lock showed the best performance, doing 12.1% and 11.8% better than Fence and PSCW respectively. One anomaly in these results is the spike in MVAPICH Performance at 2KiB message

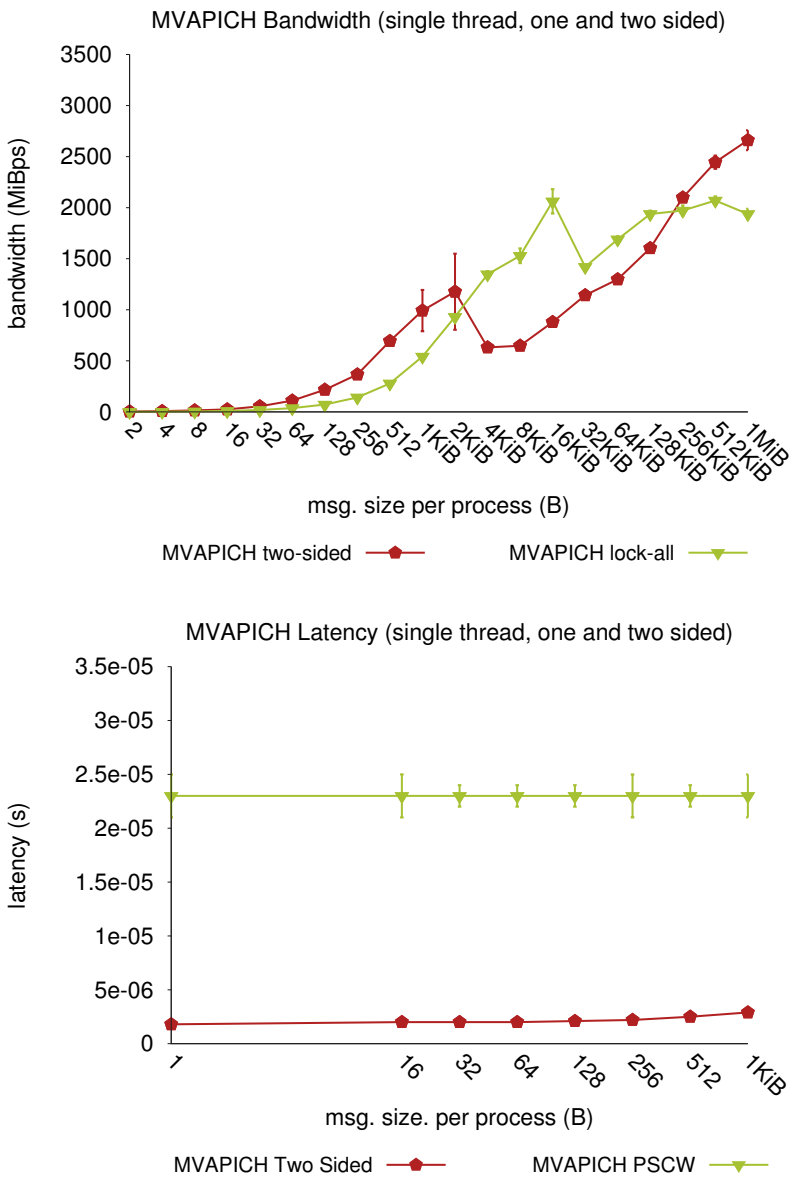

Figure 2.3: Single threaded one sided and two sided bandwidth and latency of MVAPICH for varying message size.

size. This is also observed in Section 2.1.6 and we will discuss it further there. For messages smaller than 2KiB, MVAPICH one-sided message rate performance exceeds the two-sided baseline. For those message sizes, MVAPICH Lock does an average of

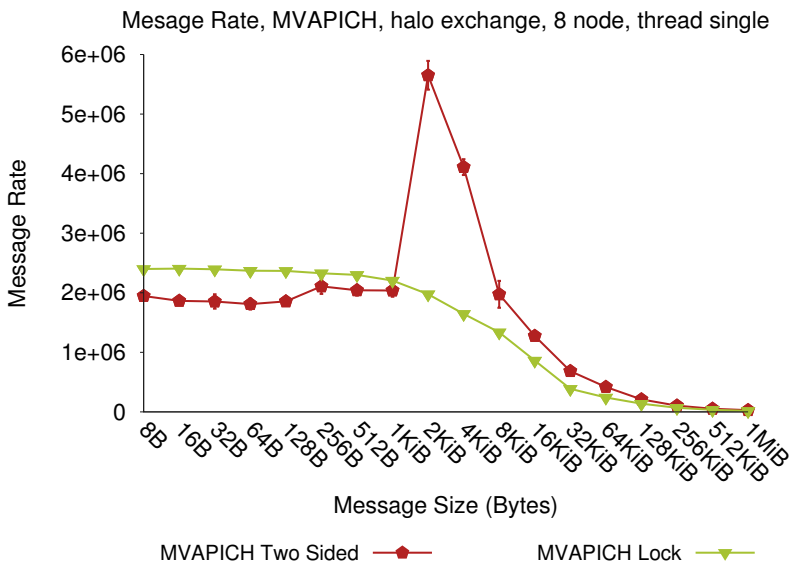

Figure 2.4: Single threaded one sided and two sided message rate of MVAPICH for varying message size.

21.4% better than the baseline.

2.1.6 RMA-MT benchmark results

In this section, we illustrate the use of our benchmark suite and how it can provide insights about the underlying system. We present and analyze the results of bandwidth, latency, and message rate benchmarks for varying message sizes, thread counts and MPI distributions. We have split these results into two separate sections by MPI distribution and caution the reader against making any comparison between the MVAPICH and Open MPI distributions for multi-threaded runs. These results should not be compared for two reasons. First, the evaluated version of Open MPI is not a released version and is currently under development. This version is not currently fully tested and on occasion our experiments fail. Of course, these failed runs have not been included in the performance results. We have included a discussion of

these failures at the end of this section to illustrate the ability of our benchmarks to evaluate correctness and functionality in addition to performance. Secondly, the system benchmarked (Skybridge) utilizes Qlogic onload network cards, which utilize the PSM interface in MVAPICH. For Open MPI, because we ran into functionality issues the RMA-MT in the PSM MTL, we used the OpenIB BTL instead. These interfaces represent different levels of optimization for the underlying hardware. Because of this, a comparison between the two distributions may be misleading, and as the goal of examining Open MPI was not to assess performance as much as it was to use the benchmarks and mini-apps to demonstrate their utility in debugging and improving MPI implementations in development.

MVAPICH release

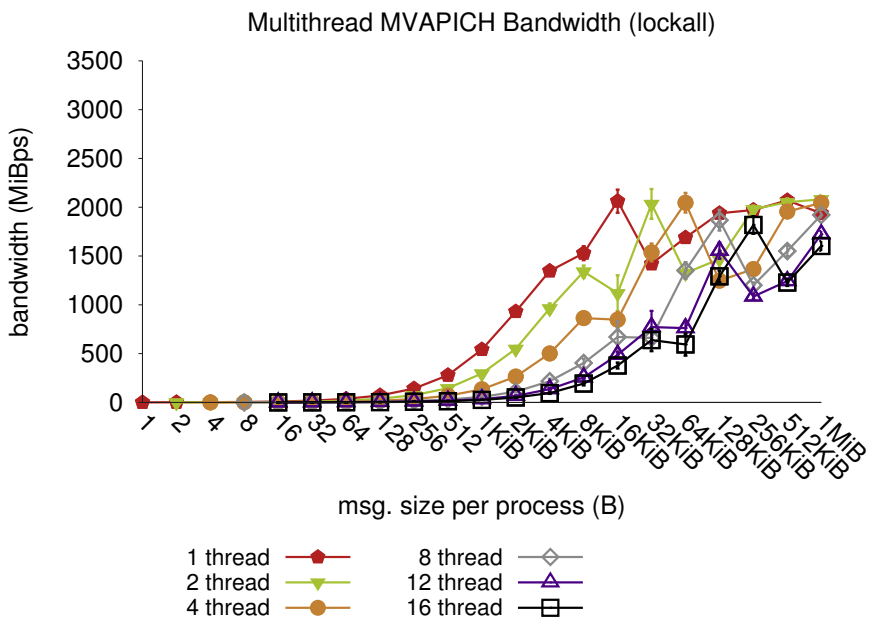

Figure 2.5: MVAPICH bandwidth results for one-sided (lockall) communication with varying thread counts.

Bandwidth Our results from MVAPICH only reported minor differences in throughput when comparing different synchronization methods. For brevity, we only include the plot for the Lock_All synchronization method because there is less than a 2% difference in average throughput across synchronization methods. As a disclaimer, we should note that the code paths for offload cards in MVAPICH have had more effort put into performance optimizations for one-sided communication, such that synchronization methods become a contributing factor to performance. In Figure 2.5 we see that the dips and peaks in throughput occur as each series reaches the same per-thread message size. To elaborate, as each thread reaches the point where it sends 16 KiB of data, we see a sharp reduction in throughput. This occurs at 16 KiB for single thread results 32 KiB for two threads, and so on. Overall as we increase the number of threads to 16, we see a 19% reduction in throughput. This occurs because the overheads of coarse grained locks that become larger as thread counts increase.

Latency The results of Figures 2.6(a) and 2.6(b) show that for small message sizes, there are significant differences to latency across the four synchronization methods and thread counts. Specifically, we see that PSCW and fence both outperform Lock and Lock_All performance significantly. PSCW achieves 44% and 27% of Lock_All latency at 1 and 16 threads, respectively. For PSCW and Fence, we see a increase to latency of almost 5X or 88 and 90 μs respectively, as we increase the thread count from 1 to 16 threads. Lock and Lock_All see an increase to latency of almost 8X or 303 and 365 μs respectively, as the number of threads increase from 1 to 16. Because there was not a significant difference in bandwidth across the synchronization methods in the previous section, the benchmark suite suggest that either PSCW or fence are preferred for the given system when using MVAPICH.

Message rate Figure 2.7 presents the results of the halo exchange message rate benchmark when run under MVAPICH. This figure shows the total message through-

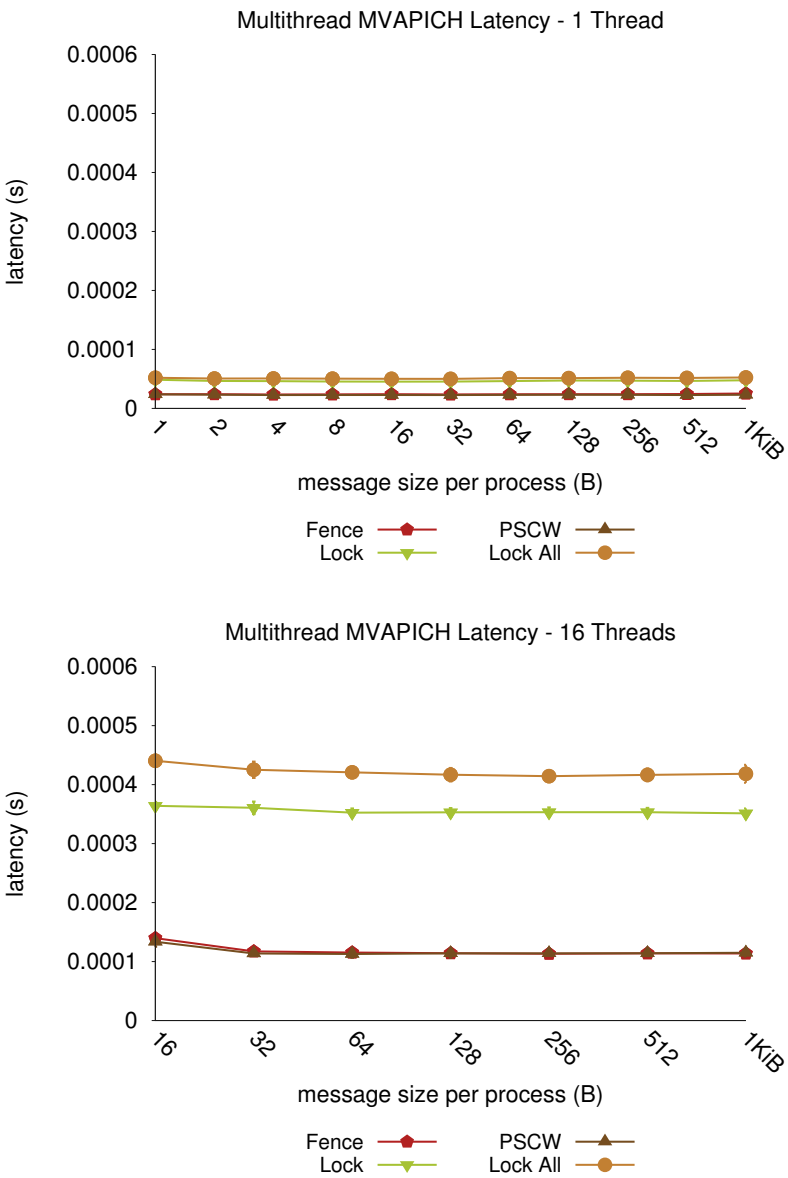

Figure 2.6: MVAPICH latency results for various one-sided synchronization methods and thread counts.

put of the benchmark for each of the synchronization methods. For space concerns only the RMA Get transfer mechanism is shown. It should be noted that the two sided baseline shown in each graph is a special case, as it is run under a single

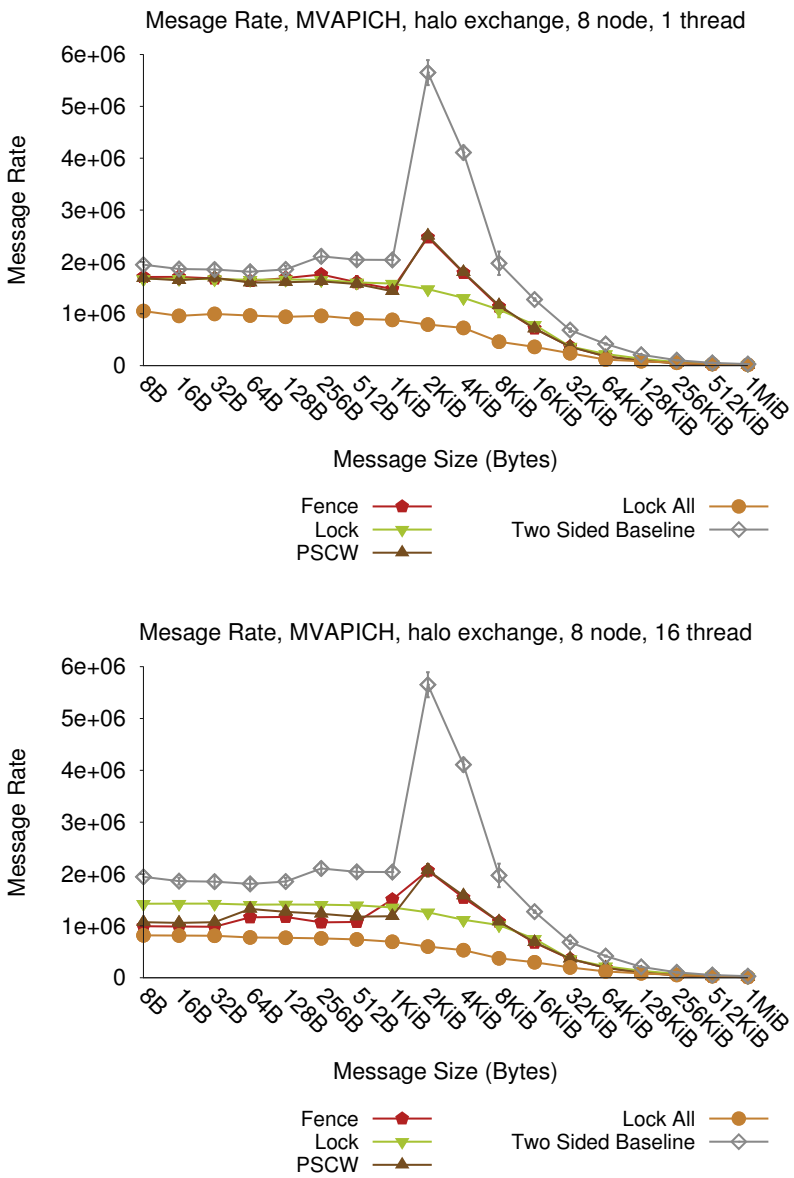

Figure 2.7: MVAPICH message rate comparison in a multi-threaded context

threaded instance of MPI where the multi-threading has been turned off at compile. This was done to compare RMA-MT to a current day implementation.

In Figure 2.7(a) we can see the effect of the extra process level synchronization of running under thread multiple. Fence and PSCW were very similar, on average there

was 2.3% difference between the two. For small messages under 2KiB, Fence, Lock, and PSCW did not show significant differences in message throughput. Lock-All on the other hand, performed significantly worse, averaging 49.5% throughput compared to the baseline, while the others averaged 85.9%. Fence and PSCW handled large messages the best out of all synchronization methods, with fence achieving a message rate that was 47.4% of the single threaded baseline.

Figure 2.7(b) shows the message rate throughput when run on a thread per core. As shown in the graph, large message rate throughput is roughly the same, which makes sense given that the bottleneck quickly becomes the network, rather than the MPI implementation itself. For small messages, there is a large reduction in performance for Fence, Lock/Unlock, and PSCW. Fence, for instance has average throughput of 68.2% compared to the version with one thread.

The most unexpected result from this series of tests is the spike in message rate for the baseline, Lock, and PSCW at 2KiB. While bandwidth is expected to fluctuate in both directions at the message size increases, message rate (which is normalized for message size should go down. The increase is unexpected, but has been confirmed in other work examining RMA message rate for MVAPICH2 [50].

Open MPI development branch

Bandwidth This section presents results about the performance for different synchronization methods, thread counts and message sizes of the chosen distribution. The keen reader may observe the lack of Lock_All data in this section. In our experiments, we found the Lock_All synchronization method of this development branch failed too frequently at high thread counts to confidently display results for, therefore it is excluded.

Examining the results of Figure 2.8(a)-2.8(b), it is evident that for single threads, the

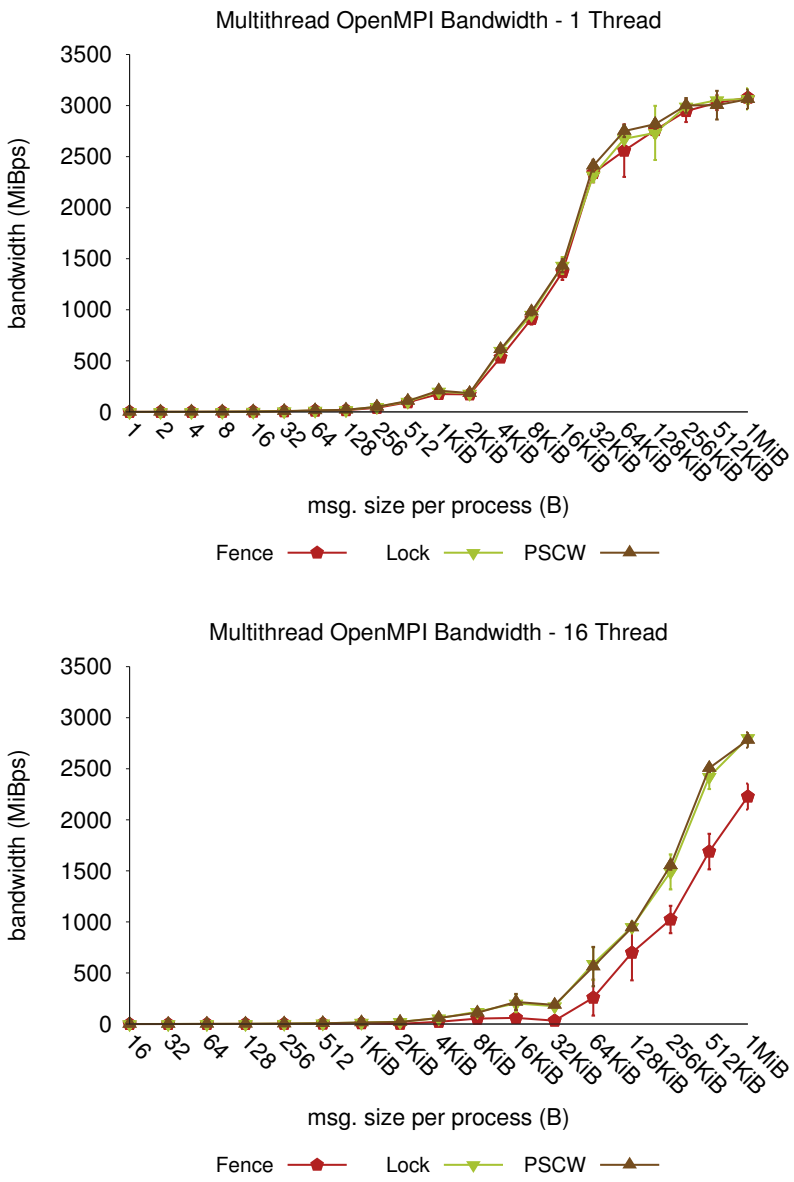

Figure 2.8: Open MPI bandwidth results for various one-sided synchronization methods and thread counts.

bandwidth at 1MiB is extremely close across the different synchronization methods (3065, 3062, and 3077 MiB/s for Lock, PSCW and Fence, respectively). However, we can see that when using 16 threads, synchronization method plays an important

role in the observed bandwidth, with Fence seeing a decrease of 573 MiB/s or 21% compared to Lock. Focusing on Fence, as we scale up the number of threads from 1 to 16, we see a decrease to bandwidth of 849 MiB/s or 28%.

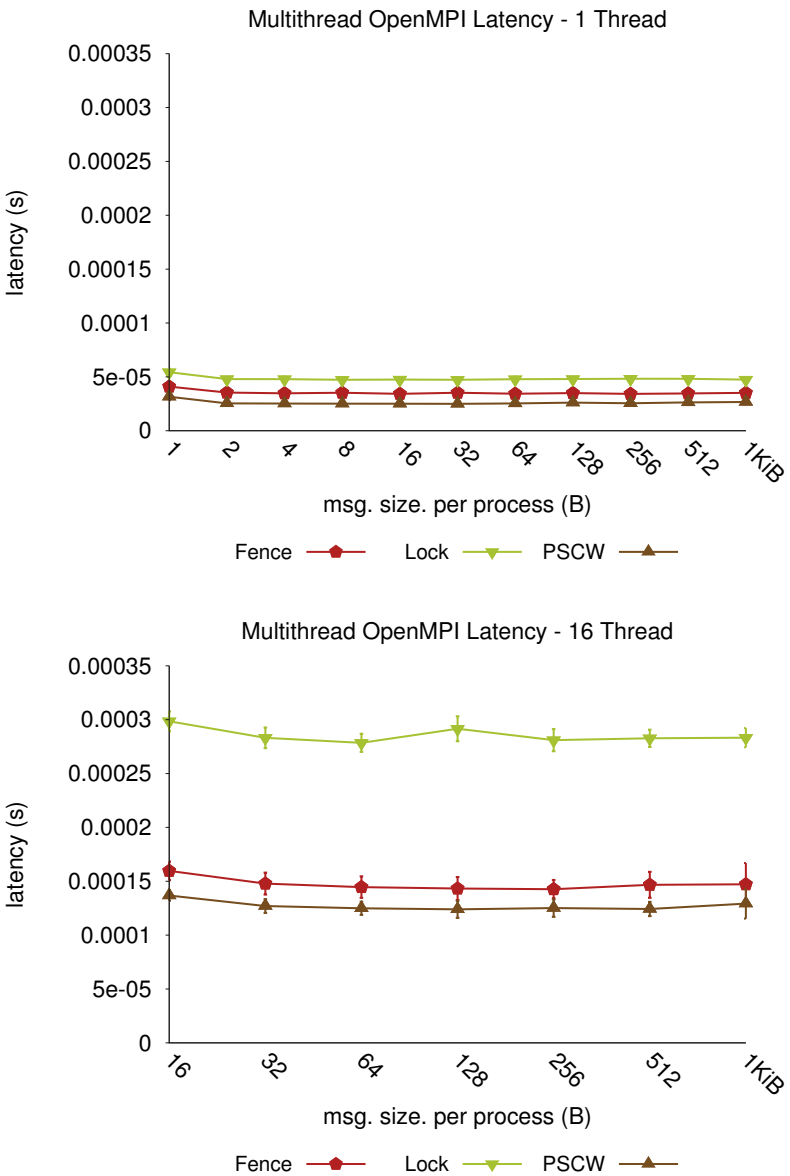

Figure 2.9: Open MPI latency results for various one-sided synchronization methods and thread counts.

Latency The results of Open MPI latency (shown in Figure 2.9(a)-2.9(b)) tell a different story than the bandwidth results earlier. For small message sizes (under 1KiB) we see that fence and PSCW provide significantly better latency at high thread counts, with PSCW providing the best latency overall. In the worst case (Lock), we see that as we increase thread count from 1 to 16, we see an increase of 235 μs or 6X. In the best case of PSCW, the increase is only 102 μs or 5X. Importantly, our benchmark suite shows that PSCW is preferred when using Open MPI to achieve the both the best bandwidth and latency.

Message rate Figure 2.10 presents the results of the halo exchange message rate benchmark when run under Open MPI. It should be noted that the two sided baseline shown in each graph is a special case, as it is run under a single threaded instance of MPI where the multi-threading has been turned off at compile. This was done to compare RMA-MT to a current best practices for running MPI. Again, Fence and PSCW performance was very similar (within 3.2% of each other). In this graph we can see the effects of using multiple cache lines, when we see the drop in performance from 64 byte to 128 byte message sizes. For small messages, the effects of RMA-MT are clear. Those effects have less impact as we increase message size. For example, Fence performs at 43.4% of the baseline for 8 byte messages. However, once we get up to 1 MiB, it performs at 96.4% the rate of the baseline. Figure 2.10(a) shows the message rate throughput when run with one thread per core. The trends here are strikingly similar to their single threaded counterparts, averaging a 1.3% difference overall.

Development branch failures The Open MPI micro-benchmark results for latency and bandwidth presented here consisted of 840 runs of our MPI benchmark, where each run performs 100 one-sided communications across 20 different message sizes. Because we were using a development branch of Open MPI we had a number

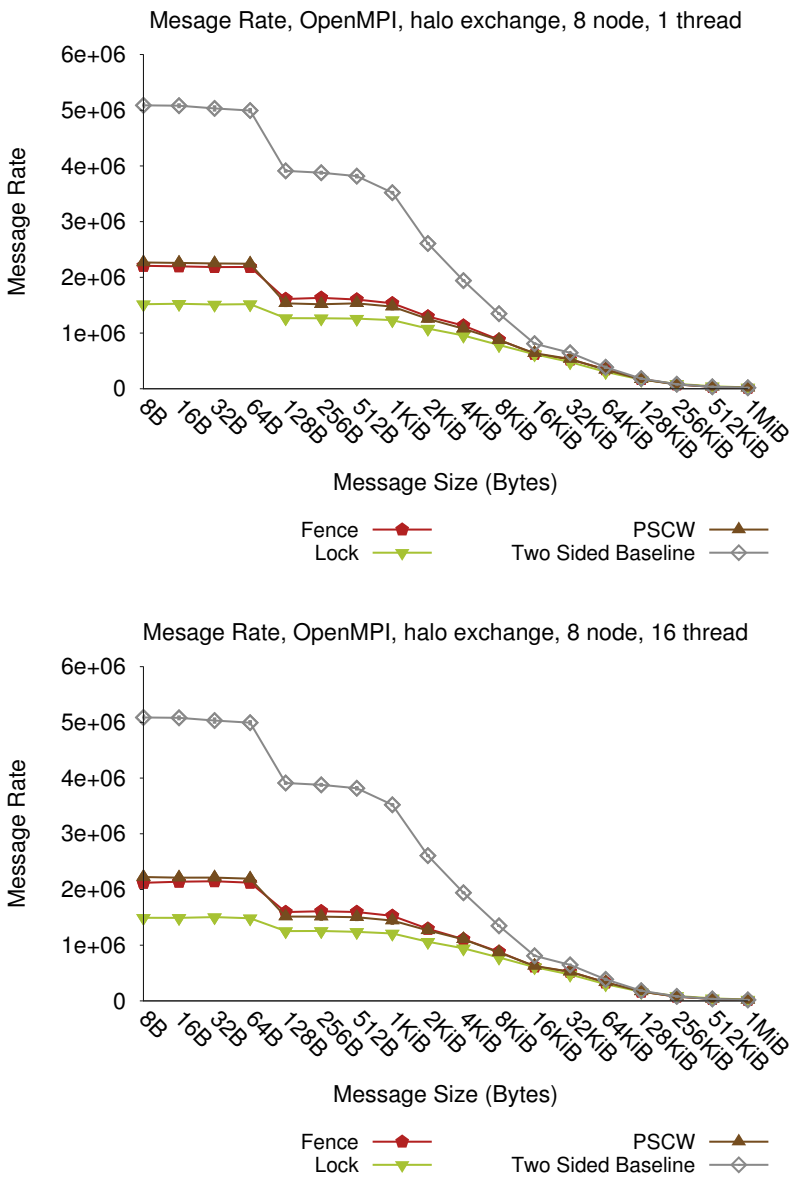

Figure 2.10: Single-threaded comparison of the different RMA operations

of runs where errors were detected. These errors were limited to multi-threaded runs and are enumerated as follows: three segmentation faults, 22 assertions, and 6 cases where the target or origin buffer did not pass a checksum, representing an error in less than 1% of the runs.

For the message rate micro-benchmarks, we ran roughly 2160 runs across all the combinations of message size, synchronization method, and transfer operation. We observed 81 failures in those runs. It should be noted that the message rate benchmark does more iterations than the other micro-benchmarks and thus has a higher probability of hitting an error. Of the errors we observed for message rate only 12.3% were associated with Get operations, only 16.0% were associated with single threaded runs, and only 19.8% were associated with message sizes less than 64 KiB.

As previously mentioned Lock_All saw a significantly larger number of errors, so was not included in our results. Fortunately, our benchmarks have brought these errors to the attention of Open MPI developers so that they may be fixed before release.

Discussion

Many of the results in this section show a degradation in performance when using RMA-MT. This degradation is due to a number of factors, one of the most apparent is thread level synchronization. Ideally, RMA would require little locking within MPI as it does not use most of the shared data structures such as the match list. However, examining the version of MVAPICH used for this study, a lock encapsulates the the entire call into MPI. This is due to multi-threaded RMA in MPI being underutilized and thus unoptimized. The benchmarks in this study provide performance data and RMA-MT capable code that MPI implementations may use to optimize their performance. In addition to synchronization costs, RMA performance has the potential to be degraded by contention for shared memory resources as seen in [42].

2.1.7 MiniApp results

This section presents the results from running the modified mini-applications. Each test was run with 16 ranks per node, had a weak scaling problem size, and had the

problem size adjusted to run for roughly a minute. The tests were run from 16 to 512 ranks using both MVAPICH and Open MPI. from HPCCG. Figure 2.11 graphs the performance of our tests normalized compared to the performance of the original non-RMA version.

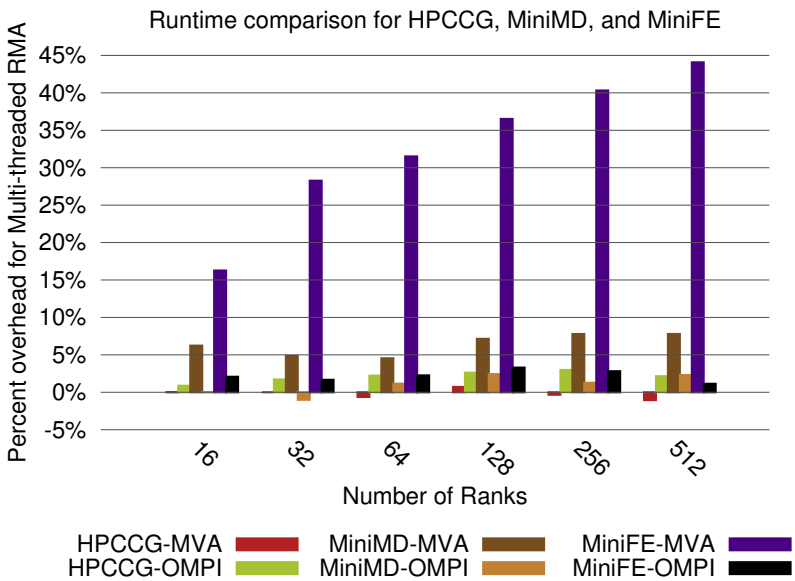

Figure 2.11: RMA-MT mini-app run time overhead compared to the regular version

HPCCG

For HPCCG, the results in Figure 2.11 demonstrate the RMA-MT overhead for HPCCG using one thread per communication between communicating rank pairs (neighbors). These runs used a 160^3 per rank problem size resulting in an average runtime of 57.8 seconds for the 16 rank MVAPICH baseline and 56.9 seconds for the 16 rank Open MPI baseline. As shown on the graph, the MVAPICH RMA-MT runs are very close to the baseline; because the standard deviation of these runtimes is often on the order of half a second, this performance difference is not statistically significant.

In contrast, the message rate halo exchange, which has an identical communication pattern, had performance difference was statistically significant performance gap. This is promising as it means that the performance gap left to bridge with application communication patterns may be less than that implied from the micro-benchmark results for future RMA-MT codes. This shows that the RMA-MT approach with Unlock_all/Lock_all is scaling well.

For Open MPI, we see a more significant increase in runtime of up to 2.9% at 256 ranks. It should be noted, that given the significant amount of errors in lock-all for Open MPI observed in previous sections, this result should be looked at skeptically.

MiniMD

For MiniMD, the results in Figure 2.11 show the RMA-MT overhead for MiniMD using one thread per communication between communicating rank pairs (neighbors). Our MiniMD implementation differs from our HPCCG implementation in that it uses Fence as the mechanic for window synchronization. These runs used a 150^3 per 16 ranks problem size resulting in an average runtime of 54.9 seconds for the 16 rank MVAPICH baseline and 54.5 seconds for the 16 rank Open MPI baseline.

Unlike HPCCG, the MVAPICH RMA-MT test show a large performance degradation from the baseline. This is due to the larger amount of communication calls in MiniMD, and the extra window synchronization required. Given this, we see an overhead of up to 7.8%. For Open MPI, we see a smaller change, of up to 2.4% overhead compared with the baseline.

MiniFE

Finally, Figure 2.11 shows the RMA overhead for MiniFE using one thread for each communication pair. The communication pattern is similar to HPCCG, as they both

are proxy apps for conjugate gradient problems; to differentiate them we have used the fence synchronization mechanism for MiniFE rather than lockall. The problem size that used for these tests was a 330^3.

The results for MVAPICH show the highest the RMA-MT overheads of any benchmark, ranging from 16.3% to 44.1%. While this is larger than the other mini-applications, it is not entirely unexpected. Both MiniMD and the message rate micro-benchmarks have significant overhead when using fence as a synchronization method. Because MiniFE uses a substantially larger problem than MiniMD, communication becomes more of a bottle neck. For Open MPI, MiniFE has an overhead of up to 3.0%, much smaller but again larger than the overhead that it had for MiniMD.

2.1.8 Outcomes of RMA-MT study

This work has presented the design of multi-threaded RMA micro-benchmarks and mini-applications. It has used the micro-benchmarks and mini-app developed to explore the performance of multi-threaded RMA on production systems, providing the first performance numbers available for such MPI usage models. Using the micro-benchmarks it was determined that up to 99% performance degradation can occur when using multiple threads to perform RMA operations for small messages in a current release of MVAPICH. However, there were a limited number of cases where multiple threads aided communication. The mini-apps saw a variety of performance effects; MiniFE and MiniMD both had a performance penalty when using MVAPICH. MiniFE in particular had a sizable penalty of up to roughly 44%. Open MPI saw less of a performance penalty for the miniapps which had a performance penalty of up to 4%. The slowdown using threads was not unexpected when compared to previous thread multiple studies [26] that have found similar multi-threading related slowdowns. However, unlike previous studies, this work has explored MPI RMA in a multi-threading context, which has fewer serialization requirements for ordering

guarantees than typical two-sided point to point communications in MPI. This work also demonstrated the use of this benchmark suite to drive development by testing functionality and correctness, in addition to the performance. This showed that the Open MPI development branch has a number of issues, ranging from triggering asserts to incorrect data transfer. The miniapps also have the ability to test functionality, correctness, and performance as the number of ranks and nodes is scaled up.

The micro-benchmarks used in this work have been open-sourced for use by the community and are available from `http://www.cs.sandia.gov/smb/`.

2.2 Prediction, Characterization and Prevention of Network-induced Memory Contention

In the previous section, we evaluated the performance of one-sided communication at the application level. To explore one-sided communication fully, we need to characterize the performance at lower levels of the network stack, where we can leverage passive data transfer. Remote direct memory access (RDMA), also called one-sided communication, is a popular and useful mechanism for implementing efficient asynchronous communication. RDMA allows a node's local memory to be read or written by a remote node without involvement of the target operating system or CPU. Such *out-of-band* communication incurs no direct overhead on the target CPU. There are many attractive use cases for RDMA, such as overlapping communication and computation phases in non-bulk synchronous parallel (non-BSP) computational paradigms, for in-memory asynchronous checkpointing and for in-situ analytics/visualization. However, little is known about the potential indirect application interference of the additional memory contention caused by RDMA communication. Consider, for example, uncoordinated[1] checkpoints that are staged

[1]The target node is unaware of when the origin node will be reading/writing a checkpoint

in the memory of remote nodes before being moved to stable storage, as in SCR [84]. Memory traffic from a remote node writing a checkpoint will contend with the memory transactions of a local memory-bound application. Generally, memory contention between local operations and out-of-band network operations can cause significant decreases in memory and thus application performance. We refer to this phenomenon as network-induced memory contention (NiMC[2]).

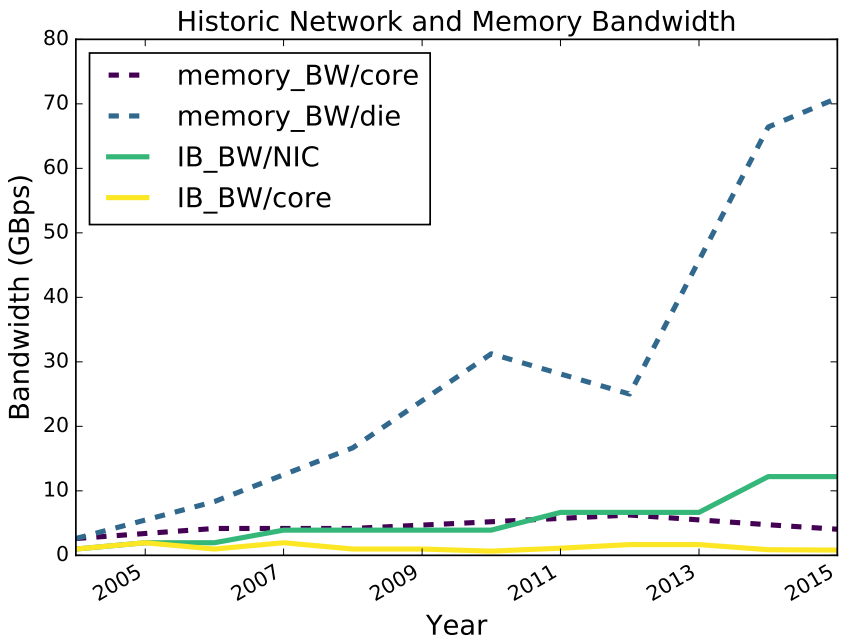

Figure 2.12: A 10 year history of network and memory bandwidths. While total bandwidth has been increasing, per-core network and memory bandwidth have not significantly increased. The result is increased contention for the shared network and memory resources.

Further exacerbating the situation, many-core technologies will be a fundamental part of the Exascale system design solution. However, a greater number of hardware threads means a greater demand for shared resources such as the network and memory sub-system. As shown in Figure 2.12, while the total off-chip bandwidth has been

[2] *nim* is an English form of the German word nehmen meaning "to take or steal". Using 'C' as "cycles", NiMC ($\backslash' nim -' s\bar{e}\backslash$) can also mean "*to steal cycles*" – the net result of network-induced memory contention.

increasing, per-core memory bandwidth has not been increasing as significantly. Coupling these trends with the increased interest in one-sided communication, *it becomes critical that we understand the potential application performance impact of NiMC*. While researchers have explored the memory subsystem as an area of concern for extreme scale systems [120, 15], the concept of NiMC was introduced by us as a part of the work included in this dissertation.

We explored several causes of NiMC – results that will inform future system design in both the existence of NiMC as well as the potential methods to reduce its performance impact. In addition, this study offers evidence to system software and application developers that NiMC should be taken into account when developing software that may be co-located with other applications or services that consume network bandwidth, even for small durations of time.

The specific contributions of this work are:

- detailed analysis of NiMC and its isolated effects;
- quantification of the performance impact of NiMC for a range of systems and applications;
- characterization of the system and application attributes that exacerbate NiMC, examining differences and commonality of these attributes across workloads;
- demonstration of machine learning to detect the presence/predict impact of NiMC with high accuracy.

The outline of this work is as follows. We start by providing a brief background on NiMC and machine learning using random forests. In Section 2.2.2, we provide an overview of related work followed by our evaluation methodology in Section 2.2.3. NiMC is characterized on many different systems and found to impact most of them in Section 2.2.4. Further investigations in Section 2.2.5, show the impact from NiMC on applications and mini-apps for one of the evaluated systems, showing an approximately

2X runtime increase for 5 of 7 of the studied workloads. In Section 2.2.6 we provide a more detailed characterization of NiMC and demonstrate the ability of random forests to detect NiMC and predict its impact. Afterwards, we prove that NiMC is an issue for some systems at scale with a real application in Section 2.2.7. Section 2.2.8 then demonstrates the effectiveness of our three proposed solutions at scale with tests conducted on two large systems comprising a total of 160,000 core hours worth of runtime. Throughout these investigations, evidence is provided showing that the slowdowns attributed to NiMC are not due to contention for compute or network resources. Finally, we discuss the outcomes of this study in Section 2.2.9.

2.2.1 Background

HPC communication

The HPC network communication stack contains dense layers of libraries, APIs, protocols, drivers, and hardware. The most common interface for HPC communication at the application level is the Message Passing Interface (MPI). MPI allows for message-passing and has been in use since 1991 and is the de facto standard for HPC applications. It provides synchronization, communication and establishes a virtual topology for multi-process applications. MPI sits on top of a range of physical fabrics and protocols. While many of the top 500 supercomputers use custom interconnects, as of this writing Infiniband networks make up 40% of HPC systems [121]. The majority of our work focuses on the Infiniband (IB) standard, because it is prolific and the standard on many of the systems available to us. Infiniband provides high bandwidth (96 Gb/s) and low latency (0.5 μs) and supports both copper and optical physical connections. Most Infiniband networks utilize the open source OpenFabrics Enterprise Distribution (OFED) stack which includes drivers, middleware, and user level interfaces for Infiniband. MPI translates application level communication

requests to low level Infiniband Verbs, which are transmitted and received by the Host Controller Adapter (HCA)/NIC.

RDMA

Traditional HPC communication is two-sided, that is both sender and receiver must actively participate (synchronize) to send and receive messages. In two-sided communication sender processes do not know where to place data in the receiver's memory. So a sender must first synchronize with the receiver before the receiver can copy data from the sender's buffer to it's own. If the processes are not perfectly synchronized, this results in delays as sending and receiving processes are forced to wait on each other. In contrast with two-sided communication, one-sided communication splits data transfer and synchronization phases. Essentially, a target process pins a *window* of memory so that a origin process may either read or write to the window. When the origin reads or writes directly through the target NIC, without involving the target CPU this is referred to as RDMA, Remote **Direct** Memory Access. RDMA is facilitated through a low level protocol such as IB verbs or RoCE (RDMA over Converged Ethernet). RDMA and RMA are not equivalent. When discussed at higher layer in the network stack, such as MPI or SHMEM one-sided communication is referred to as Remote Memory Access (RMA). An RMA programming interface provides a shared memory abstraction that may or may not be implemented on top of RDMA. Both RDMA and RMA may be referred to as one-sided. RDMA allows a node's local memory to be read or written by a remote node without involvement of the local operating system (i.e. kernel bypass) or CPU. We include an illustration of the data transfer path for RDMA in Figure 2.13. There are many attractive use cases for RDMA, such as to overlap an application's communication and computation phases in non-bulk synchronous parallel (non-BSP) computational paradigms, for in-memory asynchronous checkpointing and for in-situ analytics.

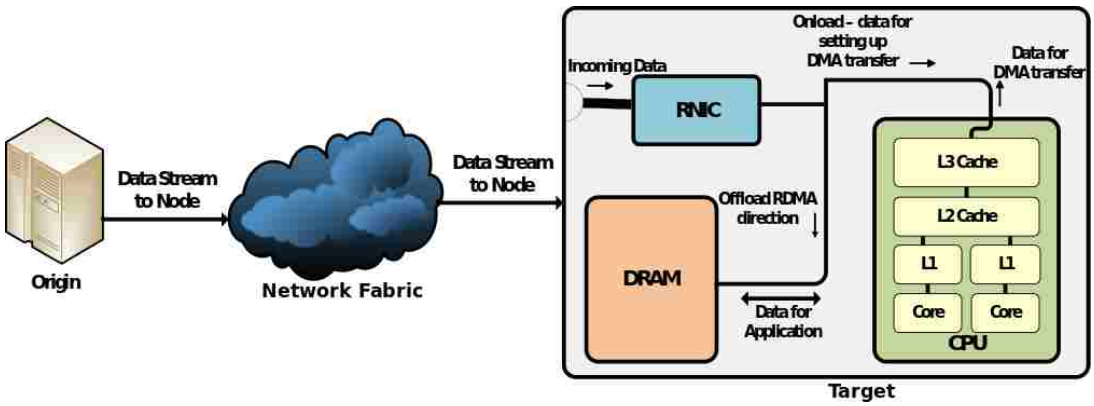

Figure 2.13: An illustration of the data transfer path for NiMC, note that the path for a fully offloaded networking approach does not involve the CPU, while the on-loaded networking approach requires some CPU intervention to setup the data transfer.

Network-induced memory contention

Remote memory operations can induce memory contention in two ways: (1) RNICs (RDMA-enabled network interface controllers) producing memory traffic in offloaded networks and (2) CPU-to-memory traffic when the CPU is used for onload network processing. For onloaded RDMA, not all traffic necessarily flows through CPU cores before being placed in memory. However, programming the DMA engines on the RNIC requires CPU intervention and causes some data to be transferred from the RNIC to the core to facilitate DMA requests. While there is ongoing debate for onloaded versus offloaded networking, this study reveals that NiMC should be an important consideration in this debate.

As Figure 2.12 illustrates, total network bandwidth is now much greater than per-core memory bandwidth. For future Exascale systems, this relationship is expected to become more pronounced [73], compounding the memory contention caused by RDMA operations. Trends away from traditional BSP programming models toward finer-grained, asynchronous models that admit higher levels of concurrency also will lead to to greater demands on the memory subsystem and the network. Lastly,

other Exascale requirements, for example application resilience, in-situ data analysis and uncertainty quantification, will generate additional local and remote memory traffic associated with activities only indirectly related to the application. NiMC can be particularly troublesome when caused by traffic not directly related to the computation.

RDMA traffic that causes NiMC has the potential to also impact application performance through congestion on the network fabric [116] (rather than memory). Therefore, this study uses methods of introducing NiMC that are realistic usage scenarios, but also ones that do not create additional contention on the network. Specifically, (1) the examination of NiMC impact on single node jobs (§2.2.5) does not utilize the network for application communication. (2) in §2.2.8 we show that the observed slowdowns at scale are caused by NiMC on the node, rather than in the fabric.

One of the additional challenges posed by NiMC is that it is difficult to detect. In our experiments we explicitly control the amount of RDMA traffic being injected on a target node. However, in a production environment the target node may not know how much RDMA traffic is being injected into its memory. Furthermore, some of the uncore[3] counters that might provide some insight are often unavailable. For this reason, we need to explore alternative ways of detecting when NiMC is occurring and predicting its impact on application performance.

Machine learning to predict the impact of NiMC

One of the challenges of NiMC is that it is not easily detected, since it originates from RDMA communication. The target node does not know when or how much RDMA data will be read or written to its memory. And while the target NIC does have counters that can expose the total number of bytes written, it does not specify

[3]Closely related to the core, but not directly part of, e.g. QPI or memory controller.

whether the data originated as an RDMA request. Furthermore, counters provided by the NIC require privileged access that is not always available. To detect NiMC and predict the impact to application runtime, we apply machine learning (random forests) to a set of easily accessible hardware counters.

Random forests Random forests [14] are an ensemble method of tree predictors, such that a collection of classifying trees with randomly selected feature-vectors vote to select the most popular class. Building a forest rather than developing a single tree the robustness of the predictions are improved. The benefits of Random forests is that overfitting is not a concern, given a large enough number of trees in the forest [14]. Additionally, random forests provide internal estimates of the generalization error, classifier strength and dependence with *out-of-bag* estimates [13]. An internal method of validation is included by default, as trees are trained on a subset of the input and are then validated against the remaining data. For this study we are interested in measuring the importance of the selected feature sets. Importance can be reported by the off-the-shelf random forests packages (python scikit-learn [98]). Additionally, once a random forest has been built using the training data, further classification can be done in parallel in logarithmic time with respect to the number of splits in each decision tree, allowing for real-time classification.

2.2.2 Related research

Memory contention has existed in many platforms over time, and tools have been developed to help understand its impacts [32]. Concerns about the ability of memory technologies to keep up with the bandwidth requirements of ever greater numbers of cores have been expressed by Rogers et al. [105]. However, code developers also deal with this issue through optimizing their code for cache use [99].This motivated investigations into the causes of off-chip memory contention by Tudor, Teo and

See [126] on modern systems for parallel applications, providing insight into the behavior of cores heavily optimized for cache performance and those that are less tuned. Given that architectures are designed to balance memory bandwidth with the ability of the cores to saturate said bandwidth, the impacts of NiMC can push a system on the edge, degrading optimized code performance.

Concerns over memory subsystem performance at extreme scale, particularly with the expected growth in core counts, have prompted investigations into memory bandwidth reductions [120] and contention [15]. Tiwari et al. [120] proposed a model for studying the anticipated reduction in per-core bandwidths expected in the Exascale time frame by varying the memory frequency on a single node of the Gordon supercomputer as SDSC. While Tiwari et al.'s model was motivated by a desire to explore the per-core memory bandwidth reduction expected for future extreme-scale systems, it was developed and tested on a single compute node. As such, the model does not account for any source of memory traffic from the network.

Casas and Bronevetsky's work [15] is the closest to this work in terms of memory contention studies. Like the Memory Bandit tool [32], they seek to create "memory bandwidth interference" in order to observe the impact on application performance. Unlike the Memory Bandit [32] work, Casas and Bronevetsky can purposefully perturb different levels of cache while introducing threads that create memory traffic unrelated to the executing application. They present methodology to introduce main memory traffic with minimal cache impact for studying off-chip memory contention. Their observations of application slowdown in the worst case of 20%-35% is inline with was observed from the STREAM/RDMA single node tests in our work. This is expected as their method of introducing interference for the application created main memory contention. However, the main difference between this work and ours is the source of the memory contention. While Casas and Bronevetsky introduce the contention from cores, they do not account for RDMA traffic.

In [11], Bhatele et. al used machine learning to identify sources of network congestion and their success inspired us to employ machine learning techniques in this work, though we are exploring different phenomena. In their work, they found that the ExtraTreesRegressor/Classifier package outperformed RandomForestRegressor/Classifier. We explored both packages and found only marginal differences in average cross validation scores, though our tests showed the RandomForest class to slightly outperform the ExtraTrees class for our data.

2.2.3 Evaluation methodology

In this section we outline our methodology for characterizing NiMC, detecting its presence, and predicting its impact on application performance.

The workloads

Throughout this study we use a variety of workloads to evaluate the impact of NiMC. In the text below, we include brief descriptions of each.

STREAM: The STREAM memory benchmark [82] performs a small set of memory benchmarking kernels (copy, sum, scale, triad) that perform a small number of reads, arithmetic operations and a write back to memory. We used these operations to measure the sustainable memory bandwidth and corresponding computation rates by working with data sets significantly larger than the available cache. We used the STREAM triad test, $a(i) = b(i) + q * c(i)$, which is the most representative of a typical workload. In some sections we specify either STREAM-DRAM (the standard) and STREAM-cache (a modified version that uses smaller arrays designed to fit in last level cache).

CNS: CNS [16] is a "simple stencil-based proxy-app for computing the hyperbolic component of a time-explicit advance for the Compressible Navier-Stokes equations using 8th order finite differences and a 3rd order, low-storage TVD RK algorithm in time." [34] CNS is intended to mimic the stencil operations of more realistic combustion applications, it does *not* mimic a typical problem found in combustion applications.

HPCCG: HPCCG [55] is an unstructured implicit finite element application, which calculates the conjugate gradient for a 3D chimney domain, running on an arbitrary number of processes. HPCCG creates a 27-point finite difference matrix, for which each MPI rank is designated a user-defined sub-block. This mini-app is generally considered to be memory-bandwidth bound, using from 25% to 75% of the total system memory. HPCCG is a designed to provide excellent weak scaling.

LAMMPS: LAMMPS [102], the Large-scale Atomic/Molecular Massively Parallel Simulator, is a molecular dynamics code modeling particles in different states. LAMMPS provides excellent weak scaling with the majority of communication occurring among nearest neighbors. In this work, we used a benchmark problem/data set to model the melt of a 3D Lennard-Jones system, using a weak scaled problem of a similar size to studies published by the authors of LAMMPS (32,000 atoms per core) [125].

LULESH: LULESH [68] represents shock hydrodynamics code solving a simple Sedov blast problem, illustrating the behavior of such solvers in ALE3D. This proxy-app distributes the spatial domain onto a set of volumetric elements defined by a mesh, where each intersection of mesh lines represents a node. Within the LULESH proxy-app, there are a variety of kernels, of which some subset are memory bound. One constraint of LULESH is that it must run with a cubic number of MPI Ranks.

Therefore, for our experiments we extracted additional parallelism by leveraging Open Multi-Processing (OMP) on any unused cores.

SNAP: SNAP [133] models the performance of a modern discrete ordinates neutral particle transport application. This proxy-app does not employ any real physics in its calculations. Instead, SNAP produces the computational workload, memory requirements, and communication patterns that match the Los Alamos National Laboratories application, PARTISN. To distribute larger problem sizes, SNAP spatially decomposes its 3D mesh and maps it onto a 2D domain of MPI ranks. MPI ranks send and receive data following wave propagation which limits the weak scaling of the proxy-app.

XSbench: XSBench [124] is a proxy-app which represents the most significant kernel (85% of runtime) in a robust nuclear reactor core Monte Carlo particle (neutron) transport simulation. This variety of simulation can have significant data usage requirements and the proxy-app is considered to be memory-intensive. XSBench focuses on modeling intra-node performance characteristics of OpenMC and is not intended to be run at scale, as communication is limited to a single reduction at the end of a run.

The platforms

Our study used nine different platforms from the Sandia National Laboratories, University of New Mexico and the Texas Advanced Computing Center consisting of a variety of architectures. We provide a concise description in Table 2.1 and Table 2.2 for the reader's convenience.

For a subset of the machines (Westmere, Lisbon, and Piledriver-1600/1866), we performed our experiments with varied memory frequencies, allowing us to see the

Table 2.1: Evaluated Architectures (1 of 2)

machine	nodes	kernel	CPU
Westmere	1	3.2.0 (Ubuntu12)	Intel E5620
Lisbon	1	3.13.6 (UN12)	AMD 4170 HE
Piledriver-1600	70	2.6.32 (RHEL6)	AMD A10-5800K
Piledriver-1866	2	2.6.32 (RHEL6)	AMD A10-5800K
Sandy Bridge-X2-FDR-offload	6400	2.6.32 (Cent6.3)	2× Intel E5-2680
Sandy Bridge-X2-onload	1196	2.6.32 (RHEL6.2)	2× Intel E5-2670
Xeon-Phi (on-chip bandwidth)	49	2.6.38.8+mpss3.1.2	Xeon Phi 3120P
Haswell-X2	33	3.14.23 (RHEL6.5)	Intel E5-2698

Table 2.2: Evaluated Architectures (2 of 2)

machine	cores	channels	DRAM	DRAM GB/s	Network
Westmere@(800, 1066 MHz)	4	2	16GB	12.8, 17.1	QDR IB off
Lisbon@(800, 1066, 1333 MHz)	6	2	16GB	12.8, 17.1, 21.3	QDR IB off
Piledriver-1600	4	2	16GB	25.6	QDR IB on
Piledriver-1866	4	2	64GB	29.9	QDR IB on
Sandy Bridge-X2-FDR-offload	8	4	64GB	85.3	FDR IB off
Sandy Bridge-X2-onload	8	4	64GB	102.4	QDR IB on
Xeon-Phi (on-chip bandwidth)	57	12	6GB	240	QDR IB off
Haswell-X2	16	4	128GB	136	FDR IB off

impact available memory bandwidth has on NiMC. Westmere and Lisbon required BIOS option changes, whereas the Piledriver systems are separate nodes with different memory modules.

All of the systems, other than Haswell-X2 and Sandy-Bridge-X2-FDR-offload, utilize an InfiniBand QDR network with a maximum speed of 32 Gbit/s. Within the Table 2.1 in the network column *on* and *off* signify whether the NIC is an onload or offload NIC. The observed bandwidth of these systems varies with the physical network topology and the degree of contention. The general observation is that larger production clusters (e.g. Sandy Bridge-X2-FDR-offload) tend to have a larger variability in observed bandwidth, since network resources are shared among multiple jobs which may compete for bandwidth. However this topic is beyond the scope of our work and we refer the reader to [12] for further discussion of performance degradation due to nearby jobs.

Characterizing NiMC

Our approach to measuring the impact of NiMC is straightforward: for each application and hardware configuration under test, we injected remote memory operations into the compute node(s) and measured the resulting application perturbation by comparing application performance with and without RDMA traffic.

First, we used a memory-bandwidth benchmark to establish a baseline for application perturbation. These experiments were executed on multiple architectures to observe the NiMC impact for different hardware configurations. Our subsequent experiments then helped us to assess NiMC impact for real applications in single node and distributed contexts.

Injecting RDMA operations: We used *ib_write_bw* from the Open Fabric Enterprise Distribution (OFED) Performance Tests [95] to generate network traffic streams. The OFED Performance Tests are a set of tests that use InfiniBand's user-level verbs API to measure IB performance. The *ib_write_bw* test uses RDMA writes to perform a series of operations between two connected nodes. As previously described, compared to traditional send/recv tests, the benefits of one-sided tests are that message delivery and synchronization are decoupled: after memory registration, writes do not significantly involve the CPU on the target node. We chose write operations as the overhead of performing read operations on the nodes running the applications potentially would perturb the tests through use of compute cores to create and issue the read requests. Write requests do not have this issue, as the source node bears the burden of creating and issuing the network requests. An illustration of the flow of traffic to an individual node is presented in Figure 2.13.

Detection and prediction of NiMC

After broadly characterizing NiMC on a range of architectures, we explore the detection of NiMC and prediction of its application impact. Namely, given commonly available performance monitoring counters can we (1) detect the presence of NiMC on onload NICs? (2) determine the volume of RDMA traffic? (3) determine the impact of NiMC on application CPU time?

We performed a more detailed analysis of NiMC, focusing on the Sandy Bridge-X2-onload cluster (which was a heavily impacted, largescale system). We applied ensemble machine learning (random forests) to examine a range of recorded performance counters (features) of approximately 60,000 processes, spanning five benchmarks and applications. We use random forest to answer the three questions enumerated above. If we can answer these questions, we may then enact a mitigation strategy. Beyond these important questions, we also evaluated the relationship between each application and the 18 reported counters/features. We looked for any features that were universally important across all applications when answering questions 1-3. In several instances, the features of highest importance surprised us, which shows the value of a principled approach such as random forests, since as system experts we might have allowed our intuition to lead us to selecting features that were not as rich in information. For the system evaluated, we were only able to collect a subset of performance counters for any given run (exactly, six non-derived counters), where a non-derived counter may be L2_dcm and a derived feature could be (l2_dcm/l2_tca). This restriction is due to hardware constraints of the CPU, which determine the overall number of reportable counters as well as incompatibilities between amongst multiple counters. For this reason we divided the experiments up into three feature sets, such that each set represents a different selection of counters. Furthermore, we are limited by the Performance Application Programming Interface (PAPI) counters made accessible to users on the system.

All of our experiments using random forests are performed with single-node runs of the application or benchmark, with a secondary node that writes RDMA traffic into the system. We avoid multi-node application runs because we want to remove outside sources of contention and noise that are known to impact system runtimes. This allows us to isolate the effects of NiMC and provides a clearer picture of what features are most indicative of a true, NiMC perturbance. Each experiment runs 200 times for each feature set, with and without added RDMA traffic.

When performing the machine learning, you must have a large enough forest to develop accurate predictions. We used 100 estimators[4] (trees in the forest), with separate runs for each feature set. That is, we ran the regression for each feature set in isolation. We use out of bag (OOB) samples to estimate the generalization error. The OOB score is the average error of observations from a sampled subset of observations. Where each observation, is evaluated by forests not trained on that particular observation. If we wanted to combine feature sets from multiple runs so that the trees were built considering all 17 features, this would be possible by substituting in the average value of a feature for runs where it was not recorded. However, increasing the number of features, increases the runtime of the learning algorithms, in that a greater number of trees must be constructed for accurate results. Additionally, given each feature set in isolation, our results were accurate enough (demonstrated by the OOB scores), so that using averages was unnecessary.

Features For our random forests, we recorded 17 features that are commonly available on most systems (displayed in Table 2.3). Alongside each feature name is a description of what it measures. We record each feature with respect to each process, which creates 6,400 samples per feature set (400 in the case of STREAM since it uses OpenMP). These features are not comprehensive of all the hardware counters PAPI provides but they do represent the supported preset events on the

[4]Trials of 300 trees did not show meaningful improvements to OOB scores.

Table 2.3: Features used in machine learning.

Set 1
icy: Cycles with no instruction issue
l1_dcm: Level 1 data cache misses
l1_icm: Level 1 instruction cache misses
tlb_dm: Data translation lookaside buffer misses
tlb_im: Instruction translation lookaside buffer misses
ib_bw: Average Infiniband write bandwidth (MBps)
Set 2
l2_dcm: Level 2 data cache misses
l2_dch: Level 2 data cache hits
l2_icm: Level 2 instruction cache misses
l2_ich: Level 2 instruction cache hits
l2_tcm: Level 2 total cache misses
l2_tca: Level 2 total cache accesses
l2_dcm/l2_tca: Fraction of level 2 data cache misses to total cache accesses
l2_tcm/l2_tca: Fraction of level 2 total cache misses to total cache accesses
ib_bw: Average Infiniband write bandwidth (MBps)
Set 3
l3_tcm: Level 3 total cache misses
l3_tca: Level 3 total cache accesses
l3_ica: Level 3 instruction cache accesses
l3_dca: Level 3 data cache accesses
ib_bw: Average Infiniband write bandwidth (MBps)

SandyBridge-X2-onload system.

2.2.4 A memory-bound benchmark

In our first NiMC experiments, we used the memory-bound, synthetic benchmark, STREAM (§2.2.3). These experiments were used to assess NiMC impact for worse-case memory intensive applications and to evaluate what architectural features can lead to increased NiMC impacts.

STREAM used one OMP thread per core in order to saturate the available memory bandwidth. A first-touch[5] memory allocation policy was used to optimize memory utilization within the NUMA hierarchy. Each experiment comprised 10 STREAM

[5]Memory allocation is done when the memory is accessed rather than malloced in Linux. On a NUMA architecture you must take this into consideration to ensure performance.

executions with 300 iterations of the triad kernel per execution, with each iteration taking a few milliseconds of walltime. The average sustained bandwidth was then calculated based on the average time to complete a triad operation.

To measure the impact NiMC has on memory bandwidth, we re-executed the same set of experiments; this time, our *origin* node continuously injects 64 KiB of data (using *ib_write_bw* as described in §2.2.3) into a buffer allocated on the different *target* node on which the STREAM benchmark was running.

The results, shown in Table 2.4, illustrate varied NiMC behavior, dependent on the underlying architecture. This performance degradation ranged from 0% to 60%. On all systems other than Xeon-Phi, Sandy Bridge-X2-FDR-offload and Haswell-X2, STREAM experienced significant (greater than 20%) memory bandwidth degradation due to NiMC. The Piledriver-1600 system and Sandy Bridge-X2-onload stand out by exhibiting a 60% and 51% performance degradation, respectively. The increased 9% degradation on Piledriver-1600 is expected due to a higher network bandwidth to memory bandwidth ratio compared to the Sandy Bridge system. We observe that all three of the systems with variable memory frequency see a decreased impact from NiMC as available memory bandwidth increases.

Of greater interest, our Sandy Bridge systems demonstrated how NiMC might impact onload versus offload networks differently. Both systems utilize Sandy Bridge CPUs, but Sandy Bridge-X2-FDR-offload utilize Mellanox offload network cards, whereas Sandy Bridge-X2-onload uses QLogic onload NICs. There are stark differences between the STREAM Triad results of these two machines. While the offload system sees no performance degradation, we see a 51% performance penalty to the onload system's Triad performance. We observe that the onload-NIC systems (Piledriver-1600/1866 and Sandy Bridge-X2-onload) are the most impacted by competing RDMA flows.

Examining the four offload NIC systems in isolation, we see a bi-modal impact of

Table 2.4: STREAM Triad Bandwidth (GB/s) with and without RDMA-NiMC

machine	Triad no RDMA	Triad w. RDMA	diff.	diff. %
Westmere @ 800 MHz	12.9	9.7	-3.2	-25%
Westmere @ 1066 MHz	16.8	12.8	-4.0	-24%
Lisbon @800 MHz	14.0	10.8	-3.2	-23%
Lisbon @1066 MHz	17.9	14.3	-3.6	-20%
Lisbon @1333 MHz	19.7	16.5	-3.2	-16%
Piledriver-1600	12.4	7.4	-5	-40%
Piledriver-1866	12.7	5.6	-7.1	-56%
Sandy Bridge-X2-FDR-offload	77.8	77.6	-0.2	0%
Sandy Bridge-X2-onload	73.4	36.1	-37.3	-51%
Xeon-Phi (on-chip bandwidth)	126.4	121.7	-4.7	-4%
Haswell-X2	116.6	116.9	0.3	0%

NiMC. When the CPU fully utilizes close to the theoretical memory bandwidth (as is the case of Westmere and Lisbon), competing RDMA traffic can degrade the Triad performance by 16-25%. Westmere and Lisbon, show effective memory bandwidth of 98% and 93% the theoretical memory bandwidth, respectively. This compares to Sandy Bridge and Haswell, which only achieve 74% and 85% effective memory bandwidth, respectively. The decrease to percentage effective memory bandwidth in Sandy Bridge and Haswell leaves additional headroom for the RDMA traffic, so that we see no impact of NiMC. However, as increases to theoretical memory speeds slow down, chip designers must increase their effective utilization in future systems. Furthermore, network speeds are increasing rapidly, with 4X EDR InfiniBand reaching speeds of 12 GBps so that we will again see larger network to memory bandwidth ratios. For these reasons, we cannot rule out the resurgence of NiMC in future offload systems.

For the Intel Xeon Phi (KNC) system, we observed a small (4%) memory bandwidth decrease. This is due partially to the Phi's unique memory architecture (as compared to traditional CPUs). For instance, all other systems under tests had a single memory controller. The Phi system has eight controllers that control 16 channels of GDDR5 memory. Each core has 64K of L1 cache and a 512k fully coherent L2 cache, which are connected over a bi-directional ring interconnect. Additionally, the Phi runs its own operating system, requiring a dedicated core for OS services. By leaving this

reserved core open, we may be leaving additional memory-bandwidth-headroom into which the RDMA traffic can fit.

In summary, we saw that NiMC impacts a range of architectures spanning multiple vendors and hardware generations. Of importance, the NIC architecture (onload vs offload) appears to play a significant role determining the impact of NiMC, as every system utilizing an onload NIC saw significant degradation. Additionally, as one might expect, the results suggest that increased available memory bandwidth reduces NiMC interference, while decreased available memory bandwidth increases NiMC interference.

2.2.5 Proxy-apps on a single node

While STREAM illustrated NiMC effects for worst-case memory-bound applications, STREAM is not necessarily reflective of typical HPC applications. By mimicking the operations of a variety of important scientific problems, the DOE proxy applications (described in §2.2.3) are more accurate HPC workload representations. We now describe our use of the proxy apps to understand NiMC effects on a single node for realistic workloads with worst-case network traffic.

We ran our workloads on single nodes to study NiMC effects in isolation from other potential interference on the network fabric. The applications use Open MPI v1.8 for inter-process communication. Additionally, LULESH and XSBench use OMP for increased on-node parallelism. For these latter hybrid applications, we used the highest performing combination of MPI processes and OMP threads[6]. Once again, we measured application execution times with and without injected RDMA traffic. Reported results are the averages of 10 runs with error bars displaying the standard deviation. For the experiments in §2.2.5 and §2.2.7, we used the Sandy

[6]For LULESH this was 8 MPI ranks (4 per socket) with 2 OMP threads per rank and for XSBench this was 16 Ranks with 1 OMP thread per rank.

Table 2.5: Application Performance with and without RDMA

Application	Run time no RDMA (s)	Run time w. RDMA (s)
CNS	8.75	8.89
HPCCG	4.08	6.07
LAMMPS	177.21	372.04
LULESH	23.07	44.08
SNAP	3.17	5.97
XSbench	72.92	146.44

Bridge-X2-onload system.

Table 2.5 shows the application time-to-solution results in the absence and presence of NiMC, and Figure 2.14 shows the application performance slow-down due to NiMC. These results illustrate several interesting behaviors. First, out of six proxy-apps, we observed significant performance penalties in five, but CNS exhibited almost no performance degradation. Second, three proxy-apps exhibited a performance degradation within 30% of that seen in STREAM. This was unexpected because we selected STREAM as a worst-case indicator of NiMC, due to its intensive memory usage. Furthermore, our pre-experiment hypothesis was that memory-intensive proxy-apps, like HPCCG, would experience the most interference among the proxy-apps; however HPCCG exhibited the second least relative interference. These results suggested that additional sources of contention, beyond the memory-channel bandwidth, may be influencing performance. Accordingly, our next set of experiments were aimed at uncovering these additional contention effects.

Pearson's R of runtime, stalls and cache counters

It can be hard to decipher precisely what is happening in a system with regards to NiMC. Often, to maintain competitive advantages, hardware vendors do not publicly share the details of components such as memory-controllers or network drivers. While it can be difficult to determine NiMC root causes, we glean insights by profiling application activity as measured by available performance monitoring

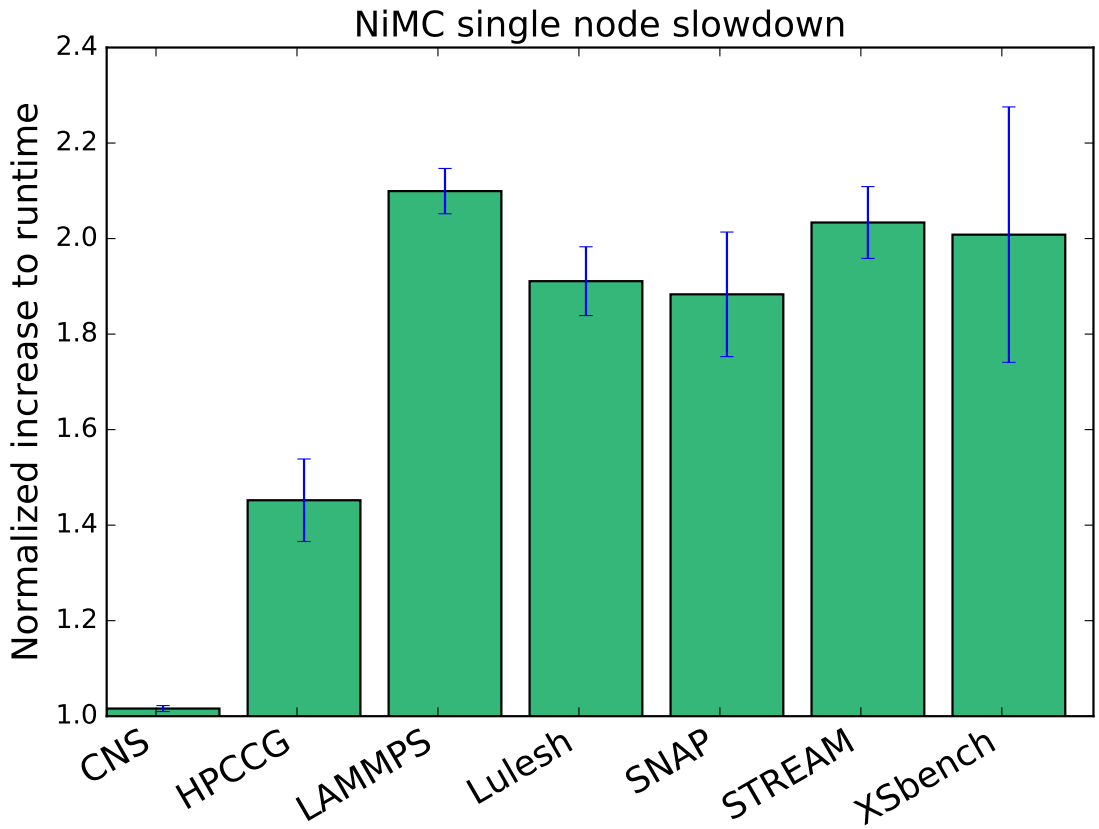

Figure 2.14: Normalized impact of NiMC on single node runs.

counters (PMCs). In this case, we use OpenSpeedShop (OSS) [109], which can provide PMC collection and analysis for large parallel applications. Using OSS, we found that for all but CNS (the sole application that did not exhibit a NiMC-based performance degradation), with RDMA traffic, NiMC impacted one core more than the other cores.

To understand why this process was performing much slower under RDMA activity, we used OSS to measure the performance of the L1, L2 and L3 caches as well as the total number of stalled cycles in the presence and absence of RDMA traffic. With the exception of CNS, we saw increases to L1 cache misses and, to a lesser extent, increased L2 cache misses. Additionally, there was a significant increase in the number

of stalled cycles in the presence of RDMA traffic. The relationship between cache misses and stalled cycles is consistent with Figure 2.14 in that proxy-apps that have more reasonable cache efficiency experience the worst NiMC interference. At the same time, applications with poor cache utilization are less affected by additional cache misses since their cache efficiency is low to begin with. These results also help to explain why only one application process experienced significant performance degradation: the largest differences in cache behavior were observed at cache levels not shared by other cores.

Among the proxy-apps, CNS is an outlier in that it experiences almost no interference from NiMC. CNS demonstrates that the slowdown seen in other runs is not primarily due to CPU (core) perturbance, as CNS receives identical amounts of RDMA traffic and services it in the same manner as the rest of the benchmarks, but observes only a 1.6% slowdown from RDMA traffic processing overhead. This is because in CNS, each MPI process utilizes an extremely small amount of memory (approximately 4 MB). Such a small working set of memory leaves space for both the one-sided RDMA and the proxy-app to effectively utilize cache. As a result, there is only a minor increase in the amount of idle cycles as we add interference from the network.

As Figure 2.13 illustrates, as a DMA transfer is serviced by an onload NIC, some amount of data is distributed throughout the cache hierarchy. This data takes up a larger proportion of space in L1 cache than L2 and L3, which is why we would see a greater impact on the performance at lower levels of the cache. When sending data synchronously, it makes sense that the application would want that information in cache so that the CPU may service communication events faster. However, when this data is sent asynchronously, we do not know when the application will require the written data (if it does at all). In the asynchronous case, loading the data into cache can be benign (as seen with CNS) or create significant bottlenecks (as seen in the other proxy-apps).

Table 2.6: PMC Correlations Across All Workloads

	Corr. Metric	Stalled Cyc.	L1 Miss	L2 Miss	L3 Miss
No RDMA	Time	-0.04	0.941	0.946	0.930
	Stalled Cycles	N/A	0.086	0.030	0.068
RDMA	Time	0.912	0.959	0.978	0.925
	Stalled Cycles	N/A	0.870	0.973	0.997

Our reasoning above presumes some relationships between the slowdown seen with RDMA traffic, activity in cache and CPU stalled cycles. To determine the strength of these relationships, we performed a correlation analysis between runtime, stalled cycles and cache related counters. We used Pearson's R for correlation,which measures linear correlation between two variables. The analysis results (Table 2.6) show that without additional RDMA traffic, the application runtimes are very strongly correlated to L1/L2/L3 misses but are not correlated with stalled cycles. Stalled cycles for the non-RDMA case do not meet required levels of significance to assert that a relationship exists. For the RDMA traffic case in Table 2.6 a very strong correlation exists between runtime and stalled cycles with a 95% certainty. We see that there is a very strong correlation between the stalled cycle count and the cache misses, particularly L3 misses, showing that the stalled cycle increase is almost certainly due to misses throughout the cache hierarchy and requests to main memory. This correlation coupled with the large rise in stalled cycles that occurs when introducing RDMA traffic leads us to conclude that the increase in runtime observed is due to time waiting for the memory subsystem.

Though cache pollution from RDMA is correlated with an increased number of stalled cycles, it is not necessarily the only factor contributing to NiMC. Other contributing factors can include: the policy and scheduling of the memory controller(s), such as open-page row-buffer management, the degree of concurrent operations the memory controller(s) can handle, the number of memory channels, and how these memory channels are written to, for example, ganged or unganged. To fully model the impact of NiMC and offer mitigating solutions, these factors must also be considered.

2.2.6 Detection and prediction of NiMC

In the previous section we found unexpected correlations between NiMC, the cache and stalled cycles. Our success motivated us to take a closer look at a wide range of performance counters. In this section, we apply more sophisticated techniques for prediction and analysis, to Sandy Bridge-X2-onload. These techniques further characterize NiMC and answer the questions set forth at the beginning of this work. Specifically, the objectives are:

1. To detect the presence of NiMC on onload NICs.

2. To predict the volume of RDMA traffic.

3. To predict the impact of NiMC on application CPU time.

In the previous section we used a limited number of runs to perform correlation analysis between a small set of PMCs. In this section we've expanded the PMCs from 4 to 17 (seen in Table 2.3). We've also expanded the number of runs to 6,400 for each feature set. We've reduced the number of workloads in this section to a subset of those analyzed in the previous section, specifically: STREAM-DRAM, STREAM-cache, CNS, LAMMPS, and HPCCG. Each of these workloads were chosen to be representative of particular characteristics. STREAM-DRAM and its variant STREAM-cache were chosen, because they are synthetic benchmarks that aggressively push the memory system and easily reasoned about. CNS was selected as a control, since it experiences almost no impact from NiMC. We chose LAMMPS since it was the most impacted and a full application. HPCCG falls between the synthetic benchmarks and full applications as a proxy app having high memory utilization, but not as much as STREAM.

Table 2.7: OOB scores for binary classification.

App/Benchmark	Set 1 Score	Set 2 Score	Set 3 Score
STREAM-DRAM	1.000	1.000	0.995
STREAM-cache	1.000	1.000	0.990
HPCCG	0.998	0.999	0.999
CNS	0.741	0.747	0.742
LAMMPS	1.000	1.000	1.000

Predicting the presence of NiMC

In our first use of random forests, we evaluate whether the machine learning correctly classifies the RDMA traffic for a binary classification (RDMA/No RDMA). Specifically, we remove the Infiniband bandwidth feature from each feature set, then create a binary classification, where any bandwidth greater than zero is labeled as *RDMA* and otherwise *No RDMA*. This binary vector is used for our target values.

As displayed in Table 2.7, the OOB score for each feature set and application was excellent with the exception of CNS. In random forest classification the OOB score, is the ratio of correct predictions over the total number of trials. This makes a value of 0.74 not particularly meaningful, since in the case of a coin flip a random guess would obtain an OOB score of 0.5. Therefore, while there are some features of value in the CNS results, they are not particularly reliable in their predictive power.

After determining the OOB score, we examine the importance of each feature in our predictions. Random forests have a measure of feature importance that is calculated differently depending on whether the forest is targeting classification or regression. In the case of classification this is commonly calculated by adding up the decrease to *Gini impurity criterion*[14] for each individual variable over all trees in the forest. Gini impurity measures how frequently a randomly selected and randomly classified sample would be incorrectly classified given the distribution of labels in the set. In the case of regression, minimizing the mean squared error is commonly used as the

impurity measure to calculate importance. It is important to note that while we can determine which features are *most* important, if two features A and B overlap in the information they provide, once a tree splits on feature A, feature B becomes less important to future splits, since the tree has already incorporated the information provided by A. So, a low ranking feature importance does not necessarily mean a feature is *not* important. In this section we use the importance measure to draw our attention to features that provide predictive power. Because of the volume of feature importances calculated, we have placed them in the appendices.

STREAM-DRAM Examining the results of feature set 1, L1 instruction cache misses was a particularly important feature. If we look at the histograms of runs with and without NiMC, we see stark differences in the distribution and values. Without NiMC we see in Figure 2.15(a) that the vast majority of runs (95%) have between 2.5×10^7 and 2.6×10^7 misses. With added RDMA writes, Figure 2.15(a) shows that the number of misses goes up substantially (in some cases nearly doubling) from 3.0×10^7 to 4.3×10^7 misses. Interestingly, though the number of L1 data cache misses is several orders of magnitude greater, the feature provides little information, since it does not change significantly with the addition of NiMC. This is largely due to the fact that STREAM is designed to marginalize the cache by using arrays several times bigger than could fit into last level cache. Because the overall data cache miss rate is so high to begin with, the added misses due to NiMC contain relatively little information when compared to instruction cache misses. Lastly, from the third feature set, we see that L3 instruction cache access was the most important feature. If we compare median of runs with and without NiMC, we see that in the case of NiMC there is a 12% increase to the number of L3 ICA.

STREAM-CACHE The behavior of STREAM-cache is identical to STREAM-DRAM with the exception that we are ensuring that the data matrices fit inside the L3

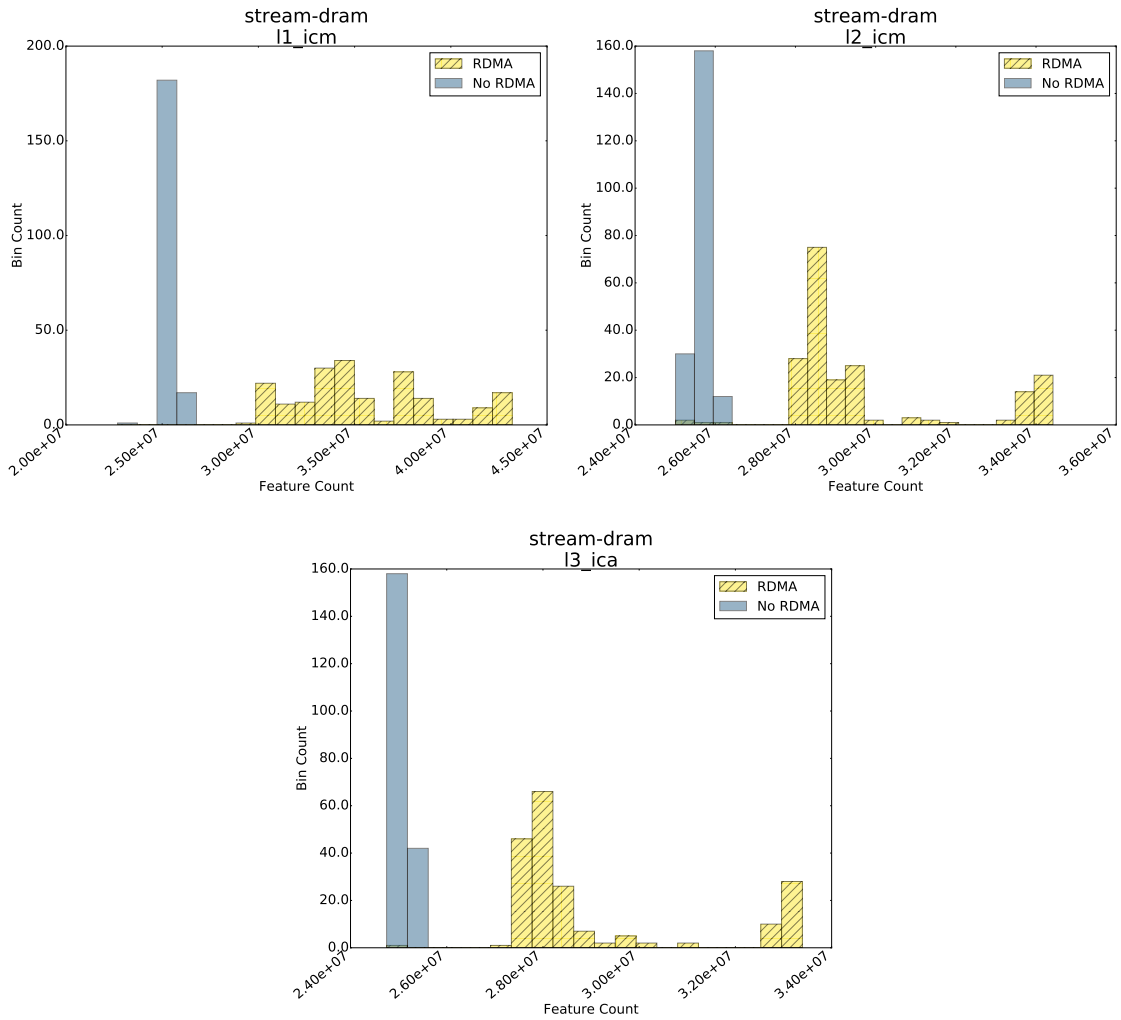

Figure 2.15: Histograms of important features for for binary classification of NiMC (STREAM-DRAM)

cache, whereas STREAM-DRAM uses matrices of at least 4 times the size of last level cache. As a consequence of the smaller matrix sizes, L1 data cache miss becomes an important feature in the machine learning. While we can clearly distinguish the two distributions in Figure 2.16(a), in reality the difference between the two distributions is less than 1% of the L1 data cache miss. Even though STREAM-cache has terrible cache utilization and a large number of L1 data cache miss, this shows that as we

increase the application's cache efficiency, we begin to see a shift in importance from instruction cache to data cache.

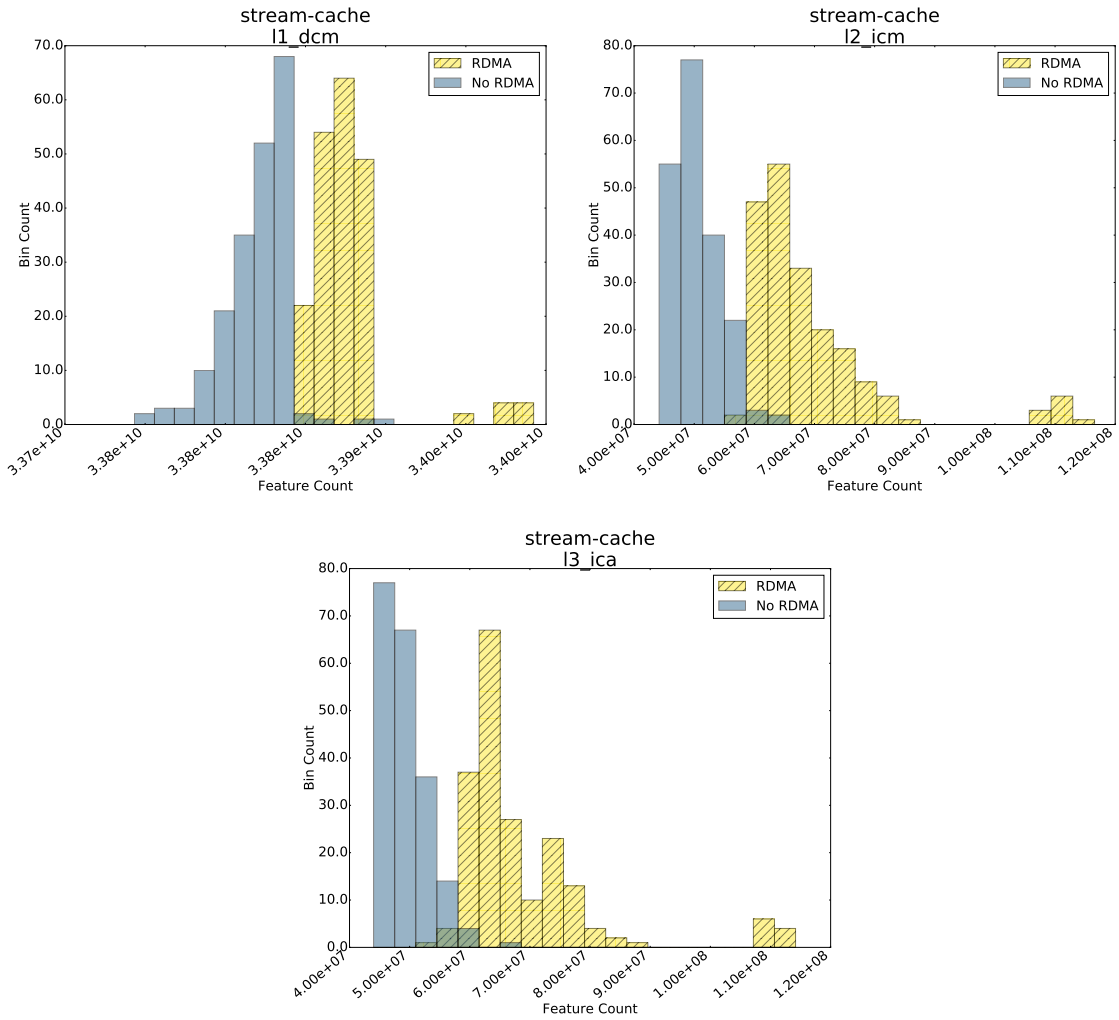

Figure 2.16: Histograms of important features for for binary classification of NiMC (STREAM-cache)

HPCCG and LAMMPS In contrast with the STREAM runs, data cache misses and total cache accesses are important features for HPCCG and LAMMPS. This makes sense given the rate of data cache access drops dramatically in HPCCG and

LAMMPS compared to STREAM. Looking closer at the results of HPCCG and LAMMPS in Figures 2.17(a)-2.18(c) we see some surprises. Specifically, for the vast majority of processes, data cache misses and total cache accesses decrease as we add a competing RDMA traffic. It is important to remember that these measurements are the sum over a run and not a rate of misses or accesses over time. In other words, we see a decrease to total cache misses and accesses as the CPU time increases for these two applications.

This is an example of how machine learning can point us in directions that we might not explore otherwise Once we know where to look, human intelligence and expert knowledge can provide a deeper understanding of the environment. The reason for observed decrease in L1 misses can be explained by a slowdown which causes a decrease to the rate that operations are issued. In other words, the L1 cache access per unit time is decreasing, which allows for better hit to miss ratios of the L1 cache. Better hit to miss ratios at the L1 create less pressure on L2 and L3, resulting in both fewer accesses and fewer misses at higher levels in the cache. Going back, evidence suggests that the slowdown that reduced pressure on the L1 for the majority of the processes is the result of waiting on a single laggard process. Looking at Figures 2.17(a)-2.18(c), we commonly see a small number of outliers in both misses and CPU time. By examining the individual runs, we confirmed these outliers are often just a single process of a run, which creates an intra-run imbalance. At points of synchronization and communication faster processes are forced to wait for the completion of the slower process. This waiting period provides the faster processes an opportunity to satisfy any outstanding requests and clear the pipeline. When the slowest process reaches the synchronization point the majority of processes are able to take advantage of the cleared pipelines and achieve better cache efficiency until they fill up.

The next logical question is, why don't we see this behavior in STREAM? We cannot

answer this with absolute certainty, but one idea is that this is because STREAM is designed to miss cache. Therefore, any benefit of clearing the processor's pipeline at synchronization point is negated as the empty pipeline rapidly fills up waiting on outstanding cache misses. We can see that when accounting for CPU time, that STREAM-DRAM/cache see an 8X increase to L1 data cache miss per unit time, compared to HPCCG.

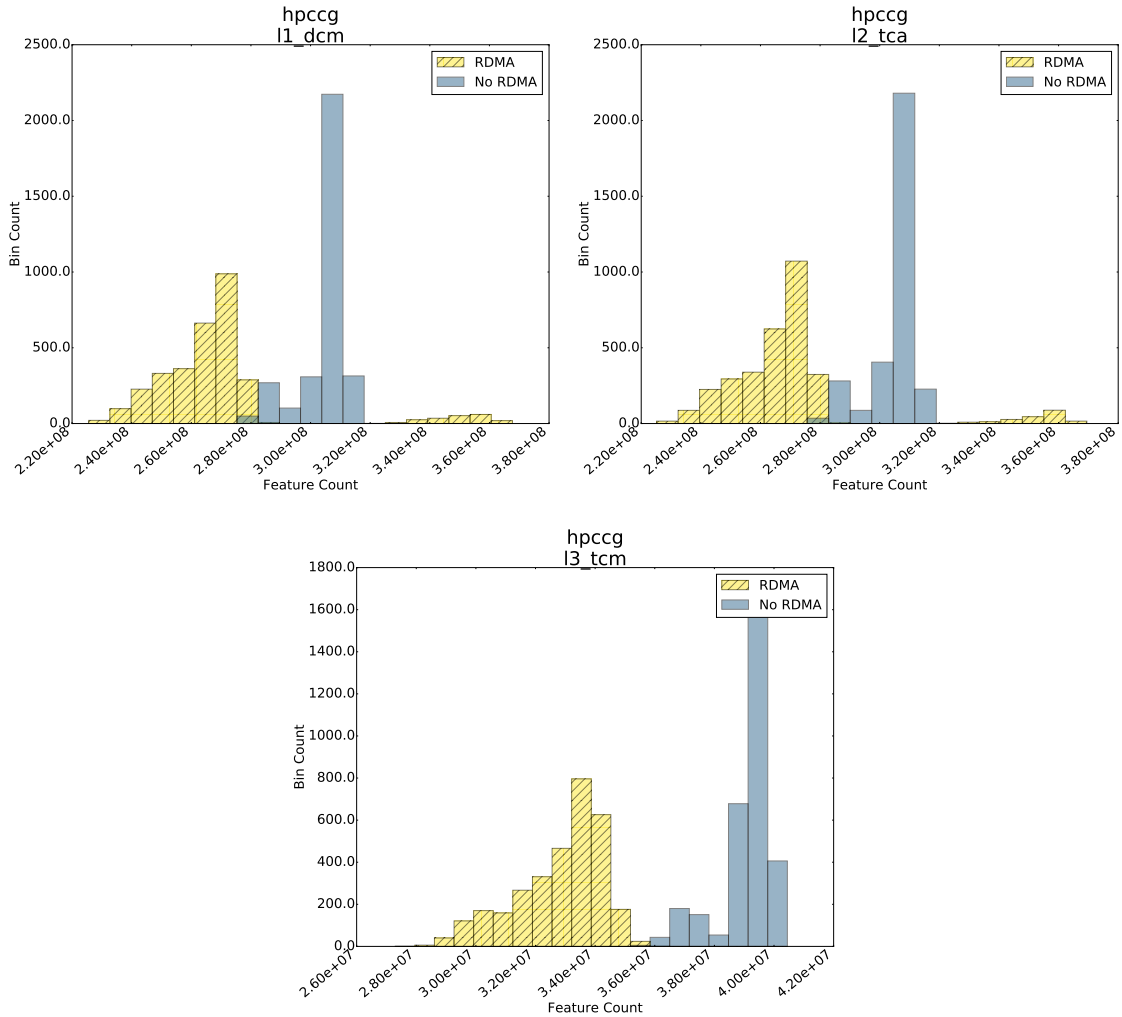

Figure 2.17: Histograms of important features for for binary classification of NiMC (HPCCG)

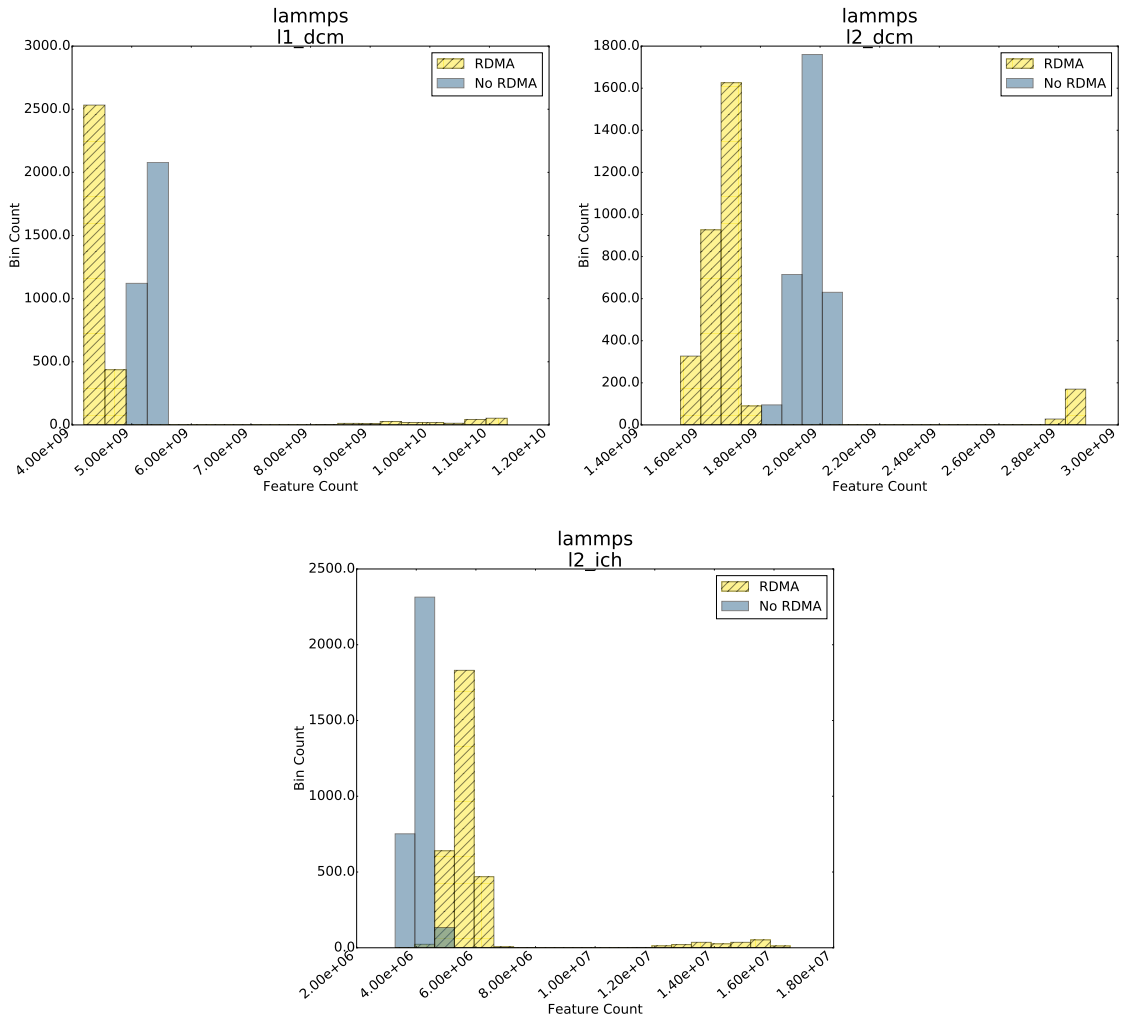

Figure 2.18: Histograms of important features for for binary classification of NiMC (LAMMPS)

Discussion For each of the applications other than CNS the classifier had an out-of-bag score of 0.995 or higher, meaning that if NiMC is present in system with Onload NICs, we can accurately predict presence of NiMC using any of the three feature sets evaluated 99.5 out of 100 times.

Additionally we saw a breadth of features were used to predict the presence of NiMC. This exemplifies how machine learning and expert knowledge may be leveraged in

future systems, since the best set of features appears to vary, dependent on the behavior of the application.

Predicting volume of RDMA traffic

Next, we wanted to determine whether we could predict the amount of RDMA traffic with more detail than just a binary classification. To do so, we used random forest regression rather than random forest classifiers on the same data as before calculating both the coefficient of determination (R^2) and OOB scores. While R^2 suggested some success, this was misleading and contradicted by low OOB scores. The reason for this discrepancy is that, given a predicted value of $\hat{y}$ and a mean value of $\bar{y}$, R^2 is calculated as:

$$R^2 = (1 - u/v)$$

$$\text{where } u = \sum_{i=1}^{n} (y_i - \hat{y})^2 \text{ and } v = \sum_{i=1}^{n} (y_i - \bar{y})^2$$

Because the distribution of injected bandwidth is often bi-modal, this makes the use of average in v particularly poor, creating an inflated score that really is not meaningful. The OOB score does not suffer from this and for the purposes of our work is more informative.

Unfortunately, the results indicated that none of the feature sets were informative enough to precisely predict the volume of RDMA traffic in the presence of NiMC. Part of the reason may be that the throughput of the RDMA traffic injected onto the target node is also dependent on contention in the network and may not be fully represented by the features recorded. Because SandyBridge-X2-onload is a shared system, the distribution of throughput varied across the runs depending on the competing workloads. While our preset counters did not provide the necessary information, this does not rule out the other performance counters that might be

explored in future work. These might include PAPI native events, which are more numerous than the PAPI preset events, but are accessed via the low-level PAPI interface.

Predicting runtime impact of NiMC

In this section we wanted to predict the CPU time of each process for the set of applications and benchmarks with and without NiMC. Initially, we used the entire feature set of Table 2.3, so that our target vector is the CPU time recorded by OpenSpeedShop, while the training input samples are comprised of the features in Table 2.3. Examining the results, we found that each feature set was able to provide quite accurate predictions of CPU time. The OOB prediction across the different feature sets and applications ranged from 0.969 to 0.997, with the worst (although still good) score being for CNS using feature set 3. This is likely due to the fact that CNS has small working set of memory and a relatively small number of L3 misses compared than other applications. Furthermore, the results suggested that each feature set was capable of accurate predictions of the CPU time.

While the OOB scores were quite high, one of the things we noticed is that the IB bandwidth feature dominated in importance, providing little information about the significance of the other hardware counters. Furthermore, we only have knowledge of the volume of RDMA traffic injected because we introduced it in a controlled experiment. In a more realistic approach this information would need to be shared with the remote (application) node incurring a significant delay due to latency. In response, we decided to remove the IB bandwidth feature from the training set and run the experiments again. What we found is that even when the IB bandwidth feature was removed, we maintained very good predictions. The largest decreases to the OOB score were around -0.03 (e.g. 0.997 to 0.967). This suggests that the IB bandwidth feature while valuable, provides information that can be gained through

Table 2.8: OOB scores for regression on CPU time.

App/Benchmark	Set 1 Score	Set 2 Score	Set 3 Score
STREAM-DRAM	0.984	0.997	0.991
STREAM-cache	0.992	0.994	0.984
HPCCG	0.966	0..5	0.966
CNS	0.981	0.978	0.966
LAMMPS	0.990	0.995	0.967

the other performance hardware counters (features) in order to accurately predict CPU time.

In the following paragraphs we use the rankings of feature importance as a tool to further analyze the results and try to understand why a particular performance counter was ranked more importantly than others.

STREAM-DRAM While many of the features that were important in predicting the presence of NiMC are important in predicting the CPU time, there are some differences. This is expected given that NiMC is only one of many components determining the overall runtime of an application. From the second feature set, the derived fraction of L2 data cache miss over L2 total cache access is an important feature in predicting CPU time that was valued much less in predicting NiMC. In general, it's not surprising that a derived metric of L2 efficiency is valuable in determining the performance of STREAM. The histogram for this feature (Figure 2.19(a)) shows that in the presence of NiMC, STREAM-DRAM distribution of L2 cache efficiency changes from a distribution that looks fairly normal to a distribution that is much harder to characterize. Specifically the feature shows some worst case runs with NiMC achieving a value of just over 0.60 compared to 0.56 without RDMA traffic.

STREAM-CACHE In general, we found that a smaller number of features were given greater importance, though the relative rankings of importance did not change

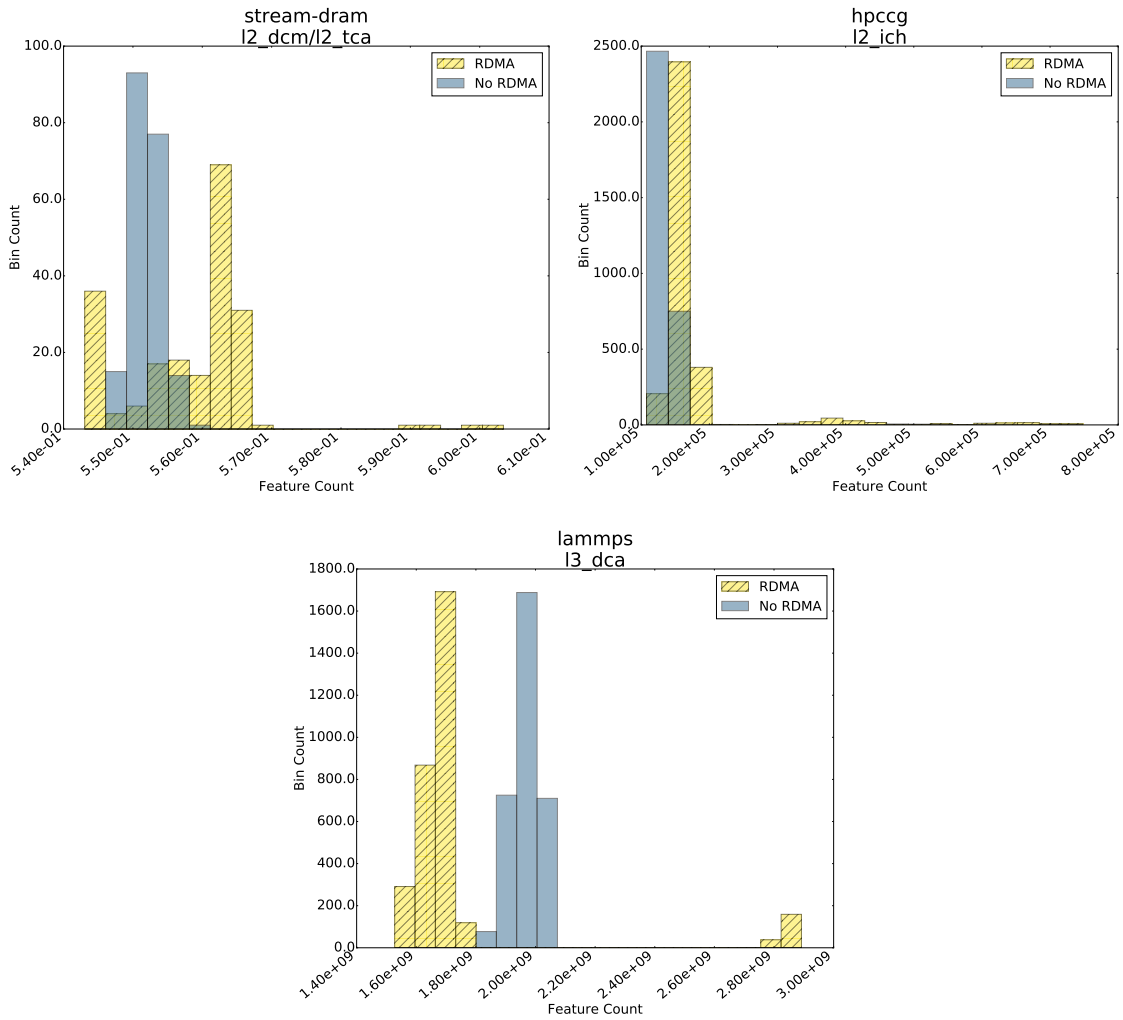

Figure 2.19: Histograms of important features (L2_DCM/L2_TCA, L2_ICH, and L3_DCA for STREAM-DRAM, HPCCG and LAMMPS, respectively), when predicting CPU time.

for STREAM-cache. Specifically, when predicting CPU time we found that L1 data cache miss, L2 instruction cache hit, and L3 instruction cache access achieved much greater importance (more than double in the case of L3 instruction cache access) than when predicting NiMC.

CNS CNS is different from the other applications because the added RDMA traffic has negligible impact on it. In fact IB bandwidth is rated as the least important feature in determining CPU time. This matches our intuition and is as expected since we are using CNS primarily as a control-case in our experiments. For these reasons, the rankings of importance with respect to predicting CPU time of CNS provide little insight about NiMC.

HPCCG When comparing the feature importance of HPCCG to predict NiMC versus predicting CPU time, there is a change in the rankings of feature set 2. We see that L2 instruction cache hit becomes a very important feature in predicting CPU time, whereas it was the 5th most important feature when detecting NiMC. Looking at the Histogram in Figure 2.19(b), we see the familiar distribution where a small number of processes are fall well outside the expected range. In the worst case, some processes saw up to a 7X increase in the number of instructions that hit L2. Whether this is due to additional instructions issued by the drivers for the onload NIC, or for some other reason, we can only speculate.

LAMMPS When reviewing the results from LAMMPS, the most important features from Sets 1, 2 and 3 are L1 data cache miss, L2 instruction cache hit, and L3 data cache access, respectively. Within feature sets 1 and 2, the importance becomes heavily weighted towards a single feature. In feature set 3, the importance of L3 data cache access decreases while L3 total cache access increases in importance. This makes sense given the fact that TCA is the sum of ICA and DCA, however this shows that DCA counters provide more information. Of the applications we ran, with the exception of CNS, LAMMPS has the best cache utilization, such that if there is an increase to L2 misses that becomes an L3 access, it is more noticeable than it would be in STREAM or HPCCG.

Discussion Several conclusions can be drawn from these experiments. First, if we exclude IB bandwidth, no single feature is universally important. For example, hardware events like L1 data cache miss were very important in determining the CPU time of STREAM-cache, HPCCG and LAMMPS, but less so when used to predict the CPU time of STREAM-DRAM and CNS. There tends to be some overlap in importance among similar applications and machine learning facilitates an understanding that compliments expert knowledge. We did see that IB bandwidth was a valuable feature, however extracting it from the system is not straight-forward. It is possible that it could be gleaned from PCI-e counters however these are not readily available. Another alternative is that the NICS or hardware drivers could report this information, but this requires that they (1) distinguish RDMA traffic apart from two-sided traffic and (2) develop a mechanism for reporting that information. A third possibility is that whatever service is pushing data into the remote system report it to the relevant applications, however this solution incurs significant latency in reporting.

Second, for some applications a majority of processes saw decreases in cache misses as a result of running slower. Because of the black-box design of most hardware, we can only speculate on the reasons behind this. We can generalize that this behavior is isolated to applications that have better cache utilization than the benchmarks which are designed to induce cache misses. Furthermore, we saw that the processes that see a reduction in cache misses spend a significant amount of time waiting on processes delayed by NiMC.

Third, it is apparent that some processes are disproportionately impacted by NiMC. Due to the synchronization points and barriers of today's BSP style programs these stragglers delay all other processes. In other words, the fastest processes are limited by the slowest. This is further magnified as the number of processes increases. It would seem that the problem of NiMC is largely an artifact of BSP style programs and

that we might expect it to become less prevalent in future asynchronous applications. However, this is not entirely true. A closer examination of our results show that even the fastest CPU times of processes for the applications are slowed down significantly. Part of this is a result of shared hardware resources between the cores, particularly last level cache. While it is true that NiMC will not impact an asynchronous program as much as current BSP programs, we expect it to remain a issue in future systems.

From a high level viewpoint, our experience with machine learning shows that it is an effective technique, that once trained, will allow us to respond to NiMC in a timely manner. Additionally, this work shows the value of machine learning in HPC, as we've successfully use these techniques to develop a deeper understanding of the relationship between performance counters, applications and NiMC. In several instances, the features of highest importance surprised us, which shows the value of a principled approach such as random forests, since as system experts we might have allowed our intuition to lead us to selecting features that were not as rich in information.

2.2.7 Large scale evaluation

From the previous studies, we gained a better understanding of the underlying causes of application interference in the context of a single, isolated node targeted with an unlikely high volume of RDMA traffic. Our final study examined NiMC for applications at scale with realistic RDMA traffic volumes.

As with our single node experiments, we use the Sandy Bridge-X2-onload cluster executing a series of weak scaling LAMMPS experiments. We selected LAMMPS for the large scale runs for multiple reasons. First, of all our workloads, LAMMPS is the only real application: it is not a proxy app nor a contrived benchmark. Second, LAMMPS scales very well for the size of system under study. Finally and most

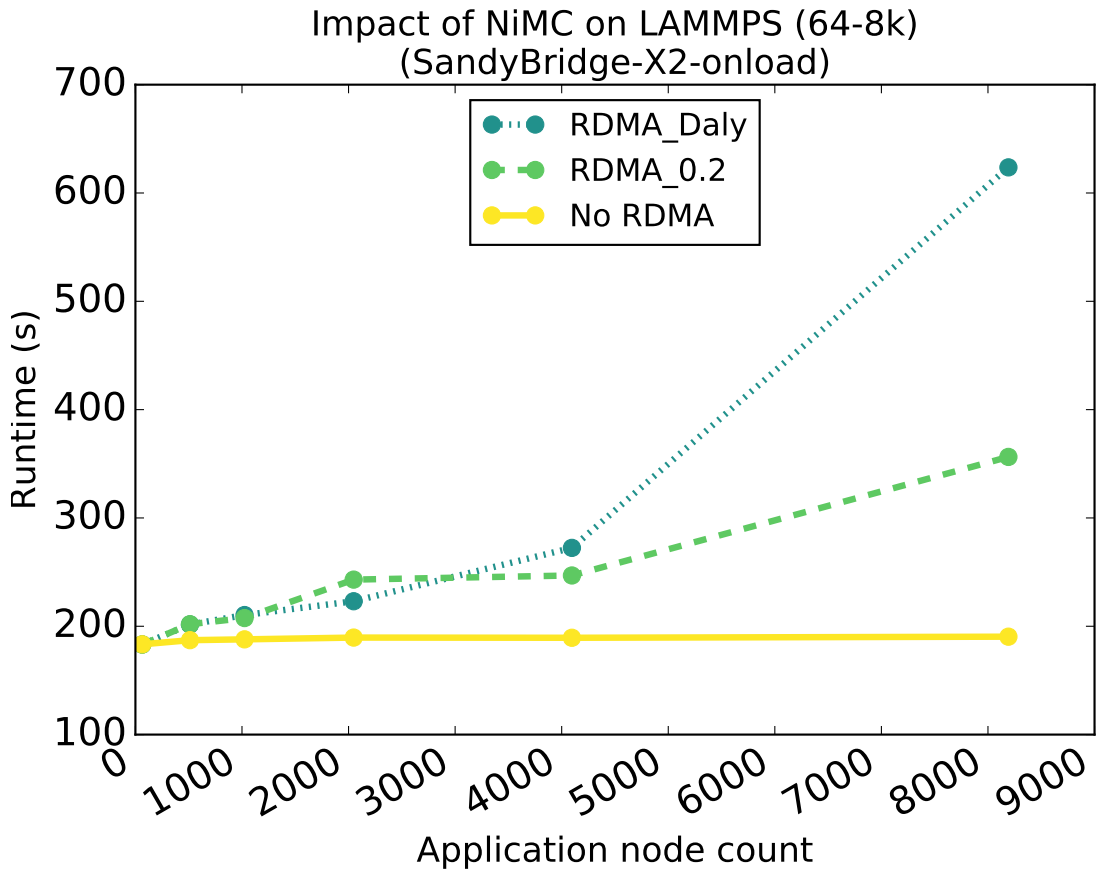

Figure 2.20: Impact of NiMC on LAMMPS for an onload system.

importantly, LAMMPS is widely regarded as an application that is resistant to external interference [85]. Therefore, LAMMPS represents a good challenge when trying to realize performance degradation due to external perturbations like RDMA traffic.

As in our previous experiments, in addition to the target nodes running our application, we reserved an additional set of origin nodes that push our RDMA traffic to the target nodes. However, unlike in our previous experiments, we limit concurrent writes to a small subset of the total nodes. We also limit the duration of each write operation. We use a hypothetical uncoordinated, in-memory or disk-less checkpointing protocol[7]

[7]Contemporary approaches for in-memory checkpointing use a coordinated protocol in

Table 2.9: Number of concurrent RDMA writes

Application node (rank) count	Writes/s (Daly) QDR-onload	Writes/s (Daly) FDR-offload	Writes/s (0.2%)
64	0	0	0
512	1	1	1
1024	2	2	2
2048	5	6	4
4096	15	17	8
8192	42	47	16

to shape our RDMA traffic pattern. Since there is no known optimal checkpoint interval for uncoordinated checkpointing, we use Daly's estimate [22] to derive optimal coordinated checkpoint intervals. We use Daly's estimate to compute the average number of processes simultaneously taking a checkpoint using a five year mean time to interrupt, and use this number as the number of concurrent writers, shown in Table 2.9. We compute RDMA write duration by optimistically assuming that all checkpoints take 46 thousand message iterations (equivalent to ≈1s for 4X-QDR IB and ≈0.5s for 4X-FDR IB). Each data point in Fig 2.20 represents the minimum runtime of 5 runs. We chose the minimum because (1) the minimum is the hardest metric to overcome, when showing the existence of NiMC at scale. (2) it shows the impact of NiMC rather than the impact of contention on the network or I/O subsystem from other jobs that are outside of our control.

The results in Figures 2.20 show that as we scale up the number of application processes the impact of NiMC becomes significant. Despite the fact we decreased write duration to a single second, scaling up the number of application processes greatly amplified the magnitude of interference. This is similar to phenomena seen in the research of OS noise, where scale amplifies the magnitude of the overall perturbance [59, 101]. Even with a constant 0.2% of total nodes as simultaneous writers, time-to-solution nearly doubles at scale due to NiMC. While we've increased the pressure on the network by introducing additional RDMA writes, we will show in

which all processes take a checkpoint simultaneously. However, for next generation systems there is a concern that coordination at massive scale can become prohibitively costly.

§2.2.8 that this additional traffic is not responsible for the increase to runtimes.

2.2.8 Solutions for NiMC

We evaluated several approaches for reducing the impact of NiMC, specifically: (1) offload hardware, (2) core reservation, and (3) software based network throttling. All of these techniques have been applied in other areas of research [29, 39, 83, 2] but this work is the first to evaluate their effectiveness in mitigating NiMC. For the results in Figs 2.21(a) and 2.21(c) we present the best (min) baseline LAMMPS runtimes and the median runtimes for NiMC based on Daly's volume. For the platforms evaluated, the difference between the minimum and the median for Daly's volume runs was negligible.

Offload NICs as a solution

In § 2.2.4, it was shown that NiMC did not negatively impact the performance of the most recent offload systems for the benchmark STREAM. In this section we show that offload NIC's continue to provide a solution to NiMC at scale for real applications. In Fig 2.21(a) we ran LAMMPS on the Sandy Bridge-X2-FDR-offload system, weak scaling up to 8192 processes. Comparing the results of *No RDMA* and *Daly*, it is evident that NiMC does not have any observable impact on offload system performance at scales of up to 8192 processes. From these results, we conclude that offload NICs provide a solution for modern systems that would otherwise experience NiMC. The main drawback to offload NICs is their greater monetary cost compared to their onload equivalents. However, this is not a guarantee that future systems will be unaffected by NiMC as the disparity between network and memory bandwidth shrinks. Even though none of the most recent offload systems were impacted by NiMC – on slightly older systems (Westmere and Lisbon) we observed a 16-25% decrease

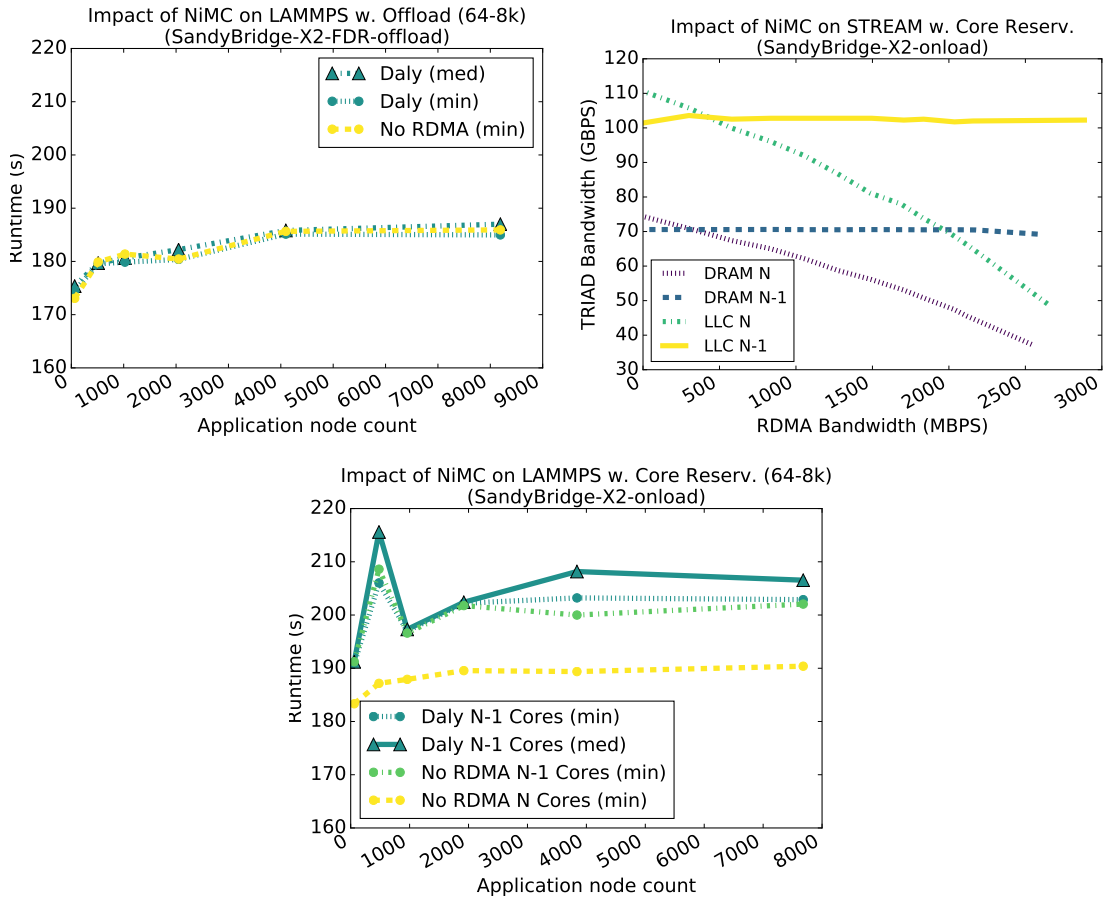

Figure 2.21: Fig 2.21(a) highlights the performance of offload cards at scale for the application LAMMPS. Fig 2.21(b) shows the relationship between RDMA traffic and available memory bandwidth in DRAM and LLC using N and N-1 cores. Every bit per second of RDMA traffic reduces DRAM and LLC bandwidth by 14 and 22 bits per second, respectively. Fig 2.21(c) demonstrates a core-reservation solution at scale for the application LAMMPS.

to STREAM Triad performance due to NiMC. We believe the future viability of an offload solution is dependent on how fully CPUs utilize memory bandwidth and by future network bandwidth increases.

Core reservation

Dedicating a core to service communication is another possible solution to mitigate NiMC. This reduces the memory throughput of the CPUs, and sets aside separate cache resources for handling network data. The downside to core reservation is that in the absence of RDMA communication the reserved core is wasted. To test the effectiveness of core reservation, we repeated the STREAM and LAMMPS tests of §2.2.4 and §2.2.7 while reserving one core per node to process communication, binding the QIB driver to the reserved core. Additionally, we evaluated core reservation for a modified version of STREAM which uses array sizes designed to fit entirely in last level cache (LLC). The modified STREAM allows us to evaluate LLC performance, with respect to NiMC. Both of these tests can be seen in Figs 2.21(b) and 2.21(c). Interestingly, Fig. 2.21(b) presents an intersection near 400-500 MBps on the x-axis, where a core reservation strategy begins to provide a performance benefit. As the RDMA bandwidth increases towards 3000 MBps we see performance gains by setting aside a core. This suggests that a dynamic strategy for reserving a core to service the network may be an attractive approach for future systems. Making this decision on a live system could be determined by using a random forest to predict the impact of NiMC (as seen in §2.2.6) and determine if core reservation was necessary.

In Fig 2.21(c), we use identical input files as the previous section, however we only utilize N-1 cores per node for the application. These results show that core reservation continues to be an effective strategy to prevent NiMC at scale, independent of the volume of RDMA traffic. These results clearly demonstrate that contention on the network is not a factor in the increase to runtime seen in § 2.2.7, Fig 2.20. This can be observed in Fig 2.21(c), where the cases with and without RDMA traffic have only a 0.1% difference in runtime, despite contention on the fabric that would be present in the RDMA results. This allows for a quantification of the induced network contention due to the RDMA streams, which is much less than the observed

contention due to NiMC. Of minor note, there is a 10% increase to the runtime of the runs that utilize 480 processes compared to the runs of 60 and 960 processes. After discussion with LAMMPS developers it was determined to be the result of increased communication due to a less efficient domain decomposition for 15 cores per node. Overall, the results suggest that core reservation incurs a runtime increase of 6.4% compared to 16-core to 15-core runs of LAMMPS without RDMA traffic. While a 6.4% increase to runtime is costly, it costs significantly less than the 330% increase without core reservation (Fig. 2.20).

Software-based solutions

There are several methods for throttling or shaping the traffic sent over the network. One such method is to artificially throttle the throughput of the network so that the same total volume of traffic is sent over a longer time period. The practicality of throttling is partially dependent on the significance of the network data to the application. It is important to remember that traffic throttling increases the time to deliver the data, but makes more memory bandwidth available to the CPU. If the network data is part of the application's critical path and the application is executing faster (due to the increase in available memory bandwidth), throttling may leave the application stalled. In Fig. 2.21(b) we plot the impact that varying network speeds have on STREAM DRAM and LLC performance. Performing a least square linear regression shows that for every bit per second (bps) of RDMA bandwidth we add we slow the DRAM performance by 14 bps. When considering LLC, this tradeoff becomes even more expensive as 1 bps of RDMA bandwidth reduces LLC bandwidth by 22 bps. On modern HPC systems where memory performance is a highly valued commodity, the large disparity of this tradeoff makes network throttling less appealing as a solution compared to a hardware offloading. Additionally, for the system evaluated, software-based throttling is only effective if you can reduce the

amount of network traffic to under 500 MBps. If an RDMA service requires more than 500 MBps network bandwidth, core reservation becomes a better solution.

Summary of solutions Our results show that a hardware offload solution is ideal for modern systems, Though (as Westmere and Lisbon demonstrate) future effectiveness is dependent on trends in CPU memory bandwidth utilization and network speeds. Secondly, many systems utilize lower cost onload NICs. For these systems there are two approaches to mitigate NiMC. The first approach of throttling the network is limited in its effectiveness. Specifically there is a crossover point where the amount of network traffic becomes large enough that core reservation becomes a better solution. This crossover point will vary system to system but was 500 MBps for our evaluated platforms. Lastly, core reservation allows for unthrottled RDMA bandwidth but at a base-level increase to runtime of 6.4%. Though we provide general guidelines, determining the best solution for each system requires a knowledge of the application, services and underlying hardware. As we've demonstrated, machine learning allows us to predict the impact of NiMC on application runtime and can facilitate a dynamic solution space.

2.2.9 Outcomes of NiMC study

In this work, we introduced the concept of NiMC and demonstrated its impact for a variety of current HPC system architectures. We showed that NiMC is a concern for both onloaded and offloaded networking hardware, with the onloaded hardware observing the largest performance impact. For all but one of our applications, we observed significant performance impact on modern onload systems due to NiMC, and we ruled out that the observed NiMC impact was not significantly attributable to CPU contention or network contention. This work examined how we might detect NiMC and predict its impact on workloads by using random forests. In this research

we found that no single PMC was universally important to all workloads, validating the use of sophisticated methods of machine learning. We demonstrated how random forests could effectively (and without expert bias) determine which PMCs were important to predicting NiMC's impact on a particular application. Using real applications at large scale, we showed that NiMC can lead to significant performance degradation even in applications like LAMMPS that have previously demonstrated performance robustness in the presence of other types of system noise. While this work only examined InfiniBand based networks, such networks are relevant for HPC as evidenced by the recent $325 million CORAL procurement [128], a 150 petaFLOP IB-based system.

Lastly, we evaluated three strategies to mitigate NiMC, namely offload NICs, core reservation, and network throttling. Our results suggest that Offload NICs appear to provide the best, albeit most expensive, solution to NiMC on modern systems, provided there is sufficient headroom between theoretical and observed CPU memory bandwidth. In the event a cluster utilizes an onload NIC, setting aside a CPU core to service the network is a viable solution for onload systems, but incurs a runtime penalty proportionate to the power of the core (6.4% in our study). The current disparity in the way memory bandwidth is provisioned between RDMA and the CPU makes the third solution (network throttling) attractive only if the required RDMA bandwidth is below specified thresholds (Fig 2.21(b)). We've demonstrated that machine learning (specifically random forests) enables the prediction and detection of NiMC necessary of a dynamic solution space.

2.3 Chapter Conclusions

In this chapter, we studied one-sided communication from the application layer (MPI) through the transport layer (IB verbs). We evaluated the performance of one-sided

communication from the application level (using both OpenMPI and MVAPICH). To conduct this study we developed the RMA-MT benchmark suite which was released to the public. These benchmarks will continue to assist future studies that focus on improving the performance of RMA communication in MPI. Later in the chapter, we showed that RDMA communication has the potential to greatly disturb application performance via Network-induced Memory Contention. Furthermore, we found that NiMC impacts a wide range of architectures, resulting in increased runtime for applications. We demonstrated that the penalty of NiMC grows with scale of the system, showing potentials for 3X increases to the runtime. Our work showed that NiMC can be detected from easily accessible performance counters using machine learning and we demonstrated three potential solutions to eliminate NiMC, namely hardware offloading, network throttling and core reservation. In conclusion, we have provided an in-depth study of one-sided communication in HPC systems that highlights both the benefits and potential pitfalls. This knowledge enables HPC systems and application designers to get the most out of next-generation communication. In the next chapter, we will expand the scope of our work, moving beyond the node-centric view of this chapter to study performance and power of the network fabric.

Chapter 3

Balancing Performance and Power of HPC Interconnects

In Chapter 2, we characterized the performance of next-generation communication methods. Specifically, we evaluated the performance of one-sided communication at both the transport layer and application layer. This provided a node-centric view of communication that largely ignored network fabric (the routers, switches, NICs and links that connect the nodes). In this chapter we transition to a more fabric-centric perspective, focusing on how network topology and bandwidth affects system power and performance. With power being a primary concern in the global race to Exascale, the network fabric (which may consume more than 10% of the total power budget) provides attractive opportunities for savings. Simulation enables us to experiment with network design at scales of hundreds of thousands of ports, which would otherwise be infeasible. We leverage simulation to provide estimates of costs and savings at scales previously unmatched. Throughout the chapter, as we evaluate the power consumption of a topology we also evaluate its impact on application performance. In addition, this work goes beyond just reporting the run time of our workloads. We want to better understand why network topology A outperforms

network topology B. So, we provide new techniques for visualizing and analyzing network utilization at port level detail. The expectation is that, by the end of this chapter the reader will have a solid understanding of the power/performance tradeoffs associated with HPC network fabric.

3.1 Stalled Active and Idle: Characterizing Power and Performance of Large-scale Networks

In this section, we perform a comprehensive, simulation-based study of large dragonfly networks and application motifs that represent workloads and communication patterns important to the HPC community. A primary motivating question was whether and how varying dragonfly network bandwidth and link configuration can improve overall network efficiency. We analyze the relationship among a network's stalled, active and idle port cycles. And we combine these metrics into a useful visualization approach that provides fine-grained insights about network utilization. Furthermore, we model the power consumption of these networks using both empirical measurements of network power as well as estimates from existing literature.

For this study, we use the Structural Simulation Toolkit (SST) [104] and a range of relevant workloads on a dragonfly topology of 110,592 nodes to examine network design tradeoffs amongst execution time, power, bandwidth, and the number of global links. Our contributions are:

1. An evaluation of how tapering global dragonfly links impacts 11 important workloads;

2. An evaluation of the performance impact of link-width-reduction on dragonfly networks at scales significantly larger than previously studied;

3. A scalable and information-rich approach for visualizing network utilization

performance.

4. Estimates on the potential for static and dynamic network power savings at Exascale, derived through empirical measurements and existing models in literature;

5. Enhancements to the SST simulator, which extend the statistics provided by the router and NIC components.

This work begins with an overview of background material and related work. We describe the specifics of our simulated hardware environment and workloads §3.1.3. In §3.1.4, we present the methodology and results of 88 simulations, examining the tradeoffs made between power, performance and the design of the network. Last, we review outcomes in §3.2.

3.1.1 Background

Network topologies

HPC network topologies can be divided into two main categories with hybrid approaches in between. These categories are primarily tree based, or mesh/torus based. Several factors lead to the selection of one topology over another. In short these factors are *latency/hop-count, bandwidth, path diversity, and monetary cost*. For each topology mentioned, we highlight these factors.

Torus and mesh networks differ from tree based networks in several key ways. Firstly, they provide an easy mapping to many of the scientific applications which predominately communicate with their nearest neighbors. Secondly, the number of links scales linearly with the number of nodes and the radix of the switch (the dimensionality of the mesh). The low radix of each switching element translates into a high worst-case latency/hop-count, while the minimal number of hops required on a mesh or torus

is 1. Similarly, on a mesh network, there is a greater worst case for congestion and contention for network resources. If the communication operations are not localized, a mesh or torus network must over-provision a greater proportion of links than its Clos or Fat-tree counterpart. The maximum number of hops required for a 2-D NxM mesh network is N+M, while a 2-D torus cuts this in half. Mesh networks have a high path diversity and an expensive monetary cost at scale.

Tree based network topologies combine higher radix switches to enable a lower worst-case hop count. In its most basic form, trees do not offer path diversity and the trunk represents a single point of failure and potential bottleneck for bandwidth. To mitigate the effects of these bottlenecks the capabilities of links grows as they are positioned closer to the trunk. Tree's that have this property are referred to as Fat Trees. Because a tree based network anticipates multiple nodes communicating across the trunk of the network, the trunk is provisioned to guarantee some fraction of the total bandwidth that the combined nodes of the system might use. For large systems, of many nodes, the worst-case latency on a tree based network is less than a mesh-based network, while the best case latency of a tree is worse than a mesh. The monetary cost of a tree based network is typically lower than that of a mesh or torus solution, since there is a decrease in the number of links required.

Between tree and mesh based approaches are hybrid solutions. A hybrid solution finds a compromise between cost, latency, bandwidth and path diversity of the previously mentioned approaches. A Clos Network is an example of this compromise, Traditional Clos Networks break up a single crossbar switch into 3 tiers/stages with each stage providing multiple, differing routes to the stages above/below. Clos networks provide a path diversity proportionate to the radix offered at each stage, with a relatively low average, worst case and best case latency. Additionally, the path diversity offers greater bandwidth in the event of congestion as traffic may be rerouted through different links of the network. Other related approaches include Butterfly,

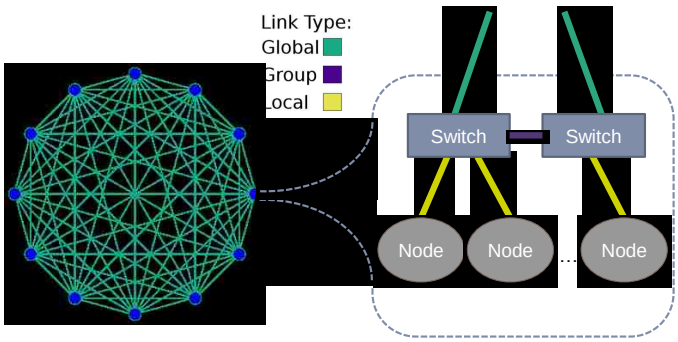

Figure 3.1: Dragonfly networks are networks built around logical collections of switches and nodes called a group. Groups within a dragonfly network are fully connected to each other by optical links. This figure is an example of a dragonfly network with 12 groups. Link connections are color coded to show local, group and global links. On the left side of the figure is the fully connected group configuration. The right side illustrates the structure of an single group.

the improved Flattened Butterfly, and Dragonfly networks [71]. Dragonfly topologies see use in many modern systems [70]. Dragonfly networks combine high radix routers and create virtual routers called a group which are fully connected to other groups by optical links. In this dissertation we will refer to local, group and global ports. Local ports connect a router to a compute node or NIC. Group ports connect routers within the same group together. For a visual reference the reader may refer to Fig 3.1. The topology within a group can vary, dependent on the requirements of the system and workloads. Global ports facilitate inter-group traffic and use optical links so that they may reach larger distance than is practical for electrical cables. One downside of optical cables is that they can suffer an increased cost compared to their electrical counterparts (depending on distance, amount purchased). Because global (optical) links are more expensive than traditional electrical links, the cost of procuring a HPC network may be decreased by reducing the number of global links. However,

reducing the number of global links has an impact on the bisection bandwidth of the network. Bisection bandwidth is defined as the minimum bandwidth across any two evenly divided partitions of nodes. Bisection bandwidth has been observed as an important indicator of application performance on many systems in the past [119, 21]. Quantifying the tradeoffs between workload performance and the number of global links in a large-scale dragonfly network is one of the contributions of this work.

Modeling network power

While performance has traditionally been the chief concern of HPC scientists and engineers, as the scale of systems increase, availability of power has become a major concern. Because Exascale systems must become an order of magnitude more power efficient, we must increase efficiency wherever possible, including in the network. Modeling network power is relatively simple, because modern networks are not power proportional, that is the ports consume the same amount of power whether they are sending data or idle. Because of this, we can generally model the power of a network link as either on or off. These models become slightly more complicated as we consider two other link parameters, namely operating frequency and link width. While changing link frequency can reduce power it comes at a cost. First, there is a downtime as ports synchronize with each other after each adjustment and secondly, a reduction in frequency increases the effective latency of the link. Because of these costs, throughout our work we assume a stable link frequency. A link's width is the number of lanes that an individual link has, these may be disabled/enabled individually to provide a greater selection of bandwidth/power options without impacting the latency of the link. Modern network links are typically configurable to either a 1X, 4X, 8X or 12X width. With this small set of configurations, we can model power an energy of a network with relative ease.

Techniques for network power saving

There is a history of work targeting power savings in the network fabric. Hoefler [57] provided a survey of network-based, energy conservation techniques. This work discussed challenges and motivated opportunities for a variety of network fabrics. Possible mechanisms included hardware based solutions, e.g. Energy Efficient Ethernet and design of network topology, as well as algorithmic approaches, such as communication/computation overlap and process migration. For our work, we are interested in approaches which manage the network fabric by adjusting the capability of individual links. These strategies can be divided into two categories; approaches which statically determine fabric requirements and approaches that dynamically tune the fabric to application usage. In 2012, Laros et al. [75] provided insight on how statically scaling the CPU and network can provide power savings on Cray XT systems. In their results it was found that very few applications fully utilized the available bandwidth and that they could scale back the network to 50% of capability with very small execution time increases for the majority of applications. One of the significant contributions of this work is that the experiments utilized real systems and applications, rather than simulations. While their work provided a proof of concept, static strategies are unable to extract power savings from short valleys of underutilization without penalizing average performance.

In 2003, Kim et al. [69] introduced a scheme for Dynamic Link Shutdown (DLS), which attempts to identify highly used links and shut down other links whose usage is below a threshold, without creating a disjoint network. If shutting down the set of underutilized links would create a disjoint network, the forwarding table is scanned for the link that provides the highest degree of connectivity. The selected link is then removed from the set of links to be shut down. The DLS technique utilizes locally adaptive routing and two additional hardware modules. In [123, 122], Totoni et al. propose the addition of hardware support for on/off link control. Furthermore, they

suggest that this control should be managed by an adaptive runtime system. This work takes a binary approach of completely disabling/enabling links (with a zero cost delay). Simulated experiments suggest that a up to 20% of total system power may be saved as a result. One of the strengths of this work is a thorough analysis of the link utilization for a set of real applications using realistic topologies. Work by Li et al.[79] presented an approach called Network Power Shifting or NPS. NPS is the idea that power saved from deactivating network links could be used to speed up computation. Additionally, they outline a hardware-based technique where switches activate downed links in response to individual messages. This strategy requires a priori knowledge of routes, so that links along a route may be enabled as well as additional hardware modules. Other techniques to disable to links forego adaptive routing or added hardware modules, instead requiring compiler support, or support from application/runtime libraries. In 2005, Li et al. [78] utilized a compiler-based scheme to predict link active and idle time, such that the compiler inserts explicit calls to turn off/on communication links. Additional research on compile-time, power aware optimizations was provided by Soteriou, Eisley, and Peh in 2007 [112, 111]. In addition to performing simulations on coarse-grained multi-chip architectures, their work considered the power savings available to embedded multicore systems-on-a-chip (SoCs). Work by Conner et al. [19] explored link shutdown opportunities by performing an analysis of individual link activity. Conner's work specifically targets MPI collective operations and suggests that software libraries explicitly inform the network of application communication needs. Later, in 2015 Alonso et al. [6] simulated the shutting down branches of links and switches in a fat-tree interconnection, while predicting power savings for a selection of synthetic workloads. This work uses adaptive routing along redundant *up-paths* of the tree, while there is only the guarantee of a single *down* path to any node in the system.

In lieu of completely disabling network links, some authors have taken the approach of reducing link voltage, frequency, or width (number of lanes per link). The advan-

tage of these approaches is that network power can be reduced without completely disconnecting endpoints, making adaptive routing optional. However, the potential power savings of these approaches is reduced compared to a complete link shutdown strategy. An approach by Shang, Peh and Jha [110] explored how Dynamic Voltage Scaling (DVS) may predictively be applied to links, based on a weighted moving average of buffer and link utilization. Additionally the authors examined the impact of varying voltage transition delays on performance. Alonso et al. [8, 7] explored the dynamic adjustment of link width in 2004 and 2006, respectively. Dickov et al. [24] simulated predictively reducing the link width of the network fabric by annotating the MPI layer. The novelty of this work relies on the use of n-gram extraction techniques [72] within the MPI profiling layer. In this work, the n-grams were a sequence of MPI (communication) calls that form an $(n-1)$-order Markov model – that allowed the authors to predict periods of underutilization.

The work of Saravanan et al. [107] is significant – in that it examines the performance of Energy Efficient Ethernet (EEE) in the domain of HPC. Their findings suggest that by default, EEE did not provide power savings, but given a reduced on/off transition delay there were overall power savings of 7.5%. Zamani et al. [132] provide a comparison of Myrinet-2000 and Quadrics QsNet to compare the energy costs of MPI operations, however both of these networks are no longer in use. Our work is primarily focused on Infiniband networks, which is the most popular network of the TOP500 (by system count).

For traditional data center environments, network power management strategies have been proposed by Mahadevan [81] and Heller et al. [54]. This work resulted in ElasticTree, a network-wide power manager. ElasticTree examines the problem of finding minimum-power network subsets across a variety of traffic patterns. Traditional data centers have different workload characteristics than HPC applications. For example, in the case of [54], the network utilization showed a bimodal usage, characterized by

working and sleeping hours of the general population.

Simulation of HPC networks

Once we have models of network performance and power, we can run simulations to study how the systems handle a diverse and complex set of workloads at scale. When simulating large scale systems we must consider many components such as the network, I/O, memory and CPU. It is important to select a simulator which finds the balance between scalability, complexity and accuracy. For these reasons, we have chosen a flexible, modular and scalable simulator called the Structural Simulation Toolkit (SST). Because of our interest in networks, we have elected to utilize lightweight, scalable modules to simulate computation, while dedicating the majority of our simulation resources to simulate the network and communication. Specifically, we utilize SST to accurately represent the packet-level routing, buffering, and internal switch characteristics of 100,000 node dragonfly networks, as well as the MPI semantics and message matching.

SST SST is a simulation framework that allows different components to be connected using a parallel discrete event simulation core. Along with the simulation core, SST provides a number of ready-to-use component libraries. SST is widely used by both industry and academic researchers. Throughout its history, the accuracy of SST has been validated in peer-reviewed publications and by hardware vendors [104, 62, 127].

Ember One of the SST libraries/components is Ember. Ember is a lightweight state-machine based event engine which replicates application communication patterns at a simulation end point. We term a single logical communication pattern a *motif*, drawing on the similar theme from Colella computational dwarfs [18]. A collection of motifs can then be arranged within each end point to represent a more complex

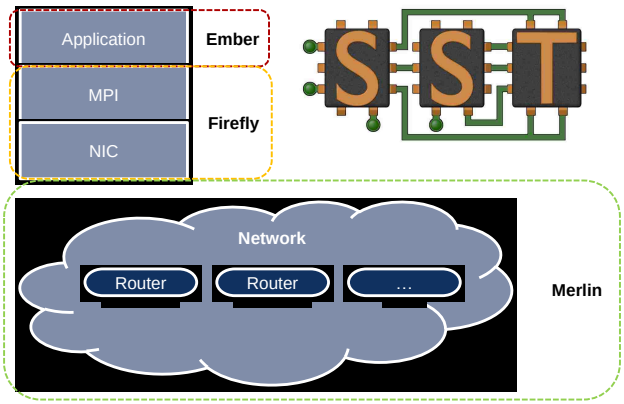

Figure 3.2: Illustration of the Structural Simulation Toolkit (SST) along with three components (Ember, Firefly, and Merlin). The SST core facilitates the use of flexible, modular components, which can simulate HPC systems at varying degrees of accuracy and scalability.

single application or even a more complete workflow.

A motif works by creating a sequence of events when prompted which contain primitive operations for communication (*e.g.* send, receive, *etc.*), computation, waits or timing markers. The events are added to a queue and then executed one by one until the queue empties. Once emptied the motif is prompted to refill the queue with additional events (while being able to see the effects and returns from the previous executed set). Thus, motifs are able to execute short sprints of events punctuated by querying or logic.

Events which relate to communication are translated into operations at the message interface layer (Firefly). For instance, a communication event is encoded in Ember and then converted into operations by Firefly which tracks the semantics associated with the request.

By using short sprints of events and very small amounts of state at each end point,

Ember is able to scale to very large simulated node counts without placing significant constraints on the amount of memory or processing required for the simulation to progress. Despite such simplicity, a collection of statistics relating to message sizes, message timing, message types *etc.* can be collected with ease.

Firefly Firefly is a pair of state-machines that implement high level functional models of the host based communication library and the network interface card (NIC) logic.

The library state-machine supports point-to-point (*e.g.* send, receive, wait, *etc.*) and collective (*e.g.* alltoall, reduce, *etc.*) operations. Message data movement between network endpoints is based on a eager/rendezvous message protocol model. Message matching is based on message tag and src. The library state-machine has several parameters such as maximum length of an eager message, latency to check a posted receive for a match, latency to copy message data between buffers and latencies that mimic the time spent in various code paths of a library.

The NIC state-machine functionally moves data to and from the host over a bandwidth constrained path (*i.e.* bus). It also has a mailbox interface to the host that the host uses to initiate sends and gets. It models latencies through a NIC and latencies of data movement over a bus. It has parameters such as host bus bandwidth, transmit and receive latency, and NIC to host latency.

Merlin Merlin consists of a set of components that allows a user to model a detailed network fabric. Merlin may be configured for a range of different network topologies. It also provides a number of tunable parameters, including buffer sizes, latencies, routing modes, and arbitration schemes.

The primary Merlin component used in this simulation is a high radix router model called hr_router. This component models a single router, including input/output

buffers, single large crossbar, and routing capabilities. The routing capability is controlled via a loadable "topology" model. The library currently supports mesh/torus, fat-tree and dragonfly topologies. All the topology models use deterministic (minimal) routing. Additionally, the dragonfly model adds two other routing modes: valiant and adaptive-local. Valiant routing chooses an intermediate group to first route to before routing to the final destination. Adaptive-local adaptively chooses between the minimal and Valiant routes using a user defined threshold. The Valiant route is chosen when the output buffer for the direct port has N (where N is the threshold) times more occupied space then the Valiant route (i.e. since there are extra hops in the valiant path, the direct path must be more congested before the valiant path is chosen).

Stalled, Active and Idle

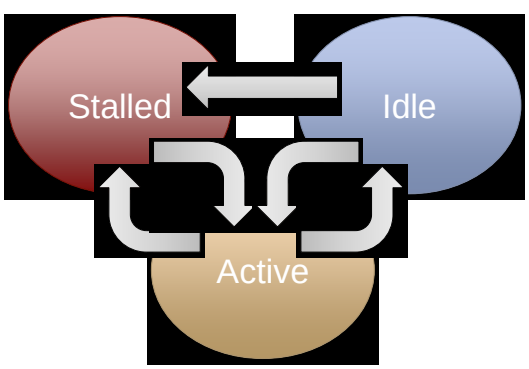

Figure 3.3: State diagram of network ports given the three states Stalled, Active and Idle.

When evaluating a system, frequently we are only given a single summarizing metric such as the runtime of an application. While runtime is a useful metric, sometimes

we desire more detail about the utilization of our system. However, when you are simulating hundreds of thousands of ports, providing meaningful representations of the system state becomes problematic. We've tackled this issue by summarizing ports with three simple states: Stalled, Active and Idle (illustrated in Figure 3.3. As a part of this dissertation, we've made additions of SAI statistics associated with the network fabric in Merlin. These counters record data pertinent to the usage of each simulated routers ports. We define these metrics as:

active: percent time a port is transmitting data.

idle: percent time a port has no data queued for transmit.

stalled: percent time not active or idle.[1]

run: simulated wall-time of the workload (normalized to 1).

By combining all three metrics, we provide a rich description of network usage, that no single metric can achieve. For brevity we refer to the collection of metrics as SAI (Stalled Active Idle). If we normalize these recorded times, SAI can be described by the following set of relationships:

$$\begin{aligned} \text{stalled} &= 1 - \frac{\text{active} + \text{idle}}{\text{run}} \\ \text{idle} &= 1 - \frac{\text{active} + \text{stalled}}{\text{run}} \\ \text{active} &= 1 - \frac{\text{idle} + \text{stalled}}{\text{run}} \\ \text{run} &= \text{idle} + \text{stalled} + \text{active} \end{aligned}$$

These relationships limit the degrees of freedom to two, specifically we can derive any single metric, given measurements of any two other. With this in mind, we simplify

[1] An example of this would be when a port has data to send but lacks credits necessary to transmit.

the presentation of SAI by using a ternary plot. Throughout this work we utilize the SAI metric to explore the utilization of each motif for a given network. Therefore, we provide the reader with a brief explanation of how to interpret the results. Fig 3.4 shows SAI in a ternary plot with five example points with a caption that provides further detail. Fig 3.5 provides further insight about what ideal network utilization looks like.

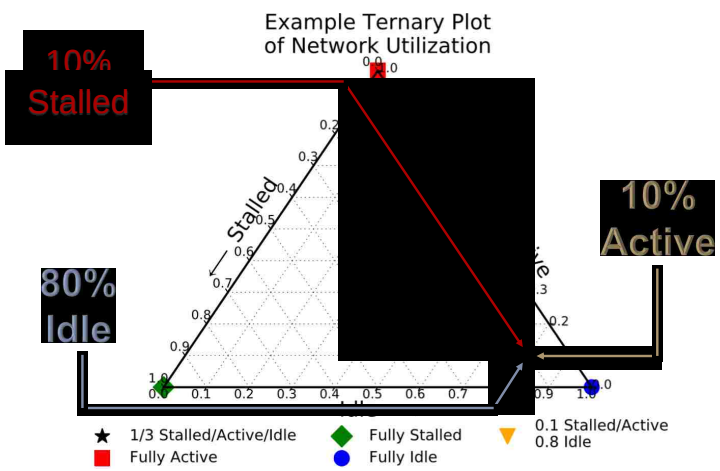

Figure 3.4: There are three axes in the plot (one for each metric) with arrows indicating the direction that a particular metric increases. The red square represents a network port which is active for 100% of the run and is located to the top of the figure. Similar points can be seen for ports that are 100% stalled or 100% idle (green diamond and blue circle). The black star represents a port that spends 1/3 of the time stalled, active and idle. The orange triangle is a port which is stalled and active 10% of the time and idle 80% of the time.

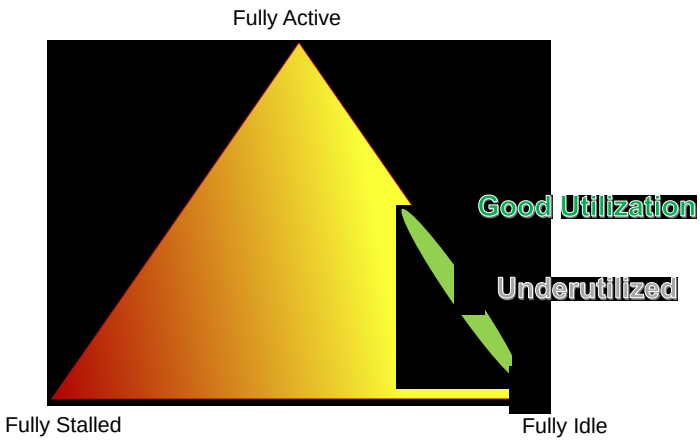

Figure 3.5: This figure provides simple guidelines for the reader about how to interpret network utilization given a ternary plot. The red regions of the plot represent an underprovisioned network, while the bottom right circle show underutilization. The area circled in green is the region of ideal utilization. The reason this does not include a fully active network is because we are considering buffered networks. In a buffered network activity leads to longer queue lengths which manifests itself as higher latency.

3.1.2 Related Work

Exascale Network Simulation

In [3], Ahn, et al. present large-scale simulations of the HyperX network topology, comparing it to other popular topologies. While we are interested in comparing dragonfly networks with other topologies (e.g. Clos-trees), it is not the focus of this work. Another simulator, LogGOPSim [60] is a LogGOP based simulator that runs on inputs of MPI traces. While it is sufficient in many cases, LogGOPSim simulates a fully connected, single hop network that does not allow for a detailed study of port-level statistics. XSim is another large-scale simulator developed at Oak Ridge

National Laboratory. While xSim has achieved considerable scale in simulation, it utilizes a simpler model of the network and has only recently incorporated models for network congestion [33]. The Rensselaer Optimistic Simulation System (ROSS) has been used to simulate large scale systems. Lui et al. have examined torus networks at Exascale size, strong scaling ROSS to use 128K cores on a Blue Gene/P system [80]. More recently the ROSS simulator was extended to included dragonfly networks, but simulation was limited to evaluation of MPI collectives [90].

Evaluating Power and Performance of HPC Workloads

Dickov, et al explored link idle times in [24] and the potential for power savings using the Venus-Dimemas simulator, given fat-tree networks. Our work has a broader focus than just network power savings, examining best practices of large scale topology design for dragonfly networks. Work by Bhatele et al. [12], explored how nearby jobs create contention in mesh-based networks. Our work is interested in similar phenomena, but focuses on dragonfly topologies which are significantly less prone to delays caused by fragmented job placement – since the diameter of a dragonfly network is constant, whereas the diameter of a mesh grows with the number of nodes. Work by Laros, et al. [75] examined how static reductions in fabric link width impacted application performance and energy. While their work was done on real systems rather than simulation, simulation allows us to take a more detailed look at the network fabric, as well as examine larger systems with newer topology designs. Zahn et al. used OMNET++ simulations for 64 node runs with an integrated power model to study the link utilization of networks running the Graph500 benchmark and NAMD on a 3D torus network [131]. They found that there were significant portions of idle time that could be exploited for power/energy savings. Unlike the work in this paper, they studied an integrated NIC/switch network, EXTOLL (Tourmalet), with different workloads and at smaller scale.

3.1.3 Simulated Environment

Network Topology

For this work, all simulations evaluate networks of 110,592 nodes. Systems of this size are believed to be similar to that of future Exascale systems. In our experiments, we vary our simulations by motifs, bandwidth and the number of global links. Our dragonfly is comprised of 24 nodes per router, 48 routers per group and 96 fully connected groups. Throughout our experiments, the number of inter-group (global) links varies, but every group has at least 1 and up to 12 connections to each other group in the network. In each experiment this will be denoted in the legends by a number between 1 and 12 followed by glbl.

Router and NIC Parameters

We simulate links with a bandwidth varying between 12.5 and 25 GBps. These bandwidths are conservative in terms of what can be expected in the Exascale timeframe, but their conservative nature means that they should be available circa 2018 and non-prohibitive in cost by the 2020 timeframe, making them a reasonable, if conservative target. The full parameters of the SST router and NIC components are provided in the following table:

Param. Name	**Value**
module	merlin.reorderlinkcontrol
flitSize	32B
port_input_lat	150ns
port_output_lat	150ns
link_lat	150ns
packetSize	2KB
link_bw	12.5 or 25 GBps
buffer_size	28 KB
rxMatchDelay_ns	100ns
txDelay_ns	50ns
hostReadDelay_ns	200ns
nic2host_lat	1ns

Processor Parameters

Each node in our system contains 5TF/s of processing power. This represents the minimum capabilities expected in the near future. Given that we simulate 110,592 nodes, 5TF/s only represents a half an EF/s computational power. To achieve true Exascale performance we would need to roughly double the FLOPs per node. While this does not present a technical challenge to the simulator, 5TF/s allows us to more closely match the expectations of next generation systems.

From the viewpoint of our simulator, the difference in processing power manifests itself in the speed at which computation portions of motifs are completed. We assume that offload NICs are in use, so that CPU capabilities do not influence the speed at which network processing is done. Additionally, offload NICs allow for further simplifying assumptions, such as reduced impact of Network Induced Memory Contention [42].

Evaluated Motifs

The selection of motifs covers a broad set of important microbenchmarks and communication kernels in HPC. The wall time to simulate a motif varies depending on many factors including the amount of simulated congestion, adaptive routing, number of simulated events and distribution of SST components across physical nodes. For simple motifs (like bcast), wall time to run a simulation may be just a single minute on a small number of nodes. Whereas, more complicated motifs increase the wall time to run a simulation and must be split across a larger number of nodes due limited memory. For example, for our parameters SST simulates bcast in a single minute, while random and sweep3d may take an hour to simulate on 32 16-core nodes.

AllPingPong The AllPingPong motif is a simple workload that divides the network into two logical groups of processes and then creates pairs consisting of one process from each group. The performance of this motif is dependent on both the bisection bandwidth and the number of hops (diameter) of the network. The communication acts in a predictable manner that is easy to reason about. Our experiments perform 1000 iterations of ping-pong, sending messages of 1024 bytes.

Allreduce In Allreduce, each node receives n-1 messages (one for each other process in the system) and then a reduction operation is performed. Our simulations measure the impact for a single iteration of AllReduce sending 4 bytes, using 1 ns. of compute time per reduction operation.

AMR3D This is a motif of a 3D adaptive mesh refinement, based from miniAMR. To accurately replicate the communication and computational characteristics of the real workload, the AMR3D motif must be given a block file which details how the mesh should be refined and defines the communication and computation that will

take place. Because blockfiles must be produced on real systems running the real workload, we were limited to runs of 65,536 nodes which utilize 59% of our simulated system. Depending on the phase that a block file represents, the communication requirements may change considerably. We simulate two different block files, one from a early time in the run which is not particularly intensive and an additional block file that is from midway through the run. We denote these two different simulations as amr3d-lite and amr3d-heavy.

Bcast A simple broadcast is performed, such that each node receives a single message from the root node of the broadcast. We use similar parameters for broadcast as Allreduce, one iteration, 4 bytes of data, 1 ns. of compute, but with a root parameter set to rank 0.

FFT3D FFT3D is a discrete conversion of a signal from its original domain, into the frequency domain with O(N log N) complexity. Fast Fourier Transform is considered one of the most significant algorithms of all time and has applications throughout digital signal processing. Our Fast Fourier Transform 3D motif uses block sizes of 1,992 for nx, ny and nz, with 125 FLOPs/element.

Halo3D Halo3D performs a 7-point stencil operation. That means a data transfer to and from each direction in 3 axis and a central point. Problem sizes in the X,Y and Z direction are set to 100. Per cell there are 16 variables being computed

Halo3D26 Identical parameters to those used in Halo3D are used In Halo3D26. However, this motif represents communication between a larger number of neighbors, each process has 26 other neighbors that they communicate with, where each neighbor represents an adjacent point (including diagonals) in a three dimensional space.

Random In the random motif, each node selects one other node on the system to send a message to randomly at each iteration. Like AllPingPong, Random utilizes the network resources significantly, but differences in distribution of traffic may create hotspots on the network. The Random motif sends 1024 Byte messages for 10 iterations with a wait-all synchronization between each iteration. At each iteration random destinations are recalculated.

Reduce Reduce is similar to Broadcast, except the flow of data is reversed (each node aggregates and reduces data rather than propagating it). Our reduce parameters are identical to broadcast for iterations, message size and compute time.

Sweep3D Sweep3D models a wavefront propagating through a mesh, where each CPU represents a 2D column in a 3D mesh. We use values of 384 and 288 for values of pex and pey, respectively. The problem size in the X, Y and Z dimension is set to 100. Per cell there are 6 variables computed and the KBA (Nz-K blocking factor) is set to 10.

3.1.4 Methodology and results

In this section we review the results of the motif simulations, examining the network characteristics of each workload and provide observations to assist in the design of next generation networks. We begin with an assessment of motif runtimes as we taper the number of global links in the network. This is followed by an analysis of how adjusting the bandwidth of the entire network (from 25GBps per link to 12GBps per

link) impacts motif performance. Finally, we look at the potential for power savings in the network.

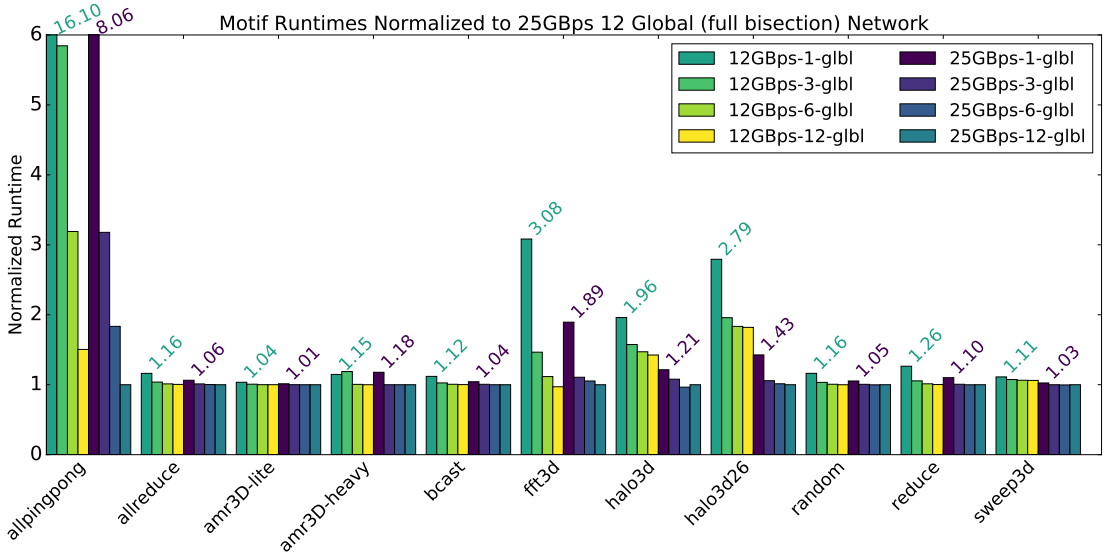

Figure 3.6: Normalized runtimes for a variety of motifs on different simulated networks. The baseline for the normalization is each run on a 25GBps per link network with half bisection bandwidth (12 global links connecting each group to each other group). The difference between simulations are the number of (1) the number of global links and (2) the bandwidth of all links. All runs other than AMR3D run on the full network (AMR3D runs utilize 59% of the nodes for reasons explained in §3.1.3).

Performance impact of reducing the number of global links

Reducing the number of global links impacts application performance by reducing available bisection bandwidth. In our simulations we begin with a network that has half bisection bandwidth. This means that each group in the dragonfly topology has 12 connections to each other group. Given 96 groups, each group has $12 \times 95 =$ 1,140 global ports. This turns into 54,720 total global links. A network with this many global links would likely be prohibitively expensive. Additionally, the expense of procuring the extra links is only a part of the total cost which includes powering

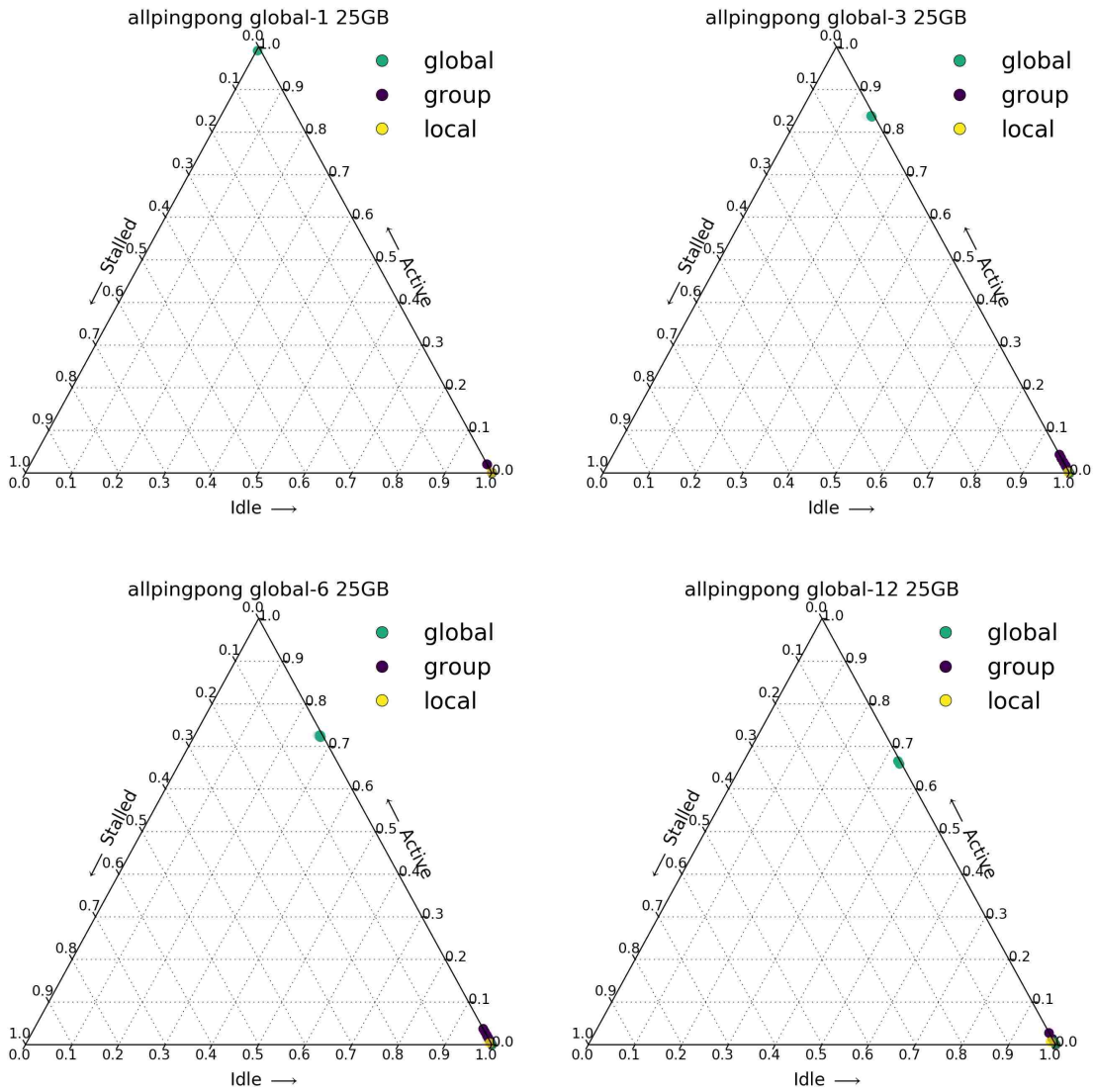

Figure 3.7: Ternary plots of ≈400k ports for allpingpong motif on a 25GBps per link network with 1, 3, 6 and 12 global links. Allpingpong was one of the most sensitive motifs to a reduction in global links. As global links are decreased, remaining global ports increase to near 100% active.

the link and purchasing higher radix switches.

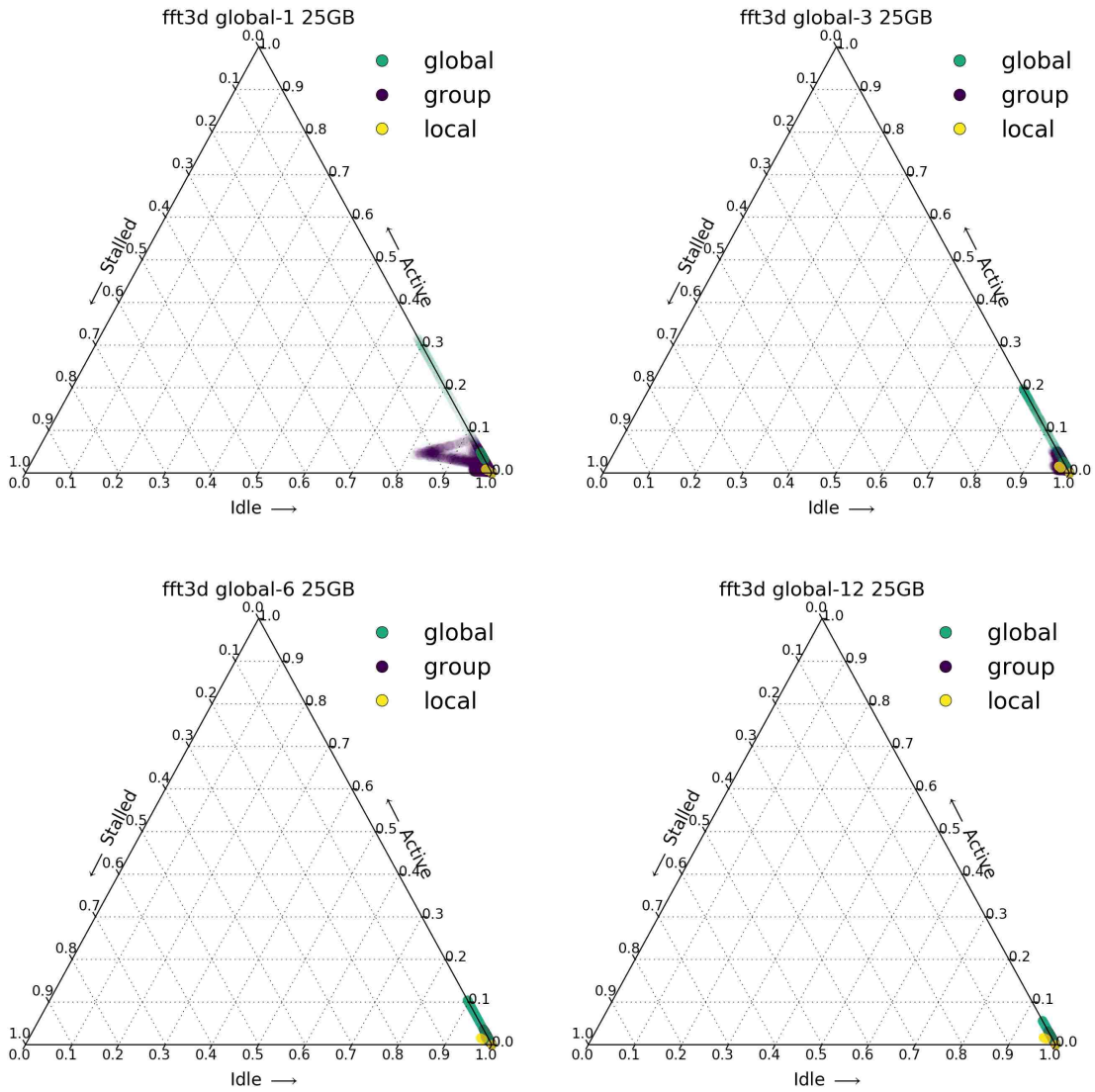

Figure 3.8: Ternary plots of ≈400k ports for FFT3D motif on a 25GBps network with 1, 3, 6 and 12 global links. Similar to allpingpong, global links become increasingly active as we reduce their number. When the network reaches 1 global link per group, we see congestion (stalled cycles) on the group links.

Research Question In this section we ask, how many global links or what percentage of half bisection bandwidth does a Exascale network require for reasonable performance?

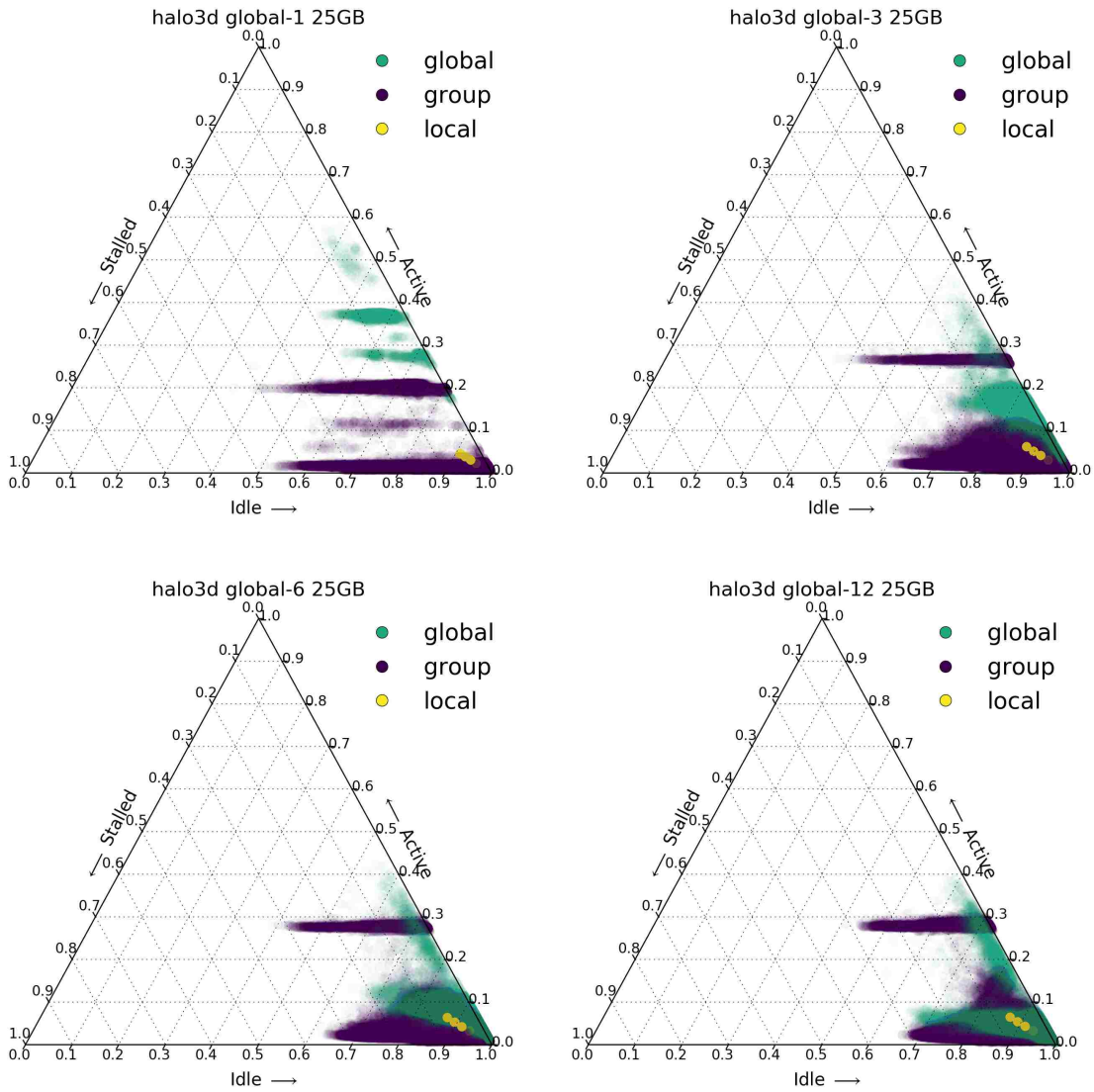

Figure 3.9: Ternary plots of ≈400k ports for Halo3D motif on a 25GBps network with 1, 3, 6 and 12 global links. Unlike the FFT3D motif, even with half bisection bandwidth the Halo3D motif sees congestion. Group ports spend a greater percentage of time stalled as we decrease global links.

Methodology Our experiments evaluate the performance of the motif and the network as we move towards quarter 1/8th and 1/24th bisection bandwidth (27,360, 13,680 and 4560 total global links, respectively). Throughout this work we use the

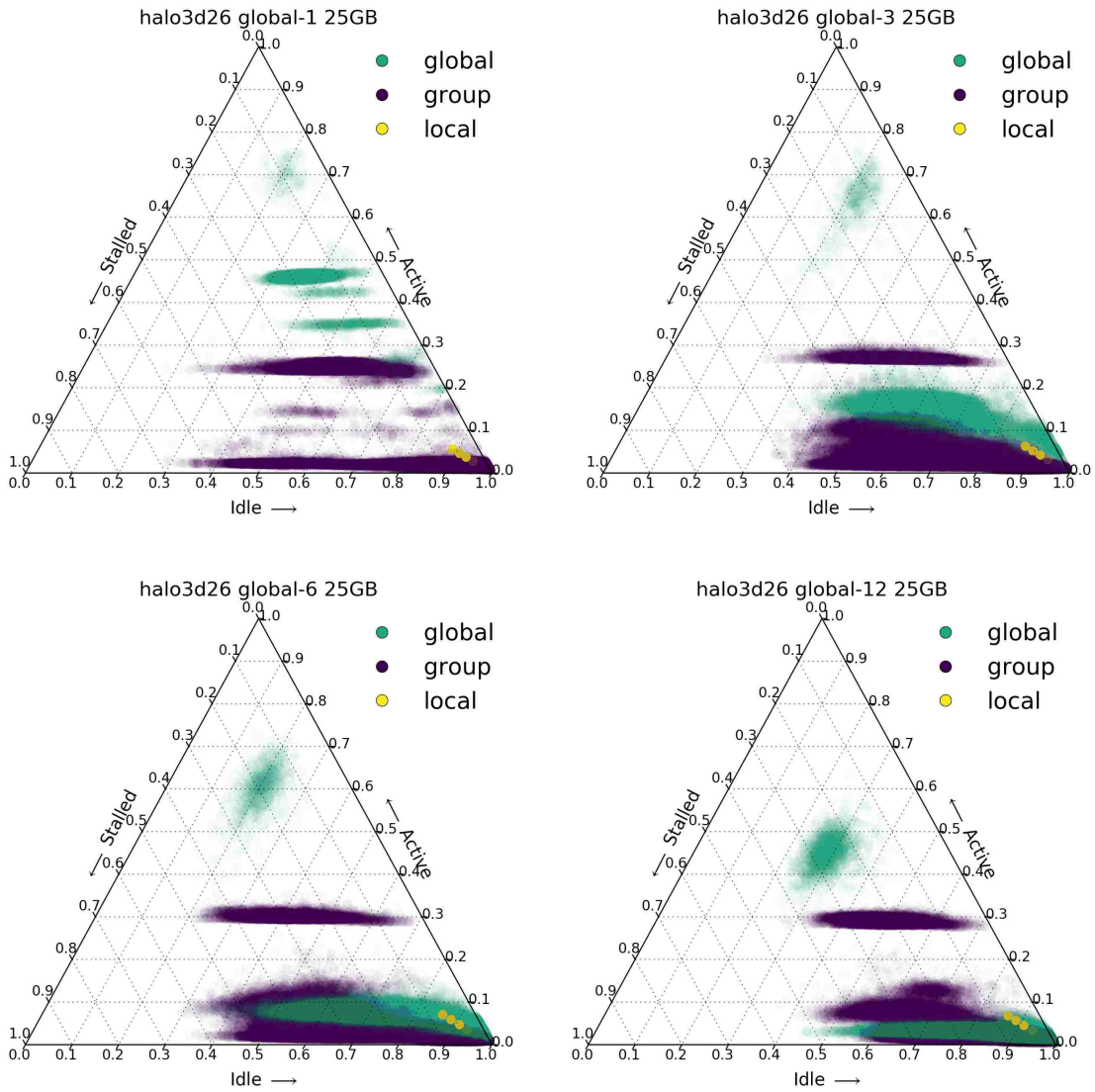

Figure 3.10: Ternary plots of ≈400k ports for Halo3D26 motif on a 25GBps network with 1, 3, 6 and 12 global links. Compared to Halo3D, Halo3D26 produces a greater volume of traffic to a larger number of neighbors. This increased traffic results in a greater amount of time stalled for global and group links.

terminology 12-global to refer to a half bisection bandwidth network for our simulated topology. Similarly, a 6-global, 3-global and 1-global refer to a quarter, 1/8th and 1/24th bisection bandwidth network.

Outcomes Fig 3.6 shows the impact of reducing the number of global links on the runtime of evaluated motifs. Looking at the results for 25GBps networks, as we reduce global links to 1/24th of the half bisection bandwidth, we see unacceptable increases to the runtime for allpingpong, AMR3D, FFT3D, Halo3D and Halo3D26 (806, 118, 189, 121 and 143%, respectively). The only motifs that see mild increases to runtime are the motifs that utilize the network the least, such as Sweep3D, broadcast, and AMR3D-lite. While 1/24th bisection bandwidth is clearly a poor choice, the majority of motifs see minimal degradation for topologies of 3 global links per group router. Specifically, all motifs other than allpingpong see a more modest (0-11%) increase to runtime for 75% reduction in global links. Allpingpong, which is essentially a measure of bisection bandwidth and network diameter, shows a more direct penalty to runtime as we decrease global links. In the ternary plot for AllPingPong (Fig 3.7), we see increased activity for global links, which continues to grow as we remove available global links, becoming fully active for the 1-global network. Similar increases in global link activity is seen in Fig 3.8-Fig 3.10. However in these figures, group links see a corresponding increase in stalls as the traffic on each global link increases.

Another observation is that the Halo3D and Halo3D26 motifs experience congestion and stalled cycles even with a 12-global network. Because these two motifs are stalled to begin with, they experience less of a relative increase to runtime than motifs like FFT3D which have relatively few stalls until placed on a 1-global network (Fig 3.8). One of the reasons for the stalls observed in Halo3D26 is that the workload does not map to a dragonfly network as well as traditional mesh-based networks. In future work, more refined mapping strategies may be able to improve this.

For brevity we limit the number of ternary plots we present to those that are most interesting, but we should note that across our results we observed that local links were generally clustered together more tightly and more idle than global or group links. Additionally, the reader may notice that within Fig 3.6, that Halo3D26 25GBps-6-glbl

actually sees a performance improvement compared to a 12-global run. We believe this is likely due to network stalls hitting a critical threshold in the routing algorithm that enables adaptive routing more readily with this number of links.

Performance impact of reducing total bandwidth (link widths)

One of the common methods proposed for saving power on the network is reducing the link width, which only partially limits a link rather than completely disabling it.

Research question In this section, we ask what are the performance implications of reduced link width are on the evaluated motifs?

Methodology To evaluate this, our simulations only reduce link bandwidth and keep other parameters such as latency identical to those used in the 25GBps simulations.

Outcomes Results in Fig 3.6 suggest that for the less bandwidth intensive motifs, (such as AllReduce, Random, Reduce and Sweep) we may be able to statically reduce link bandwidth by 50% for a run and see modest increases to runtimes. This topic has been explored before with regards to smaller systems [75] using Cray Seastar interconnects and our results suggest that static reductions in network bandwidth may continue to provide power saving opportunities with modest runtime costs for a subset of workloads at Exascale. Another interesting observation comes from comparing the runtimes and ternary plots of Halo3D26 simulations of 12.5GBps networks (Fig 3.11) and 25GBps networks (Fig 3.10). Decreasing the total available bandwidth to this bandwidth sensitive application increases runtime as expected, however given the volume of stalled cycles in the 25GBps run, we expected Fig 3.6 to show a greater than 2X increase to runtime for a 50% link reduction (25GBps to 12GBps). Examining

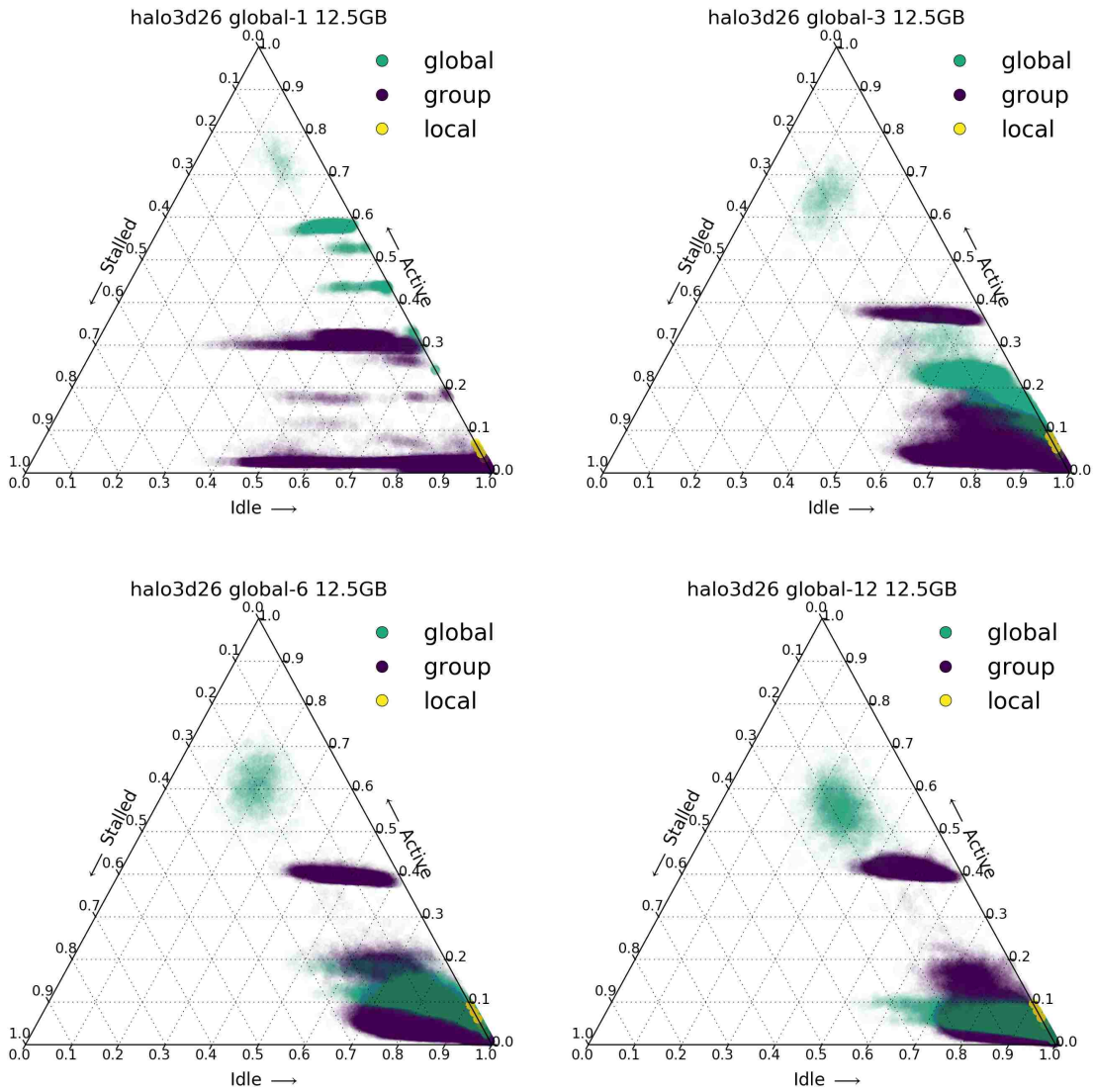

Figure 3.11: Ternary plots of ≈400k switch ports for Halo3D26 motif on a 12.5GBps network with 1, 3, 6 and 12 global links. Compared to Fig 3.10, the 12GBps run surprisingly shows a decrease to stalled time and an increase in active time.

Fig 3.11, we see proportion of time spent in stalled cycles decreases as ports spend an increasing proportion of time active. The reason for this behavior is that the adaptive routing threshold in Merlin is determined by the number of packets waiting in the

outbound queue. As we decrease the bandwidth to 12.5GBps switches begin to buffer larger numbers of packets, which leads to more frequent enabling of Valiant routing, which reduces stalls. With the reduction in stalls on the group and global ports, the figures show that local ports transition to an almost entirely active or idle state.

Potential for power and energy savings

One of the goals of this paper was to examine potential power and energy savings on large scale dragonfly networks for relevant workloads. In this section we have several research questions, namely

1. What are the energy savings of a power proportional network?
2. What are the power savings from tapering global links?
3. What are the power savings from a static reduction to link width?
4. Is there potential for dynamic power saving solutions?

Methodology Since we measure the per-port idle time for each workload, we can derive upper bounds on energy savings if a network was power proportional. A network is power proportional if the network only consumes power corresponding to the amount of data in transmission. While modern networks are not power proportional, a large body of work has proposed dynamically and statically altering the width and frequency of network links to reduce the amount of wasted energy. The established work varies from shutting down the link completely [69, 122] to approaches that only partially reduce link frequency or width [75, 110, 24]. The success of each approach is dependent on a number of parameters, but dynamic solutions which adaptively alter links must ensure they can disable/enable a link within a window of idle time. Idle time windows smaller than the disable/enable time must be forfeit as opportunities for power savings.

To explore these questions, we require an estimate of how much power links utilize at different widths and frequencies. Many of the existing power estimates in this area are either theoretical or pertain to architectures that are more than a decade old. While we are not suggesting these estimates are invalid, we believe it is worthwhile to take empirical measurements of network power and include our findings in this section.

Beginning with optical global links, 3W of power is the commonly used assumption for transceiver power [110, 1]. We were unable to perform empirical measurements on any optical interconnects for this work, so rely on industry datasheets.

Considering electrical interconnects, Soteriou and Peh [112] reported a 0.3W and 0.2W power consumption for IBM Infiniband 12X LPE TX and RX, respectively. This measurement can be used to determine potential power savings when reducing the width of an individual link. In our measurements we used WattsUp! power measurement device to record switch (Mellanox MTX3600) power as we adjust link widths and frequencies. We used PowerInsight [74] to measure power of Qlogic QDR Infiniband NICs. When we adjusted the network from a 10Gbps 4X network to a 2.5Gbps 1X network we found savings of 1W per port on the Mellanox switch and 0.57W for the Qlogic NIC. At a reduction from 4X to 2X the NIC saw a reduced savings of only 0.29W. Each reported power savings is the average of 40 measurements with a standard deviation of 0.05W and 0.16W for switch port and NIC, respectively. Our measured power savings are significantly less than the theoretical savings commonly cited in literature. This is in large part because the Serializer/Deserializers are not disabled on our measured hardware to reduce power savings. Regardless, these numbers provide a lower bound of what we would expect to save on future systems and we can provide an upper bound using models and measurements of previous literature.

Given these power estimates we can assume that electrical switch ports on our network

(local and intra group ports) consume 0.65MW of power for a 4X link width or almost 2MW for 12X link width. The optical global links take up an additional 0.33MW of power for our half bisection topology. The NICs increase this by another 0.76MW (6.83W/NIC in our measurements). **Total estimates for the power consumed by the fabric would be between 1.73 to 3.02 MW of power**[2].

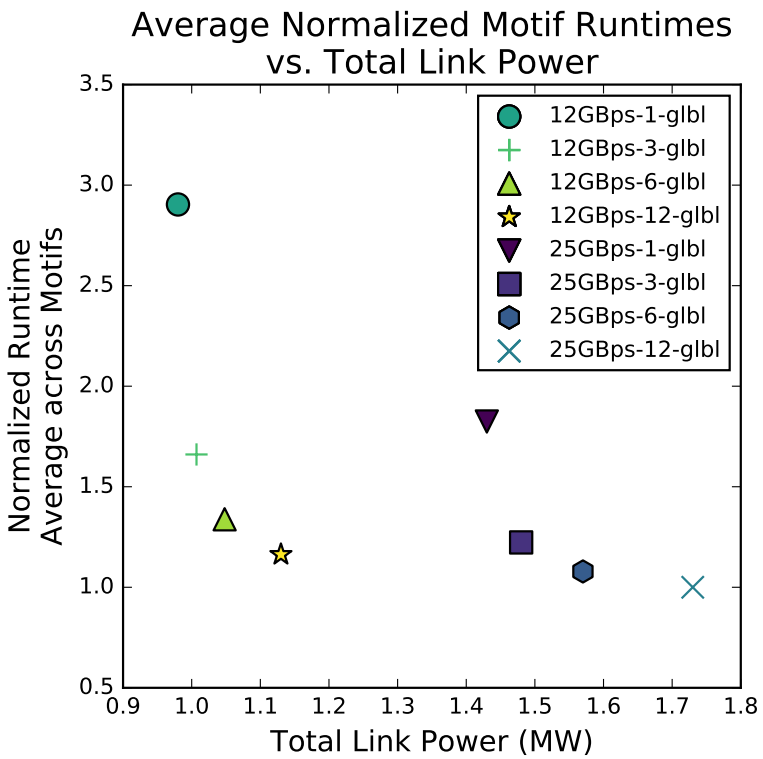

Figure 3.12: This figure shows the normalized runtime (averaged across all the motifs of a given network simulation) against the total link power costs of the network. Networks with a good balance of power and performance include the 25GBps 6-global and 3-global, as well as the 12GBps 12-global networks. For the power estimates we assume 4X links The per port power costs in this figure are set to 0.5W per RX+TX for electrical ports and 3W per transceiver for optical links.

[2]This is not total network power which would be higher due to additional logic in the switches, switch cooling, etc.

Outcomes – power proportional network The best case power savings would be if the network was power proportional, so that we only paid power costs for active time. Looking across all of the motifs evaluated, Halo3D26 had the largest amount of active and stalled ports, so we use this motif as a lower bound on the amount of power that could be saved from a power proportional network. For this motif, the average percent of idle time for global ports was 84%, which would be a power savings of 0.28MW just for global ports. If we consider electrical ports, they average 82% idle which is an additional power savings of 0.71MW to 2.13MW (4X and 12X, respectively) of additional power savings. **This totals 0.99 to 2.41 MW of potential power savings within the network of our simulated system for the most communication intensive motif we evaluated.** Other motifs like Sweep3D use less network resources and leave links 99% idle on average for our simulations.

Outcomes – reducing global links As shown in previous sections, most of the motifs simulated do not require half bisection bandwidth to achieve satisfactory performance. By reducing the number of global links we not only save money on the cost of the initial system procurement, but save on power costs throughout the lifetime of the system. **Specifically for the simulated 25GBps network, we can reduce global links by 50% and save 164KW of power. A further reduction to 1/4 or 1/12 global links results in a savings of 246KW and 300KW, respectively.** However, as demonstrated, a reduction to 1/12 the number of global links is impractical for performance reasons. Reducing global links to 25% of half bisection incurs some performance penalty for Halo3D and FFT3D motifs. However, this reduction may be practical for 75 GBps HDR networks, projected for 2017.

Outcomes – reducing link width A static reduction in link width for the entire network is a transition that could be enacted at low frequency from a power aware resource manager, dependent on the characteristics of the workload. Given our simulations of 12.5 GBps networks, we can estimate the amount of power saved by reducing link widths. Because we keep latency constant throughout our simulations, this provides an accurate mapping to a reduction in link widths for small messages, where link width reductions do not significantly impact the performance (latency) of small sparse message communication. Link width reductions are not being done in combination with link speed reductions, so the impact of width reductions is mostly limited to medium/large messages. Here, we present a pessimistic estimate given our empirical measurements of power reduction and a more optimistic estimates, derived from the power savings in literature.

First, lets examine a pessimistic model of static power savings. Given 327,168 electrical group and local switch ports, each of which at minimal would save 0.5W per port for a 2X reduction in link width, our group and local switch ports could save 164KW. Adding the 110,592 NICs, which minimally might save 0.29W for a reduction from 4X to 2X link width, we gain an additional 32KW of savings. If we consider the global links for a network that has quarter-bisection bandwidth, there are an additional 109,440 ports to derive savings from. If we assume each of these ports could save 1.5W for a reduction to 2X link width, we gain an additional 164KW of power. **In total, our lower bound for power savings, given a static reduction to link widths totals 0.36MW.** This is around 1.8% of the projected Exascale power budget. **If we consider a network built with half bisection bandwidth and more optimistic models of power savings (mentioned in §3.1.4) we can increase this estimate to 0.60MW or 3% of an Exascale power budget.**

Motif	% idle	% events > 1μs	% events > 1ms
FFT3D	99%	66%	0%
Halo3D	89%	44%	0%
Halo3D26	83%	37%	0%
Sweep3D	99%	8%	0.4%

Table 3.1: Median time idle of ports across four motifs, as well as the percentage of idle events greater than 1μs and 1ms.

Outcomes – dynamic link width The difference between the static 0.60MW power savings and the 0.99MW power savings possible in a 4X power proportional network may be reclaimed by dynamic network power strategies such as those proposed in [69, 110, 112, 122, 24]. However all of these dynamic strategies require idle intervals of network ports sufficiently long to disable or slowdown some portion of a network link and bring it back up before it becomes active. While we are not proposing any new solutions to predict idle and active intervals in this work, we present summary statistics of the duration and percentage of time that the network is idle for our simulations, which informs future endeavors in this area. Typically, clock-matching Phase-Locked Loops are viewed as the bottleneck to increase a link's width after it has been decreased (aligning input and output phases takes around 400ns [31]). For our work, we consider any idle interval greater than 1μs as an opportunity for dynamic power savings strategies. Additionally, we present the percentage of idle events whose duration is longer than 1ms.

In Table 3.1 we report the median idle time for 4 workloads as well as the percentage of idle events greater than 1μs and 1ms in duration. It's clear that most of the network ports remain idle for a majority of the runtime. These results suggest that a large portion of the idle events (8-66%) could be targeted by dynamic power savings strategies. Only Sweep3D has idle periods longer than 1ms (0.4% of idle events).

3.2 Chapter Conclusions

With power becoming a primary concern in Exascale system design, we must extract power savings from every facet of the system. In this chapter we have explored the tradeoffs between power and performance at Exascale through simulation of dragonfly networks for a variety of workloads. Our models show that a dragonfly network of 100,000 nodes would consume between 1.73 to 3MW of power for a 4X or 12X network, respectively. We have found that significant power savings can be realized by scaling back links during idle periods, such that 2-12% of the total system power budget may be reclaimed.

We assessed how link width as well as global tapering impacts motif performance. As an upper bound, a power proportional network would achieve power savings of 82% of the total network power costs for the most communication intensive workloads. As a lower bound, we've shown with conservative estimates of power, statically adjusting link-width would save 20% of the network power cost (0.36MW). While some motifs were sensitive to these reductions, we observed 7 out of 11 motifs were able to withstand significant reductions available bandwidth with only minor impact to runtime. In systems where applications are not run across the whole machine, heterogeneous networks may be a possibility, which merits further study for organizations that do not run demanding application types at a whole system scale. In addition, we have shown what configurations of network bandwidths and global link counts provide the best balance between power costs and execution time for the workloads studied.

As a final contribution, this chapter has introduced a new method for visualization/analysis of hundreds of thousands of network ports. We identified the relationship between stalled, active and idle states, showing how these states can effectively summarize the utilization of a port. As we conduct further research on HPC networks,

we believe this technique will provide valuable insights about the effectiveness of network designs.

Chapter 4

Monitoring Large-scale Networks

In the previous two chapters, we have explored performance and power of HPC networks, examining communication from varying levels of the network stack and the network fabric. In each chapter, we provided the motivation for dynamic solutions that could adapt to their environment, to achieve the best possible performance and power. In Chapter 2, two different solutions were given (bandwidth throttling and core reservation) for NiMC. The first solution (bandwidth throttling) was desirable when you could keep the RDMA bandwidth below a certain threshold, but as the volume of injected traffic increased the target node would be better suited by core reservation. Similarly, in Chapter 3, we saw how a dynamic link-width reduction could provide additional power savings beyond what a static link width reduction might. For each of these dynamic solutions, knowing when to make an adjustment requires knowledge of the environment. This information might be the amount of RDMA traffic being injected between two nodes, or the percent time a port has been stalled, active or idle. Collecting and analyzing this information is dependent on the capabilities of monitoring systems. The monitoring system must gather, store and disseminate information about components, while avoiding perturbance of the underlying application. This is a challenge on individual nodes, but for a distributed

system the difficulty increases at scale.

In this chapter we explore how large-scale network monitoring is currently done in practice and how it might be improved, to become more scalable and responsive through in-network push-based solutions(Section 4.1). By necessity, scalable monitoring and data collection implies a hierarchical or tree-based structure. To facilitate efficient design of these hierarchies, we extend a canonical model for parallel computation, improving its accuracy when modeling large-scale data aggregation (Section 4.2). Our model allows us to understand the scope and limitations of data collection in HPC networks. This knowledge is crucial for designing effective and efficient dynamic solutions that future systems demand.

4.1 In-network, Push-based Monitoring

Traditionally, computer and computational science fields have been dominated by computation-intensive problems. In recent years, we have seen a dramatic rise in data-intensive problems, involving the transmission and analysis of massive volumes of data from large networks of sensors or other acquisition devices, simulations or social networks. In addition to new generations of algorithms and data management technologies, the efficient extraction, filtration, mining and knowledge discovery from these data require scalable approaches for network traffic engineering and quality of service (QoS).

Effective traffic engineering and QoS services rely on capabilities that enable localized as well as more holistic profiles of network interactions and performance. This means that the network (both its endpoints and intermediate points) must be instrumented with monitoring capabilities. Current network monitoring typically employs the Simple Network Management Protocol (SNMP). A major shortcoming of SNMP is that it is not scalable for retrieving large collections of data. Wu and Marshall show

that SNMP does not provide sufficient mechanisms to achieve payload efficiency when sending a stream of data from even small routing tables [45]. This inefficiency is due largely to extraneous protocol data and SNMP's pull-based approach.

To continue to be feasible and effective, network monitoring techniques must evolve to become more responsive and less intrusive. We propose a new distributed, push-based approach to network resource monitoring that promises to be more scalable, efficient and responsive than the traditional SNMP-based approach. Network switches have grown beyond mere ASICs into full machines that run unmodified Linux kernels with more standard interfaces and capabilities. We leverage these capabilities for on-switch or in-network information collection, dissemination, filtration, aggregation and analysis. Our push-based approach reduces the feedback loop of network diagnostics and enables network-aware applications, middle-ware and resource managers to have access to the freshest available data.

In this part of the dissertation, we detail our motivations, approach and preliminary experiences with this new approach. Our prototype framework utilizes the OpenTSDB framework [96]; in this environment, we implemented two basic data collection agents to demonstrate the benefits of in-network, push-based network monitoring. Our preliminary results demonstrate performance benefits and show the feasibility of extending on-switch monitoring to production systems.

4.1.1 Background and related work

SNMP in data center networks

Traditionally, SNMP has been used for collecting information about network devices and protocols. SNMP data collection can be divided into two approaches: a pull-based approach, where managers query SNMP agents for system information and a push-based approach, where SNMP traps are triggered, leading to an asynchronous

communication from the agent to manager. Both of these methods rely on an universally defined data structure called the Management Information Base (MIB). While the MIB guarantees a universal view of network devices and protocols, its usage often leads to unnecessary overhead.

SNMP traps are a feature of the protocol meant to communicate asynchronously from the SNMP agent to manager. Traps are classified into 6 generic types: coldStart, warmStart, linkDown, linkUp, authenticationFailure, egpNeighborLoss, and enterpriseSpecific. Usage of SNMP traps have traditionally been focused on providing coarse grained information about device status. Though custom traps can be developed within the enterpriseSpecific domain, SNMP trap still suffer much of the same structural overhead since both managers and agents must parse the MIB on message transmissions and receipts.

One of the major operational issues with SNMP in a large corporation, for example Yahoo, is that polling SNMP on network devices could result in very high CPU utilization for the device. This high CPU utilization may delay or even halt important information processing, such as route re-calculation. By using a push model, we do not have to restrict ourselves to SNMP polling once every x minutes. Instead, we can have the device report data on-demand as data changes such that it does not impede other more important jobs.

One other issue with SNMP is that certain components of the protocol are not very efficient. For example, a router normally stores its route table in a hashed format to conduct fast subnet-to-route lookups. However, SNMP responses for the route are required to be returned in lexicographical order per RFC 1213. Therefore, for each SNMP request the router receives, the hash table must be sorted lexicographically before a SNMP response (protocol data unit) can be built [17]. The larger the route table, the more CPU intensive the sort. On the other hand, by using a push model, we are more flexible and do not have to make the expensive and unnecessary sort

before reporting the route data.

Real world usage of SNMP typically suffers from an additional deficiency in terms of reliability and robustness to failure of management nodes. All system information is delivered from the SNMP agents to the SNMP manager. If a manager node fails, the agents cease to report the information needed for monitoring. Though distributed approaches to management have been explored [115], in practice SNMP connections are formed by agents reporting to a single management node [108].

One alternative to traditional collection is to perform push-based monitoring from the switch into existing, scalable monitoring solutions. Modern switches have become more powerful, hosting multi-core CPUs, in addition to several gigabytes of memory and optional solid state storage. These switches can now be leveraged to enable intelligent reporting of metrics without the burden of traditional pull based solution and SNMP overhead. We believe that by utilizing on-switch monitoring network-aware applications, middle-ware and data managers will have access to the freshest available data. This is possible due to lower overheads in processing, a shorter feedback loop and intelligent collection.

OpenSM and HPC Networks

Rather than SNMP, high performance networks typically perform monitoring through an actor called the subnet manager or SM. The requirements of the subnet manager are established in the Infiniband specification and commonly implemented as a part of the Open Fabrics Enterprise Distribution (OFED). In this distribution, the subnet manager is referred to as OpenSM. OpenSM is in charge of establishing a subnet and managing/monitoring the connections within. Each host device participating in a subnet must run a daemon or agent process. Any request to adjust or query a portion of the subnet must be made through the OpenSM via specific subnet management packets (SMPs). These request are routed to the OpenSM from the

host, then OpenSM makes a query to a port. The queries may take a variety of forms such as the number of bits transmitted by a port or the active connection speed. In many ways this is not very different than the system that SNMP uses – both being centralized pull-based approaches. While this section focuses on SNMP, the shortcomings of network monitoring apply to both data center and HPC networks.

4.1.2 Motivation

In the era of large clustered systems, there has been an significant demand in inter-node communication bandwidth within a data center. Many applications in large Internet companies and national laboratories consist of thousands of distributed nodes. Efficient communication among these nodes is critical to the performance of these applications. For example, Yahoo has been running one of the largest Hadoop clusters [130] consisting of more than 10,000 nodes. Each of these nodes must perform significant data shuffling with other nodes in the same cluster to transport the output of the map phase before the reduce phase can be performed.

To control costs while enhancing performance, new designs for next generation networks [4, 91, 40, 46, 47] have emerged in recent years. The newly proposed networks leverage a large number of commodity Ethernet switches and provide a very low over-subscription rate for all connected hosts. In contrast to a single high radix switch, several low-radix switches decrease the procurement cost, but increase the complexity of a system.

This additional complexity poses significant challenges, particularly for network monitoring, managing and troubleshooting. The number of switches and the number of links among them are significantly larger than in the comparative traditional networks. And if any component of a cluster fails, the configurations on the rest of the devices may need adjustments to minimize the failure's impact and re-balance the

traffic. This needs to be done in a short time frame, ideally, by an automated system. To meet these challenges, we need a scalable, fast and efficient monitoring system.

4.1.3 Prototype framework

The OpenTSDB monitoring framework

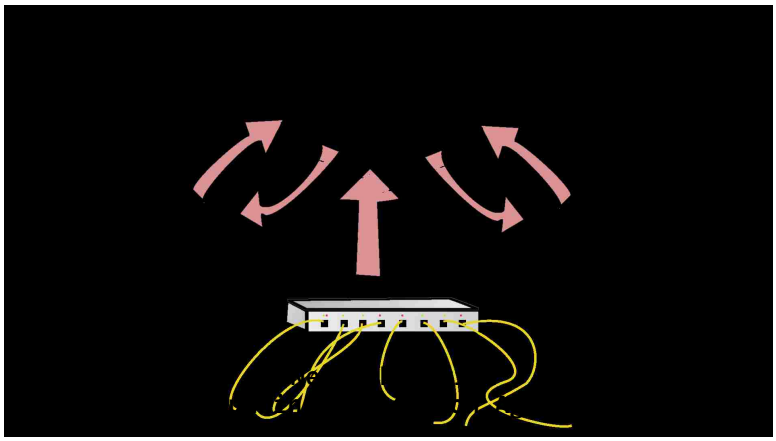

Figure 4.1: A diagram representing our usage of the OpenTSDB monitoring framework.

We chose to utilize the Open Time Series Database (OpenTSDB) as a scalable monitoring back-end [96]. OpenTSDB is a system built to store, serve and index system metrics on top of HBase [53] and is designed to handle billions of metrics per day. We selected OpenTSDB as our monitoring solution for several reasons. First, it is free and open source. Secondly, by utilizing a distributed data store such as HBase, it is capable of higher throughput than some traditional monitoring systems. Lastly was its usage of *tags* to easily organize metrics. OpenTSDB can be organized by its three key components – Time Series Daemons (TSD's), on device collectors, and a distributed data store as seen in Figure 4.1.

TSD's are responsible for receiving metrics from device collectors and pushing those

metrics into the distributed data store. The daemons also query the data store on behalf of the user and present the data via a web or command-line interface. In the event that a set of TSD's cannot adequately service a set of collectors the number of daemons can be increased to provide further scalability of data collection.

The collectors are scripts running on the device being monitored (in our case, virtual switches as we describe later). Collectors are responsible for retrieving and reporting time series to the TSD's, where a time series is a combination of four elements: a metric name, Unix time-stamp, a floating point or 64-bit integer value and a set of tags. The tags provide a mechanism for filtering and classifying the data in a meaningful way. For example, a metric for inOctets might additionally have the tags hostname and interface. With these tags the TSD's could present the time series as either sums of input traffic across all hosts and interfaces or a specific interface on a single host. OpenTSDB's related tcollector package [97] provides added functionality to all the user written collectors, such as on-device deduplication, where devices delay reporting metrics that have not changed since their last posted update. Preprocessing like this benefits the monitoring system by saving bandwidth and extraneous computation by the management system. In future work we wish to explore other types of preprocessing, as we believe on-switch preprocessing may be a useful avenue for finding savings in network bandwidth and data management.

Lastly, the distributed data store, HBase, is what provides the underlying scalability of the system. By utilizing Hadoop and HDFS [48, 49], HBase supports querying and writing tables with billions of rows and millions of columns.

For the purposes of our prototype cluster all of the collectors, virtual switches, TSD's and distributed data store were hosted on a single physical machine. It should be made clear that this will not be the case when the system is put to real use.

Interface and routing collectors

We implemented two prototype collectors for our initial testing on our virtual cluster, *eos_interface* and *eos_routing*, which record interface and BGP metrics, respectively. Both collectors were written to take advantage of Arista's EOS operating system. The *eos_interface* script collects interface statistics and counters by utilizing the information stored in Arista's Sysdb. At the time, the BGP routing statistics were not available via Sysdb, so we pulled them using the show commands from the command line interface (CLI). We deployed each collector as extensions onto the Arista switches by packaging each script in the required SWIX format and copying them into the switch's flash storage. Once a script is running as an EOS extension, there are mechanisms to daemonize or immortalize the extension so it is restarted in the event of failure. To ensure that the extensions persist between switch restarts they are added to the boot-extensions file. Our interface collectors ran every 15 seconds on each switch, while the routing collector ran every 60 seconds. Each metric uses the OpenTSDB tag system to provide custom filtering. This facilitates flexible examination of the system metrics, providing both macro and micro views of the system metrics. A complete list of the statistics and counters collected with their tags can be seen in Table 4.1.

Virtualized environment

To experiment with our monitoring solution, we built a virtualized cluster comprised of virtual switches and connected them with Ethernet bridges. Arista provides a slightly modified switch operating system (vEOS) image to their customers. This vEOS image can be run on a number of common hypervisors such as QEMU-KVM, VMware Player, VirtualBox and etc. To create a cluster, we use libvirt to instantiate 24 such virtual switches with Arista's vEOS image. As depicted in Figure 4.2, these switches are divided into two virtual chassis (VCs) and a group of 8 Top of Rack

Name	Tags
intf.outBroadcastPkts	host, interface
intf.outUcastPkts	host, interface
intf.inMulticastPkts	host, interface
intf.outErrors	host, interface
intf.inBroadcastPkts	host, interface
intf.outOctets	host, interface
intf.outDiscards	host, interface
intf.inOctets	host, interface
intf.inUcastPkts	host, interface
intf.inErrors	host, interface
intf.inDiscards	host, interface
intf.outMulticastPkts	host, interface
intf.inBitsRate	host, interface
intf.outBitsRate	host, interface
intf.outPktsRate	host, interface
intf.statsUpdateTime	host, interface
intf.inPktsRate	host, interface
bgp.msgrecv	host, local AS, neighbor, neighbor AS
bgp.msgsent	host, local AS, neighbor, neighbor AS
bgp.up_down	host, local AS, neighbor, neighbor AS
bgp.state	host, local AS, neighbor, neighbor AS
bgp.pfxrcd	host, local AS, neighbor, neighbor AS
bgp.summary.num_nodes	host, local AS, neighbor, neighbor AS
bgp.summary.num_routes	host, local AS, neighbor, neighbor AS

Table 4.1: Table showing the different metrics collected by the interface and routing collectors. The tags represent different methods of filtering data with respect to each metric.

switches (TORs). Each VC consists of 4 spine switches and 4 leaf switches. Each of these switches has 16 in-band data ports and 1 management port. To connect these virtual switches, Ethernet bridges are created within the host machine between any two ports where we would run a cable in real world.

Once the virtual switches and Ethernet bridges are instantiated, we can configure the virtual cluster and run routing protocols the same way as if we configure clusters with real switches. In this particular virtual cluster, we assign IP addresses on each

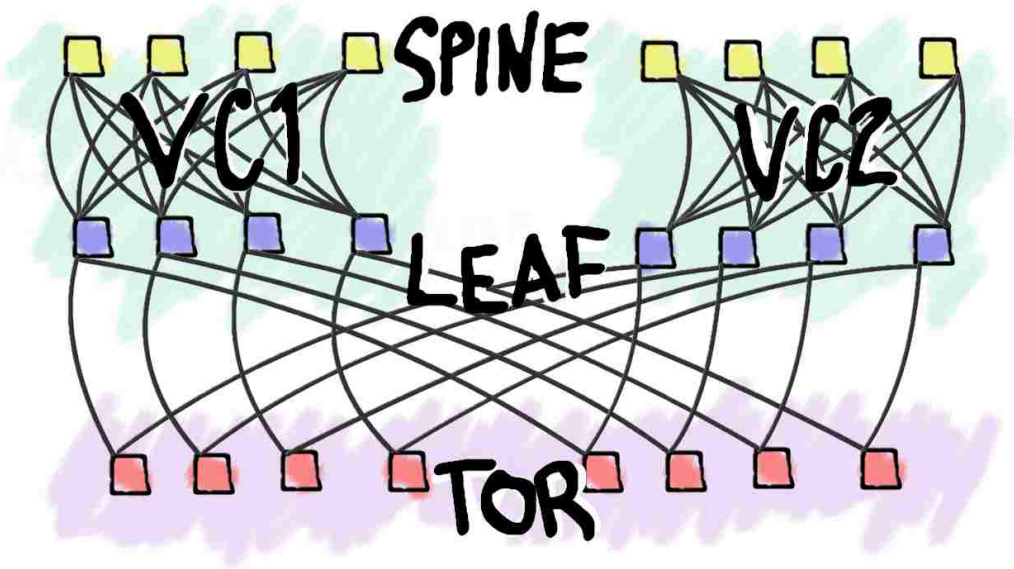

Figure 4.2: A visual representation of the 24 switches that make up the virtual cluster, divided into leaf, spine and top of the rack (TOR) nodes.

of the interfaces on the virtual switches and run stock BGP protocol among them.

The virtualized environment provides a useful testbed to prototype on-device collection without provisioning the actual hardware. To set this up on dedicated hardware, we would require 24 switches and hundreds of network cables. With this virtualized cluster we can conveniently adjust the link quality or emulate real world problems. Some of this emulation would be more difficult on physical switches where traffic is passed through the ASIC directly.

4.1.4 Results

Even though the CPU performance of virtualized switches does not necessarily reflect the expected performance of physical switches, we can inform the reader that for our cluster the CPU utilization peaked to approximately 15% on the virtual switches and these peaks lasted for under a second as collectors gathered necessary metrics.

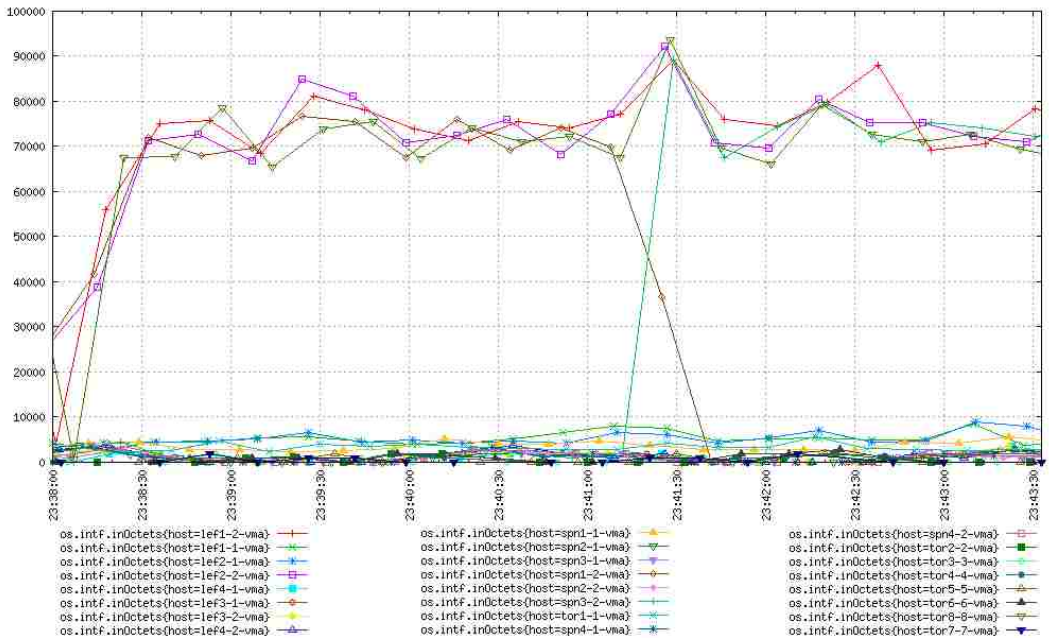

Figure 4.3: This chart represents an observed data flow from TOR1-1 to TOR8-8. This chart captures the transition from using the intermediate switch SPN1-2 to SPN3-2. The y-axis is the inOctets rate per 15 seconds.

On dedicated switches we expect the performance to be better, since our 16 core machine was hosting 24 virtual switches, the TSD and the distributed data store. The performance overhead on dedicated machines will be explored in detail in future work.

From the front-end web interface, we saw responses at a varying resolution of a milliseconds upwards to around two seconds. This was highly dependent on the amount of data points queried. This response should be greatly improved when the data store becomes distributed across multiple physical nodes. Furthermore, the TSD and collectors would have independent computational resources. In our virtual cluster, the HBase nodes were hosted on the same physical node as the TSD's and collectors. In a real system this will be distributed to provide better performance. Despite the over-subscription of virtual machines to cores we were able to retrieve

metrics at a scale and frequency not practical with current SNMP based systems. Extensive comparisons of scalability and frequency will be explored in future work.

To evaluate our prototype network, we performed a series of tests where traffic flow was generated through the network via an iperf server and client. During this time approximately 2000 metrics were collected every 15 seconds, corresponding to the combination of metrics and interfaces on each switch. We saw no trouble in keeping up with this rate, despite the over subscription of virtual machines to cores. Using the OpenTSDB monitoring solution in combination with on-switch collectors we observed a range of behaviors. In Figure 4.3 we were able to see the shift in flow as traffic pivoted from spn1-2-vma to spn3-2-vma. We present this change in flow in Figures 4.4 and 4.5. We believe this was likely due to the fact that the virtual switches lack the hardware to consistently map communication flows that an ASIC would normally provide. In additional testing we used traffic controller (tc) combined with netem to emulate packet loss on specified interfaces throughout the network. We were able to observe how this leads to a breakdown of the BGP topology, leading to an eventual loss of application traffic and we were able to observe this in real-time. If we had been using traditional SNMP approaches to monitor the network, our view of network health would only be updated every 5 minutes. With an SNMP based approach, it is possible that we might have missed this event entirely. We believe that having this rapid access to network data can facilitate better network QOS in addition to providing better support to network-aware applications. In future work we want determine what kind of analysis an on-switch monitoring solution is capable of and explore what kind of failures this system can detect that previous solutions could not.

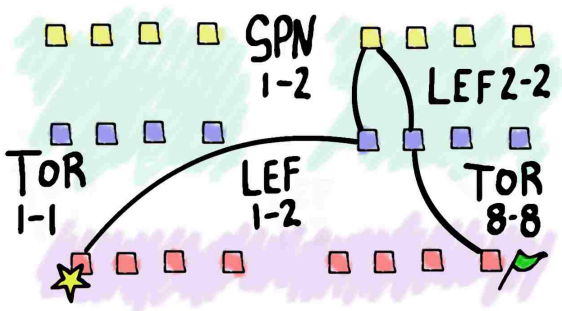

Figure 4.4: This graphic demonstrates the original flow observed from TOR1-1 to TOR8-8.

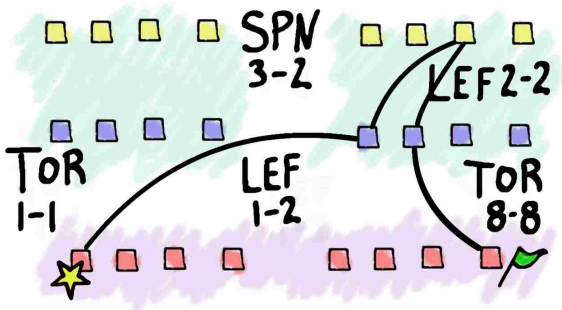

Figure 4.5: This graphic demonstrates the adjusted flow from TOR1-1 to TOR8-8.

4.1.5 Outcomes of In-network Push-based Monitoring

As large systems continue to grow, network-aware applications middle-ware and data management will be crucial to efficient system usage. We've shown that current strategies for network monitoring fall short in terms of scalability and performance. Furthermore, to facilitate dynamic and responsive networks, we've demonstrated several criteria necessary of future monitoring solutions:

- In-network collection to achieve better performance over the network, devices and monitoring managers.
- Monitoring solution situated on top of a scalable, distributed data store to support an era of large systems and big data.

- In situ processing, (e.g. data deduplication) of the data to further distribute the workload and save resources.

It is our belief that when a monitoring solution meets the above criteria it will provide five benefits to the user and the application: (1) distributed intelligence, (2) faster feedback loops, (3) smoothing of performance spikes, (4) decreased network usage and (5) ease of use by leveraging existing large scale solutions (HBase, HDFS, Hadoop in our case). As a result, our solution is more scalable and accurate with less overhead on the device, while being easier to manage.

As we move forward towards larger and more dynamic networks in the future, we want to explore enhancements to in-situ processing, so that the switch further reduces extraneous reporting of data and can reduce the burden of processing for host CPUs. With this monitoring system in place, we can explore the real-time analysis of network QOS – examining the causes of congestion, downed links and other failures.

Furthermore, features such as adaptive routing, which are becoming more common on next generation networks, perform better if they can make informed decisions about the environment they operate in. We need to evaluate the cost of monitoring overhead on physical switches and explore how we can smooth the performance spikes associated with collection. Understanding the performance cost and developing a model of data aggregation is the subject of the next portion of this dissertation.

4.2 Modeling Tree-based Data Aggregation

To provide scalability, monitoring and analysis frameworks must be built hierarchically. Internally, hierarchical collection and reduction operations can be thought of as trees, aggregating and transforming data as it progresses from leaf nodes to a root node. We refer to such networks as tree aggregation networks (TAN). For our purposes,

a TAN is a reduction tree such that a set of leaf nodes are connected to a root node directly or indirectly via a set of optional internal nodes, where these internal nodes perform partial reductions. Pertaining to the monitoring frameworks of the previous section, TANs facilitate compression/deduplication of raw data, in-situ analysis, and the scalable generation of summary statistics. These reductions are common throughout large scale distributed systems. Frameworks like MapReduce [23], MPI [87] and MRNet [106] all facilitate hierarchical data reduction. As the demands for these services grows with system size, several questions arise including "*How can we precisely model the TAN performance?*" and "*How can we design topologies that promote efficient operations for a given set of leaf nodes?*"

The primary goal of this portion of the dissertation is to introduce a new performance model for tree aggregation networks. In addition, this work makes several contributions by providing:

- A novel extension of LogP for non-uniform tree aggregation networks with or without contention
- A framework (MRNetBench) for deriving model parameters for MRNet
- A case study for a TAN performance model on a real system

When developing a performance model it is important to find a balance between the accuracy of the model and its ease of use. Our approach for a TAN performance model is one that focuses on four characteristics:

- **Accuracy** - Our model closely matches the performance seen on real machines.
- **Generality** - The parameters of our model are representative of the important underlying hardware characteristics, yet are simple enough that the model translates across multiple architectures and applications.
- **Compositional** - Our model is compositional in nature. Developed from the

ground up, its constituent parts can be dissected at any level.

Some of the characteristics above may conflict (such as Accuracy and Generality), but a well developed model will find the right compromise between these characteristics. In many ways, the canonical LogP[20] model strives for similar goals and we use it as the foundation of our TAN model, such that *latency, overhead, gap* and *the number of processing units* all are incorporated into this work. However, our work expands on the basic LogP parameters, adding *height* and *fanout* as additional parameters. Unlike the LogP model, we recognize the significance of contention that is inherent to reduction operations and account for this when calculating overhead.

We begin by providing an overview of background and related work. In Section 4.2.2, we describe the model in detail and discuss the significance each parameter. Afterwards, we describe how the parameter values are derived in Section 4.2.3. This is followed by Section 4.2.4, in which we present a comparison of the accuracy of our model and the LogP model. Finally, we provide a summary of our contributions to TAN modeling in Section 4.2.5 .

4.2.1 Background and Related Work

In order to simplify, and summarize complex computational systems, we rely on models. Historically, both the PRAM [36] and BSP [129] models of parallel computation were popular. However for the last twenty years the LogP [20] model has been the de facto standard. In 1993, LogP was introduced as a framework for building accurate models of parallel computations. The work focused on the most significant parameters that could be used to deliver a model that could reveal important bottlenecks without making the analysis of interesting problems intractable. Four parameters were introduced in the original model: *latency*, *overhead*, *gap* and *processing units*. Latency describes the time it takes the first bit to move across the wire from one node to

another. Overhead is the time it takes the NIC and CPU to process the packet, while Gap is the reciprocal of bandwidth – essentially, the amount of time it takes for all the data to move onto the wire. P is simply the number of processing units. These four simple parameters provide a concise yet powerful representation of network performance. As described in [20]:

> [A] parallel model [should] be realistic, yet simple enough to be used to design algorithms that work predictably well over a wide range of machines. The model should allow the algorithm designer to address key performance issues without specifying unnecessary detail. It should allow machine designers to give a concise performance summary of their machine against which algorithms can be evaluated.

As an example, the creators of LogP demonstrated how the model could facilitate the design of optimal broadcast trees. This model has been leveraged many times since, including by Goehner et al.[38] to develop an algorithm for optimal process launching in HPC resource management systems. Since its introduction, LogP has become the foundation for many other models of parallel computation. Several notable extensions of the LogP model have been introduced throughout later years. In the LogGP model [5], Alexandrov et al. introduce an additional parameter (G) which better models long messages. This has been followed by other works [76, 66] that examine how as message size changes, communication protocols may switch to create different performance classifications.

The LoGPC [86] and LoPC [37] extensions both incorporate the effects of contention, though LoGPC focuses on network contention, while LoPC incorporates contention in the message processing facilities. The LoPC model includes the effect of message processing overhead, however it assumes an underlying queue handles the messages. With this assumption, Little's Result, Bard's Approximation and Mean Value Analysis are used to approximate queue length and time to handle a message. Our model

differs from LoPC by making no assumption about the underlying protocols or NIC transfer method.

More recently Hoefler, et al. [61] introduced the LogGOP model which has two important distinctions from the previous models. The first is that the model keeps CPU overhead and network gap separate, to model potential communication/computation overlap. The other extension of LogGOP is that overhead is measured as a per-byte cost rather than a constant cost.

The LogP model and its derivatives are used extensively in HPC modeling and simulation including the work done as a part of this dissertation. The Merlin network simulator in Chapter 3 implicitly relies on these models of performance. In this chapter, we introduce an extension of the LogP model that targets tree-based aggregation networks (TAN). These aggregation networks are representative of the data-reduction operations typical of large distributed systems, such as monitoring power and system statistics and performing reductions of application data.

4.2.2 Model

Architectural Assumptions

Our TAN model represents systems which have a one to one mapping of physical nodes to tree processes. While a single TAN process per node is generally conservative of current architectures, extending our model to multi-core/multi-process architectures requires collecting model parameters in two groups, one for inter-node modeling and one for intra-node modeling. Other than this adjustment, the model would remain mostly unchanged. We reserve such an extension for future work. Another constraint of our TANs is that a process does not fulfill multiple roles within the topology. For example, a back-end or leaf process may not transition to a front-end or root process after pushing its data into the network. While there exist models and workload

optimizations for this type of network [77], it is beyond the scope of our current work. Furthermore, there are scenarios, such as a pipelined workflows, where this assumption of static roles makes the most sense. A final assumption of our model is that the architecture is uniform across the nodes of a system. For example, if one node in the system has a message co-processor, all the nodes in the system have a message co-processor. This enables our model to derive the performance characteristics for the entire system by sampling the performance of a subset. In most high performance systems, this is a reasonable expectation.

Despite these constraints, our model relaxes several assumptions of previous models. We do not make any assumption about the topology of the TAN, so that our model handles both simple and complex trees including those of non-uniform fan-out. Additionally our model is able to capture contention in message processing overhead as well as in the interconnect. Another feature of the TAN model is that it does not make any assumptions of the underlying NIC transfer method, whereas several previous models are forced to assume an interrupt driven, RDMA or polling mechanism.

Model Parameters

These four parameters are included in most LogP based performance models, though they have been slightly modified to better represent TANs and the effects of contention.

L**:** latency is measured as an median of the time or cycles taken to send a message from a source processor to target processor.

$o(x)$**:** message processing overhead is a period of time which a processor is working to transmit or receive a message with respect to the fanout of that node (x). In the absence of a dedicated communication processor, work cannot be done by the processor during this transmission or receipt.

g**:** the gap is defined as the minimum interval between sending or receiving consecutive messages by a processor. Gap is the reciprocal of bandwidth, such that $g =$ 1/bandwidth.

P**:** is defined in our work as the number of processes. This is not an explicit parameter of our model but is incorporated by taking the height and fanout of the tree into account.

In the original LogP paper [20], latency is defined as the upper bound on *latency*, or *delay*, incurred in communicating a message containing a word or (or small number of words) from its source module to its target module. We differ from LogP by utilizing the average rather than an absolute upper bound. We found on real world machines, that the upper bound did not properly represent typical performance. Real world techniques for measuring latency, are often effected by noise such as congestion, changes in route, connection establishment protocols, hardware affinity, or DNS look-ups. While network jitter is visible in our work, we save exploring it in depth for future work. Particularly with regards to reduction operations, we found network jitter to be more important at lower fanout and less important at higher fanout. For the systems evaluated, at low fanout network jitter is responsible for an error of $\pm$ 4 percent. At high fanout, where communication overhead is greater, this error becomes an order of magnitude less significant.

The authors of LogP state that they expect overhead costs will disappear as architectures improve. While this has become true of some current architectures [56] and workloads, in our environment message processing overhead was the largest contributor to performance costs. Furthermore, we include results in Figure 4.6, showing that while modern communications methods may have little overhead in the absence of contention, message processing overhead grows to become the dominating factor in communication performance as the number of participants increases. Specifically, Figure 4.6 shows the time it takes to complete N-to-1 communication operations over

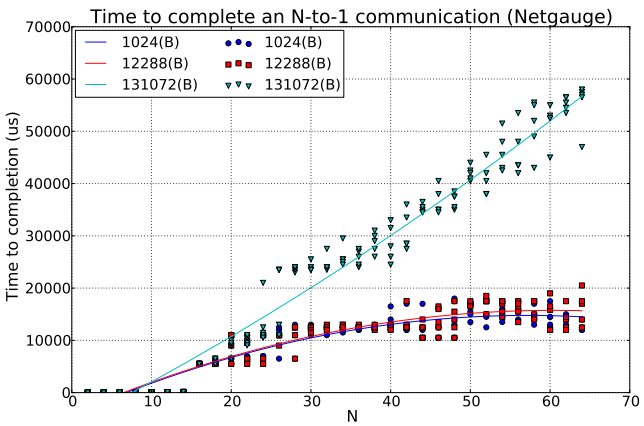

Figure 4.6: We compare the cost of N-to-1 communication operations for varying packet sizes over Infiniband. It is clear that contention becomes the bottle-neck in communication costs as the number of participants increase.

Infiniband, for three varying packet size of 1KB, 12KB, and 131KB as we increase the value of N. Five runs of these experiments were performed using Netgauge [58] on the Cab cluster (whose technical specifications are outlined below). Because the latency in an N-to-1 operation occurs in parallel, we can attribute the increase in cost to message processing overhead contention. This figure shows how performance cost is minuscule when reduction operation contains a low number of participants, but as the number of participants grow, message processing overhead becomes a non-trivial cost.

We have elected to ignore network gap in Section 4.2.3, since overhead is a significantly larger cost. If a user wishes to distinguish between gap and overhead, this could be accomplished in a manner similar to previous LogP models. Another difference between our parameters and that of LogP is that we model overhead as a function of fanout (x). This enables us to model the effect of contention, whereas the original LogP model cannot. Similar to LogP, we do not explore how varying message size affects the performance of our model. If we elected to do so, extending the model in a manner similar to [5, 76, 66] would be possible. Additionally, by keeping the

message sizes at 1 KB, we stay well under the maximum throughput of the system.

We leverage the structural properties of TANs by incorporating two additional parameters to the LogP model:

y: the height is denoted by an implicit variable y, which represents number of hops needed to traverse the tree from the farthest leaf to the root node.

x: the fanout is denoted by the variable x which is a measure of the number of children directly linked to a parent node.

Height is implicitly built into our model by its recursive structure. This is discussed further in in Section 4.2.2. Fanout is explicitly built into the TAN model, denoted as (x) at each level of the tree. In later sections it will become apparent how we derive the function $o(x)$.

The TAN Model

The composite model is given as follows:

$$\begin{aligned} T_i &= L + o(x+1) + g + \max(T_{child}), \\ &\quad (i \in N, \quad i \neq leaf) \\ T_{leaf} &= C \end{aligned} \tag{4.1}$$

In this model T_i is the time to process the wave up to node $i \in N, i \neq leaf$. T_{child} represents the time to process the wave for a child of i. T_{leaf} represents our base case in our recursive model, with the parameter C representing constant "one-time" costs. These are costs that only occur once for a given tree, independent of height or fanout. This model is applied recursively to give an accurate estimation of performance.

Pipelining, or steady state throughput simplifies the model in that one-time costs can be ignored and we can remove the recursive structure from the model above.

Assuming a sufficiently large number of waves, the height of the tree is amortized and the model becomes:

$$T = L + \max(o_i(x+1)) + g,\ i\ \in\ N \tag{4.2}$$

To determine the time taken to process w waves (for a sufficiently large w), we simply multiply our model by w. For the scope of this work we evaluate non-pipelined workflows and include the pipelined model variant for completeness' sake.

4.2.3 Determining parameter values of the model

To test the performance prediction of the TAN model, we generate model parameter values for latency, communication overhead, and application overhead on the Cab cluster at Lawrence Livermore National Labs. We do this through a 3 stage processes where we first measure latency using Netgauge framework. Second, we model the performance of our communication framework, the Multicast Reduction Network (MRNet) [106] , using an application we developed, called MRNetBench. The following subsections provide an in depth discussion of the methodology and reasoning used to generate our model parameter values. Throughout the course of this modeling work, unless stated explicitly, the unit of measurement is seconds. We include all the parameter values from our experiments in Table 4.2.

The Cab cluster at Lawrence Livermore is a 1,200 node cluster, where each node consists of a 16-core Intel Xeon E5-2670. Each node has 32 GB of memory and runs the TOSS 2.0 operating system. The entire cluster is connected via a InfiniBand QDR (QLogic) interconnect. The scale of our experiments is limited to 256 nodes, the maximum allocation allowed without special reservation or administrative processes for releasing acquired data. While larger trees are interesting, they consist of compositions of smaller trees. For instance, two million nodes leaf nodes might connected by three levels of internal nodes with 128 fanout. While we leave large scale runs to future

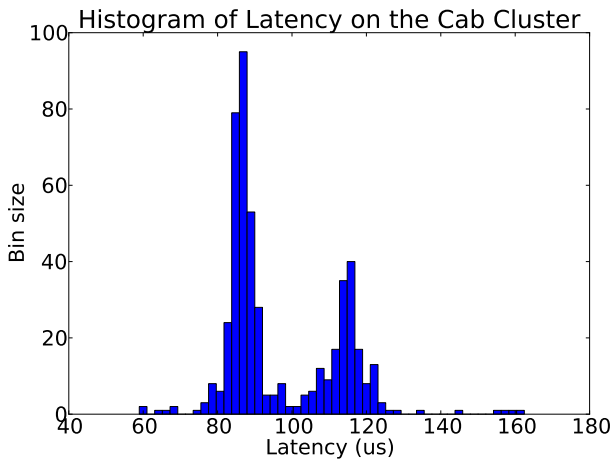

Figure 4.7: Histogram of 500 latency samples using two hosts on the Cab Cluster. A single outlier of 672 microseconds was removed for the presentation of this figure.

work, our model is a constructive one, such that there are no technical hurdles to applying the model to these scales.

Determining latency

To generate the latency values for the TAN model, we perform a small set of experiments using Netgauge. Netgauge is an open source framework for implementing network benchmarks. The Netgauge framework offers a wide variety of communication patterns and protocols and has been used in a large number of publications for determining network performance. We calculate latency characteristics of the system by selecting 2 nodes at random. For this pair of nodes, we run Netgauge using the *distt* mode and TCP over IB protocol. We use the TCP over IB protocol because this is representative of the applications we use for validation. Netgauge samples the round trip time 500 times between the two nodes. In Figure 4.7, a histogram of the latency shows that the distribution is clearly bimodal. One possible reason for this bimodal behavior could be the selection of the core/socket to process incoming and

outgoing packets relative to the core utilized by the application. This behavior has been observed in previous work [51, 100] and is further supported by the fact that Cab contains two sockets. From this sample the maximum, average, minimum and standard deviation for latency was 0.672, 0.098, 0.059, and 0.030 ms, respectively. Even though reduction operations rely on the slowest participant for performance, 0.672 ms was not representative of typical network performance. Because of this, we assign L, in our model (and the LogP model), the value representing the average latency. On Cab this value was 0.098 ms.

For our set of experiments the gap of the system is ignored. We can ignore the cost of g in our system because the cost of communication overhead is significantly more expensive, especially as contention increases with fanout of the tree.

Determining communication overhead

After the value of latency is determined we must determine the value of communications overhead. For the purposes of this research, we will be using MRNet as our communications framework. MRNet is a software overlay network which provides multicast and reduction communications for parallel and distributed systems. To evaluate the performance of MRnet we developed a MRNet benchmarking application – MRNetBench. MRNetBench was designed to enable researchers to test aspects of performance for a wide variety of TAN topologies and filters. The application allows the user to specify the packet size, filter duration, frequency, and synchronization mode. Filter duration is approximated by performing k iterations of matrix multiplication repeatedly until the specified duration has been reached. For our experiments we utilize an empty MRNetBench filter so that we can isolate the communication overhead and latency costs. Additionally, MRNetBench provides 3 modes of operation: *synchronous*, *timed*, and *number of waves*. In synchronous mode, the root and leaf processes are synchronized using NTP so that leaf processes

send a communication wave to the root process simultaneously. In the timed mode there is an initial synchronization, but subsequent waves are sent without explicit synchronization attempts. Waves continue to be pushed through the system according to the specified frequency for the duration of the timer. In number of waves mode, the root node requests the leaves to initiate an aggregate communication and records the round trip time required. MRNetBench is a relatively simple application in practice, but it is an example of the type of framework necessary for deriving parameters for LogP based models.

When measuring the time it takes to complete an aggregate communication, there are two possible approaches. The first approach is to use properly synchronized clocks to schedule the communication operation in the near future and then take the difference from the end and start times (MRNetBench *synchronous* mode). The advantage of this method is that the leaf nodes initiate the communication simultaneously. The downside to this approach is that it relies heavily on the synchronization of each participant's clocks. The second approach is to initiate the aggregation from the root node and record the round trip time (MRNetBench *number of waves* mode). This approach benefits from relying on only a single clock, but it includes the additional latency and communication overhead costs which must be subtracted. We tried both methods and found that the method utilizing round trip time and relying on a single clock had significantly less noise. The noise from the synchronized approach was so significant that even with the extra latency and overhead costs, the round trip approach resulted in a faster recorded time. For these reasons, we measure a communication operation using the round trip time approach. In order to account for the extra latency and overhead costs associated with the round trip approach, we make the assumption the scatter and reduction operation are symmetrical and divide the round trip time result by two.

The cost of communication overhead is determined through two sets of experiments:

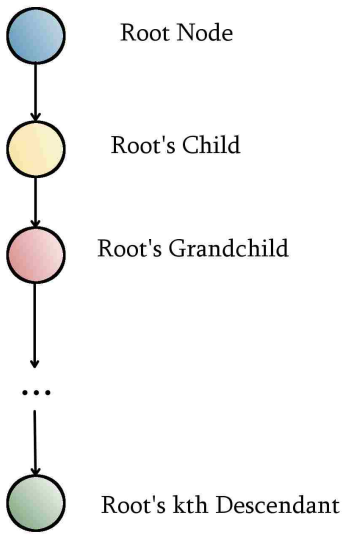

Figure 4.8: Illustration of a "chain" topology.

chain and N-to-1. These runs measure the time to process a wave for an increasing height and fanout, respectively. The purpose of the chain experiments is to derive *per-level* and any *one-time* costs in the model, y and C respectively. N-to-1 experiments utilize data from the chain experiments and then perform regression analysis to derive remaining coefficients for the communications overhead function. In both runs, leaf nodes send 11 waves of reduction operations. We discard the first wave because the performance is an order of magnitude worse due to bootstrapping costs that are not a part of this model. The following two subsections describe each experiment and how we combine them into a communications overhead function.

The Chain Experiment

In this experiment, we use MRNet to measure the time it takes to process a wave as we increase tree depth. The tree is a chain topology, such that each node has a single child. An example of this topology is seen in Figure 4.8. We limited the height to

32, since a binary tree of height 30 represents over 1 billion processes. In practice, a TAN topology would likely not exceed a height of 30 on even the largest systems. In MRNet we set timers to record the time a communication operation begins at the leaf node and ends at the root node. This is collected 10 times at each depth from a height of 1, to a height of 32. After we collect this information, we use our previously determined value for L and multiply it by the depth of the tree. This is essentially the number of hops times the latency. We then subtract this value from each respective data point. Because there are two overhead operations at each level (one send and one receive), we divide each value by 2 so that we are left with just the communication overhead cost. By performing a simple regression analysis on the data of Figure 4.9, we are able to derive a per-level overhead, and any one-time costs. This analysis generates the value of b in the function $y = a + bx$. Our simple linear regression provided an estimate of $y = 4.93 \times 10^{-5} x$ seconds and a value of 0 for a, with a coefficient of determination $R^2 = 0.89$ for the linear regression. We believe this value of R^2 is sufficient for our model, but in future work we will explore more accurate methods of representing latency and system noise. If the coefficient a had mapped to a value greater than 0, we would have assigned this to the *one-time* cost C in the model. We label the value which we derived for b as a *per-level* communication overhead, and incorporate it the sub-section that follows.

The N-to-1 Experiment

The goal of this second experiment is to isolate and extract the communication overhead with respect to the fanout (x) of a root node. This is accomplished by measuring the time it takes to process a wave in a N-to-1 topology as we increase the value of N. An N-to-1 topology is a tree which consists of only a root and leaf nodes such that the leaves are directly connected to the tree root. An example can be seen in Figure 4.10. First we subtract the latency and a single *per-level* overhead cost

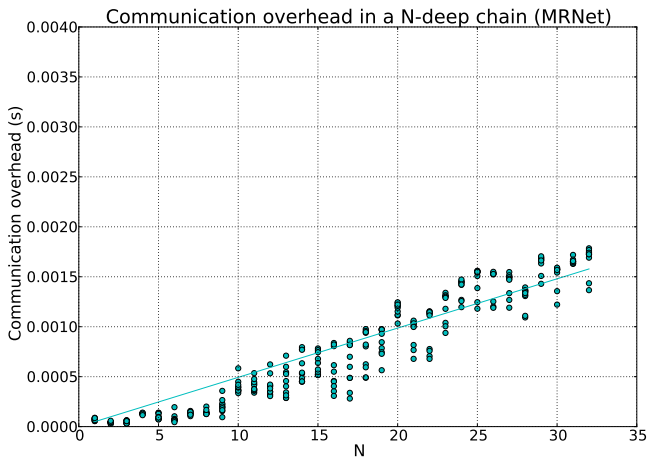

Figure 4.9: Time taken to process a single wave in a chain topology over an increasing number of internal tree processes.

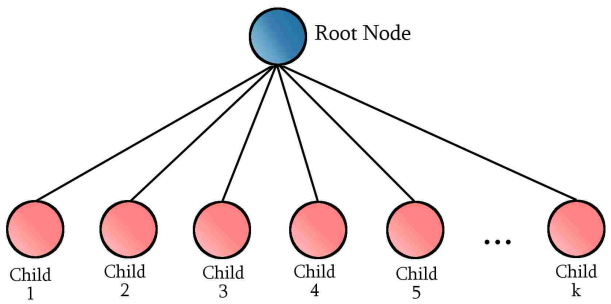

Figure 4.10: Illustration of a "N-to-1" topology.

(leaf node send) from the data. Then, similar to the chain experiment, we perform a regression analysis on data seen in Figure 4.11. When comparing the results to a linear, logarithmic and quadratic fit, the best fit for this data was a quadratic fit. By finding the best fit for the equation, $y = bx + cx^2$, we are simply deriving the cost of overhead relative to the fanout of the topology. the value derived for the coefficients b and c are 7.83×10^{-7} and 1.57×10^{-7} seconds, respectively. The coefficient of determination, R^2 for this experiment was 0.90. Following this second analysis, we have all the coefficients necessary to fully model the communication overhead of

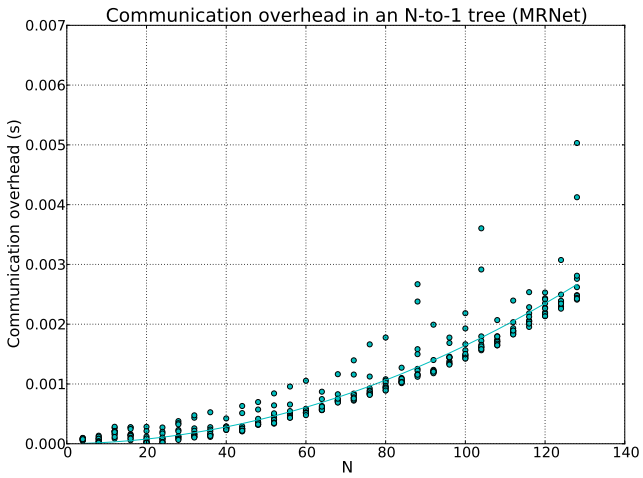

Figure 4.11: Communication overhead (MRNet) for a single wave in a N-to-1 topology over an increasing number of leaf processes.

Table 4.2: Model parameter values (s)

L	9.8×10^{-5}
o_{MRNet}	$4.93\times10^{-5} + 7.83\times10^{-7}(x) + 1.57\times10^{-7}(x^2)$
g	N/A
C	N/A

our model. The values derived for these coefficients and the *one time* overhead, are all combined in Table 4.2 as o_{MRNet}. If the data from the N-to-1 experiment had been linear, logarithmic or exponential, the same process would apply, with the least significant coefficient being populated by values derived for latency and per-level overhead.

In conclusion to this section we include all the parameters of our TAN model into Table 4.2. These parameter values are specific to the Cab cluster, such that if we desired to model an additional system, we would need to derive that system's parameter values as well.

4.2.4 Complex Topology Validation

In order to evaluate the performance of our model, we compare the accuracy of the TAN model and the original LogP model for the MRNet framework. The purpose of these experiments is to show the accuracy of the TAN model for predicting communication performance and to evaluate when the TAN model may be beneficial in lieu of the traditional LogP approach. The only difference between the parameters values of the LogP Model and the TAN model in our experiments is the calculation of message processing overhead. In the case of LogP, we assign the overhead the value derived from the chain experiments, whereas in the TAN model, overhead is represented by a more complex function. In these validation experiments, we tested the performance of two sets of topologies. The first set of topologies consist of trees with 32 leaf nodes and a varying internal structure. Specifically, the four trees in this set are: an unbalanced two-level tree, such that the root node has 6 children, of which have either 5 or 6 leaf nodes as children, a binomial tree, a binary tree, and a N-to-1 tree.

The second set of topologies consist of trees with 128 leaf nodes and an internal structure similar to the first set. Specifically, the four trees in this set are: an unbalanced two-level tree, such that the root node has 11 children, of which have either 11 or 12 leaf nodes as children. a binomial tree, a binary tree, and a N-to-1 tree.

These topologies are representative of tall/thin (binary), short/fat (n-to-1), balanced (2-level), and skewed (binomial) configurations. Our choice in this variety demonstrates the models applicability to a range of configurations. In both sets of experiments we take the average of 10 runs for each topology to represent the observed performance.

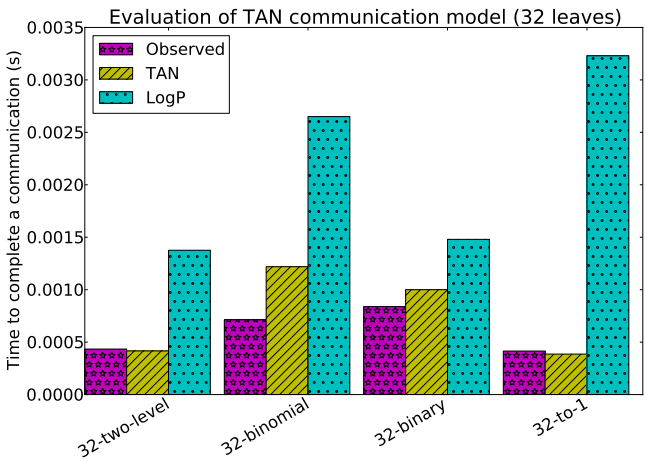

Figure 4.12: Time to process a single communication wave for 4 varying topologies, each with 32 leaf nodes.

Evaluation of Communication Modeling.

In Figure 4.12 and Figure 4.13, we compare performance of complex topologies without considering application overhead. In Figure 4.12, the best results for the TAN model were achieved predicting shallow trees. This is not surprising, given the relatively high value for the coefficient of determination for the experiments that generated the values for message processing overhead. LogP's best case match of observed performance was for the binary tree. This is because a binary tree is the topology most similar to a chain topology and it is the topology that experiences the least amount of contention for message processing. That being said, the TAN model provides a closer approximation of the observed performance in every single case. Of the TANs tested with 32 leaves, the best performing were the two level and the 32-to-1 topologies.

In the experiments with 128 leaf nodes we begin to see the cases where the TAN model really shines. Because contention becomes more significant as fanout increases,

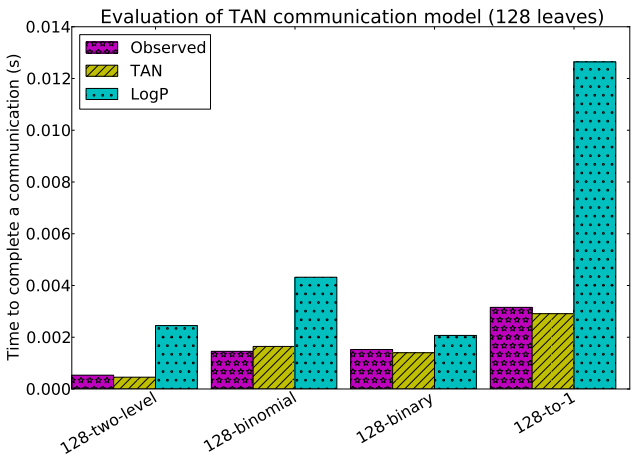

Figure 4.13: Time to process a single communication wave for 4 varying topologies, each with 128 leaf nodes.

the TAN model provides a better estimate than LogP and matches the observed performance with greater precision than that of the experiments involving 32 leaf nodes. Of the TANs tested with 128 leaves, the two level topology significantly outperforms all other topologies. These experiments validate the accuracy of the TAN model and show how it can facilitate the design of efficient topologies.

In the context of analyzing application performance, the power of the LogP model is the flexibility it provides, so that communication overheads may be separated from application overheads. The TAN model still provides this important functionality, but additionally allows the performance to be further deconstructed, so that topology effectiveness can be evaluated on a per-level basis even for topologies with non-uniform branching. This flexibility facilitates efficient topology design, scheduling of workflows, and the resolution of localized bottlenecks.

4.2.5 Outcomes of the TAN Model

In this part of the dissertation we have extended the LogP model by introducing the TAN model. This model considers the affects that contention may have on performance and provides a fine grained approach for predicting the performance of reduction operations in tree aggregation networks. In order to test our approach we developed a benchmarking tool (MRNetBench) for the MRNet framework. We then used this tool to show how model parameters could be extracted from a simple set of two experiments. Our model shows a significant improvement in performance prediction versus the standard LogP model for all of the topologies tested. Furthermore, this work provides insight about the performance of reduction operations at a scale of 256 nodes of the Cab cluster at Lawrence Livermore National Labs.

4.3 Chapter Conclusions

In this chapter we asked, *"How can we improve the responsiveness and scalability of network monitoring?"* and *"How can we better model the cost of hierarchical data aggregation?"* These two questions are central to developing effective and scalable monitoring solutions that facilitate adaptive solutions. We demonstrated that traditional approaches to network monitoring were cumbersome and implemented a in-network push-based agent to improve responsiveness. Placing the monitoring agents on the devices being monitored (i.e. the switch) relieves the application nodes and reduces the network traffic associated with monitoring, due to in-situ processing techniques like deduplication. Furthermore, we extended the LogP model, so that it could better represent the cost of large-scale data aggregation typical of HPC monitoring frameworks. We evaluated the effectiveness of our model at scale and showed significant improvements to accuracy of the base model. In particular, we demonstrated that our model led to a better selection of tree in a hierarchical

aggregation. As we move toward the Exascale era, the contributions of this chapter will assist in the development of dynamic solutions that adapt to their environment. However, in order to judge the impact of a dynamic solution, our understanding of large-scale monitoring needs to be grounded in reality. Information is not free and as a community we must make sure we include the "hidden" cost of data collection, storage and dissemination.

Chapter 5

Conclusion

In this dissertation, we set out to improve the performance and power of HPC systems with a focus on networks, through new models, benchmarks and monitoring techniques. We explored the characteristics and hidden costs of emerging techniques in HPC communication, providing insights, solutions, and a suite of benchmarks for evaluating performance (Chapter 2). Later, in Chapter 3, we evaluated HPC systems from a macro level, developing new approaches for visualizing the performance of Exascale networks. We took our simulations and applied empirical measurements of power as well as existing models in literature, to provide conservative and optimistic projections of network power savings on future systems. These results contribute to the field's understanding of how to build high performance, power efficient large scale networks. In order to facilitate high performance and adaptive networks of the future, we examined the cost of monitoring a large-scale network, developing on-switch, push-based approaches to scalable monitoring as well as improved models for data aggregation (Chapter 4).

5.1 Contributions

While this dissertation includes a large number of contributions that are summarized in the individual chapters, there are several major contributions of this work:

1. Evaluate and characterize one-sided communication in HPC systems, introducing the concept of Network-induced Memory Contention with three candidate solutions.

2. Characterize power and performance trade-offs in large-scale dragonfly networks, with new methods for fine grained analysis of network utilization.

3. Develop on-switch, push based network monitoring agent as part of a scalable network monitoring solution, with new performance models of data aggregation.

These contributions provide a breadth and depth of a large body of work related to the power and performance of HPC networks. The impact is evident in five peer-reviewed publications [41, 44, 43, 42, 30]. Beyond publications, this work has had direct impact on publicly available software [114, 89, 94, 103], resulting in numerous releases and improvements. As we continue to expand our understanding of HPC systems, it is crucial to have a firm grasp of trends in HPC communications and an understanding of how they impact node-level performance. These models for node level performance must be integrated into a macro level models of network topology, routing and application performance/power constraints. Lastly, in order to implement the dynamic and adaptive solutions that are becoming increasingly common, we must have an understanding of the performance cost associated with network monitoring. Too often these costs remain undiscussed in literature, though they are critical in determining the capabilities of the system. As such, this dissertation sets the stage for a rich body of future work. And though it closes the door on several open research questions, more importantly, it provides the necessary foundation to move forward.

Appendices

Appendix A

Supplemental Data of Chapter 2

A.1 Feature Importance of PMCs for CNS and Varying Prediction Criteria

Below are tables containing the feature importance values referenced in the work on detecting NiMC and predicting its impact on application performance. These tables contain the coefficient of determination, Out-of-Bag scores (OOB), and the feature importances for the listed features in binary classification or regression using the Random Forests package of scikit-learn in Python. There are four tables for each feature set, representing predictions of CPU-time (with and without the ib_bw feature), predictions of the volume of RDMA traffic, and binary classification of whether or not the workload is experiencing NiMC.

Table A.1: Random forests predicting CPU time of workload in the presence of NiMC. (cns, feature set:1)

Coefficient of Det.:0.9975						
Out-of-Bag Score:0.9816						
Features	ib_bw	l1_icm	icy	l1_dcm	tlb_im	tlb_dm
Importance	0.0031	0.0844	0.1549	0.1628	0.1670	0.4278

Table A.2: Random forests predicting CPU time of workload in the presence of NiMC. (cns, feature set:1)

Coefficient of Det.:0.9974					
Out-of-Bag Score:0.9807					
Features	l1_icm	icy	l1_dcm	tlb_im	tlb_dm
Importance	0.0853	0.1551	0.1636	0.1674	0.4285

Table A.3: Random forests predicting volume of RDMA traffic on target node. (cns, feature set:1)

Coefficient of Det.:0.9142					
Out-of-Bag Score:0.3745					
Features	tlb_dm	tlb_im	l1_dcm	l1_icm	icy
Importance	0.1129	0.1275	0.1357	0.1737	0.4501

Table A.4: Random forests binary classification to detect NiMC. (cns, feature set:1)

Coefficient of Det.:1.0					
Out-of-Bag Score:0.7406					
Features	tlb_im	tlb_dm	l1_dcm	l1_icm	icy
Importance	0.1393	0.1450	0.1525	0.1623	0.4008

Table A.5: Random forests predicting CPU time of workload in the presence of NiMC. (cns, feature set:2)

Coefficient of Det.:0.9971								
Out-of-Bag Score:0.9789								
Features	ib_bw	l2_tcm/l2_tca	l2_dcm/l2_tca	l2_icm	l2_tca	l2_tcm	l2_dcm	l2_ich	l2_dch
Importance	0.0018	0.0040	0.0059	0.0322	0.0330	0.0404	0.0607	0.2780	0.5441

Table A.6: Random forests predicting CPU time of workload in the presence of NiMC. (cns, feature set:2)

Coefficient of Det.:0.997								
Out-of-Bag Score:0.9782								
Features	l2_tcm/l2_tca	l2_dcm/l2_tca	l2_tca	l2_tcm	l2_icm	l2_dcm	l2_ich	l2_dch
Importance	0.0042	0.0061	0.0332	0.0405	0.0420	0.0513	0.2782	0.5445

Table A.7: Random forests predicting volume of RDMA traffic on target node. (cns, feature set:2)

Coefficient of Det.:0.9164								
Out-of-Bag Score:0.3958								
Features	l2_tcm	l2_tca	l2_dcm	l2_icm	l2_ich	l2_dcm/l2_tca	l2_dch	l2_tcm/l2_tca
Importance	0.0536	0.0587	0.0613	0.0705	0.0966	0.1287	0.1311	0.3994

Table A.8: Random forests binary classification to detect NiMC. (cns, feature set:2)

Coefficient of Det.:1.0							
Out-of-Bag Score:0.7467							
Features	l2_tca	l2_ich	l2_icm	l2_tcm	l2_dcm	l2_dch	l2_dcm/l2_tca	l2_tcm/l2_tca
Importance	0.0847	0.0882	0.0903	0.0998	0.1074	0.1569	0.1580	0.2147

Table A.9: Random forests predicting CPU time of workload in the presence of NiMC. (cns, feature set:3)

Coefficient of Det.:0.9957					
Out-of-Bag Score:0.9687					
Features	ib_bw	l3_tca	l3_ica	l3_dca	l3_tcm
Importance	0.0045	0.1109	0.1915	0.2464	0.4467

Table A.10: Random forests predicting CPU time of workload in the presence of NiMC. (cns, feature set:3)

Coefficient of Det.:0.9954				
Out-of-Bag Score:0.9664				
Features	l3_tca	l3_ica	l3_dca	l3_tcm
Importance	0.1216	0.1927	0.2380	0.4477

Table A.11: Random forests predicting volume of RDMA traffic on target node. (cns, feature set:3)

Coefficient of Det.:0.9181				
Out-of-Bag Score:0.4071				
Features	l3_tca	l3_ica	l3_dca	l3_tcm
Importance	0.1306	0.1697	0.2206	0.4791

Table A.12: Random forests binary classification to detect NiMC. (cns, feature set:3)

Coefficient of Det.:1.0				
Out-of-Bag Score:0.7423				
Features	l3_ica	l3_tca	l3_dca	l3_tcm
Importance	0.1619	0.1782	0.2292	0.4306

Table A.13: Random forests predicting CPU time of workload in the presence of NiMC. (hpccg, feature set:1)

Coefficient of Det.:0.9969						
Out-of-Bag Score:0.9775						
Features	tlb_dm	tlb_im	l1_icm	icy	l1_dcm	ib_bw
Importance	0.0043	0.0047	0.0072	0.1183	0.1303	0.7352

Table A.14: Random forests predicting CPU time of workload in the presence of NiMC. (hpccg, feature set:1)

Coefficient of Det.:0.9953					
Out-of-Bag Score:0.9659					
Features	tlb_im	tlb_dm	l1_icm	icy	l1_dcm
Importance	0.0074	0.0096	0.0346	0.1053	0.8431

A.2 Feature Importance of PMCs for HPCCG and Varying Prediction Criteria

Below are tables containing the feature importance values referenced in the work on detecting NiMC and predicting its impact on application performance. These tables contain the coefficient of determination, Out-of-Bag scores (OOB), and the feature importances for the listed features in binary classification or regression using the Random Forests package of scikit-learn in Python. There are four tables for each feature set, representing predictions of CPU-time (with and without the ib_bw feature), predictions of the volume of RDMA traffic, and binary classification of whether or not the workload is experiencing NiMC.

Table A.15: Random forests predicting volume of RDMA traffic on target node. (hpccg, feature set:1)

Coefficient of Det.:0.9989					
Out-of-Bag Score:0.9923					
Features	tlb_im	tlb_dm	icy	l1_icm	l1_dcm
Importance	0.0055	0.0104	0.0110	0.0891	0.8839

Table A.16: Random forests binary classification to detect NiMC. (hpccg, feature set:1)

Coefficient of Det.:1.0					
Out-of-Bag Score:0.9983					
Features	l1_icm	tlb_im	icy	tlb_dm	l1_dcm
Importance	0.0431	0.0575	0.1573	0.1963	0.5458

Table A.17: Random forests predicting CPU time of workload in the presence of NiMC. (hpccg, feature set:2)

Coefficient of Det.:0.998									
Out-of-Bag Score:0.9854									
Features	l2_dcm	l2_tcm	l2_icm	l2_ich	l2_tca	l2_tcm/l2_tca	l2_dcm/l2_tca	l2_dch	ib_bw
Importance	0.0012	0.0013	0.0036	0.0101	0.0206	0.0624	0.0657	0.0972	0.7378

Table A.18: Random forests predicting CPU time of workload in the presence of NiMC. (hpccg, feature set:2)

Coefficient of Det.:0.9965								
Out-of-Bag Score:0.9747								
Features	l2_icm	l2_tcm/l2_tca	l2_dcm/l2_tca	l2_tca	l2_tcm	l2_dcm	l2_dch	l2_ich
Importance	0.0085	0.0400	0.0408	0.0417	0.0583	0.1014	0.1416	0.5678

Table A.19: Random forests predicting volume of RDMA traffic on target node. (hpccg, feature set:2)

Coefficient of Det.:0.9993								
Out-of-Bag Score:0.9945								
Features	l2_dch	l2_icm	l2_tca	l2_tcm/l2_tca	l2_dcm/l2_tca	l2_ich	l2_tcm	l2_dcm
Importance	0.0006	0.0008	0.0014	0.0019	0.0019	0.1198	0.1663	0.7072

Table A.20: Random forests binary classification to detect NiMC. (hpccg, feature set:2)

Coefficient of Det.:1.0								
Out-of-Bag Score:0.9989								
Features	l2_dcm/l2_tca	l2_tcm/l2_tca	l2_icm	l2_ich	l2_dch	l2_tcm	l2_tca	l2_dcm
Importance	0.0295	0.0316	0.0527	0.0927	0.1296	0.2090	0.2113	0.2436

Table A.21: Random forests predicting CPU time of workload in the presence of NiMC. (hpccg, feature set:3)

Coefficient of Det.:0.9973					
Out-of-Bag Score:0.9803					
Features	l3_tca	l3_dca	l3_tcm	l3_ica	ib_bw
Importance	0.0043	0.0044	0.0543	0.2461	0.6908

Table A.22: Random forests predicting CPU time of workload in the presence of NiMC. (hpccg, feature set:3)

Coefficient of Det.:0.9953				
Out-of-Bag Score:0.9663				
Features	l3_tca	l3_dca	l3_ica	l3_tcm
Importance	0.0069	0.0075	0.2514	0.7343

Table A.23: Random forests predicting volume of RDMA traffic on target node. (hpccg, feature set:3)

Coefficient of Det.:0.9995				
Out-of-Bag Score:0.9964				
Features	l3_dca	l3_tca	l3_ica	l3_tcm
Importance	0.0005	0.0006	0.0014	0.9976

Table A.24: Random forests binary classification to detect NiMC. (hpccg, feature set:3)

Coefficient of Det.:1.0				
Out-of-Bag Score:0.9992				
Features	l3_ica	l3_tca	l3_dca	l3_tcm
Importance	0.0321	0.2186	0.2453	0.5039

Table A.25: Random forests predicting CPU time of workload in the presence of NiMC. (lammps, feature set:1)

Coefficient of Det.:0.9993						
Out-of-Bag Score:0.9949						
Features	tlb_dm	tlb_im	l1_icm	icy	l1_dcm	ib_bw
Importance	0.0013	0.0017	0.0071	0.0526	0.2269	0.7104

Table A.26: Random forests predicting CPU time of workload in the presence of NiMC. (lammps, feature set:1)

Coefficient of Det.:0.9986					
Out-of-Bag Score:0.9896					
Features	tlb_dm	tlb_im	l1_icm	icy	l1_dcm
Importance	0.0025	0.0052	0.0882	0.2156	0.6886

A.3 Feature Importance of PMCs for LAMMPS and Varying Prediction Criteria

Below are tables containing the feature importance values referenced in the work on detecting NiMC and predicting its impact on application performance. These tables contain the coefficient of determination, Out-of-Bag scores (OOB), and the feature importances for the listed features in binary classification or regression using the Random Forests package of scikit-learn in Python. There are four tables for each feature set, representing predictions of CPU-time (with and without the ib_bw feature), predictions of the volume of RDMA traffic, and binary classification of whether or not the workload is experiencing NiMC.

Table A.27: Random forests predicting volume of RDMA traffic on target node. (lammps, feature set:1)

Coefficient of Det.:0.9995					
Out-of-Bag Score:0.9965					
Features	tlb_dm	tlb_im	icy	l1_icm	l1_dcm
Importance	0.0006	0.0086	0.0383	0.0419	0.9105

Table A.28: Random forests binary classification to detect NiMC. (lammps, feature set:1)

Coefficient of Det.:1.0					
Out-of-Bag Score:1.0					
Features	tlb_dm	tlb_im	l1_icm	icy	l1_dcm
Importance	0.0216	0.0349	0.1316	0.2113	0.6005

Table A.29: Random forests predicting CPU time of workload in the presence of NiMC. (lammps, feature set:2)

Coefficient of Det.:0.9995								
Out-of-Bag Score:0.9968								
Features	l2_dcm/l2_tca	l2_tcm/l2_tca	l2_dcm	l2_tcm	l2_ich	l2_tca	l2_dch	l2_icm	ib_bw
Importance	0.0021	0.0031	0.0056	0.0109	0.0144	0.0281	0.0300	0.2049	0.7009

Table A.30: Random forests predicting CPU time of workload in the presence of NiMC. (lammps, feature set:2)

Coefficient of Det.:0.9993								
Out-of-Bag Score:0.9949								
Features	l2_tcm/l2_tca	l2_dcm/l2_tca	l2_tcm	l2_dcm	l2_tca	l2_dch	l2_icm	l2_ich
Importance	0.0024	0.0085	0.0087	0.0139	0.0254	0.0468	0.2000	0.6944

Table A.31: Random forests predicting volume of RDMA traffic on target node. (lammps, feature set:2)

Coefficient of Det.:0.9996								
Out-of-Bag Score:0.9969								
Features	l2_tcm/l2_tca	l2_dcm/l2_tca	l2_icm	l2_tcm	l2_dcm	l2_dch	l2_tca	l2_ich
Importance	0.0001	0.0002	0.0023	0.0076	0.0086	0.0092	0.0138	0.9581

Table A.32: Random forests binary classification to detect NiMC. (lammps, feature set:2)

Coefficient of Det.:1.0								
Out-of-Bag Score:1.0								
Features	l2_dcm/l2_tca	l2_tcm/l2_tca	l2_icm	l2_tcm	l2_dch	l2_dcm	l2_ich	l2_tca
Importance	0.0118	0.0201	0.0386	0.1311	0.1700	0.2058	0.2102	0.2123

Table A.33: Random forests predicting CPU time of workload in the presence of NiMC. (lammps, feature set:3)

Coefficient of Det.:0.9993					
Out-of-Bag Score:0.9949					
Features	l3_tcm	l3_tca	l3_dca	l3_ica	ib_bw
Importance	0.0023	0.0597	0.0620	0.1628	0.7132

Table A.34: Random forests predicting CPU time of workload in the presence of NiMC. (lammps, feature set:3)

Coefficient of Det.:0.9957				
Out-of-Bag Score:0.9688				
Features	l3_tcm	l3_ica	l3_tca	l3_dca
Importance	0.0129	0.1679	0.3676	0.4516

Table A.35: Random forests predicting volume of RDMA traffic on target node. (lammps, feature set:3)

Coefficient of Det.:0.9994				
Out-of-Bag Score:0.9957				
Features	l3_tcm	l3_ica	l3_tca	l3_dca
Importance	0.0016	0.0397	0.4747	0.4839

Table A.36: Random forests binary classification to detect NiMC. (lammps, feature set:3)

Coefficient of Det.:1.0				
Out-of-Bag Score:1.0				
Features	l3_ica	l3_tcm	l3_dca	l3_tca
Importance	0.0357	0.0680	0.4434	0.4529

Table A.37: Random forests predicting CPU time of workload in the presence of NiMC. (stream-cache, feature set:1)

Coefficient of Det.:0.9996					
Out-of-Bag Score:0.9969					
Features	tlb_dm	icy	tlb_im	l1_icm	l1_dcm	ib_bw
Importance	0.0002	0.0002	0.0065	0.0261	0.0748	0.8922

Table A.38: Random forests predicting CPU time of workload in the presence of NiMC. (stream-cache, feature set:1)

Coefficient of Det.:0.9987					
Out-of-Bag Score:0.9916					
Features	tlb_dm	icy	tlb_im	l1_icm	l1_dcm
Importance	0.0004	0.0019	0.0796	0.1594	0.7586

A.4 Feature Importance of PMCs for STREAM-cache and Varying Prediction Criteria

Below are tables containing the feature importance values referenced in the work on detecting NiMC and predicting its impact on application performance. These tables contain the coefficient of determination, Out-of-Bag scores (OOB), and the feature importances for the listed features in binary classification or regression using the Random Forests package of scikit-learn in Python. There are four tables for each feature set, representing predictions of CPU-time (with and without the ib_bw feature), predictions of the volume of RDMA traffic, and binary classification of whether or not the workload is experiencing NiMC.

Table A.39: Random forests predicting volume of RDMA traffic on target node. (stream-cache, feature set:1)

Coefficient of Det.:0.995					
Out-of-Bag Score:0.967					
Features	tlb_dm	icy	tlb_im	l1_icm	l1_dcm
Importance	0.0046	0.0057	0.0595	0.1208	0.8094

Table A.40: Random forests binary classification to detect NiMC. (stream-cache, feature set:1)

Coefficient of Det.:1.0					
Out-of-Bag Score:1.0					
Features	tlb_dm	icy	l1_icm	l1_dcm	tlb_im
Importance	0.0086	0.0339	0.2382	0.3547	0.3647

Table A.41: Random forests predicting CPU time of workload in the presence of NiMC. (stream-cache, feature set:2)

Coefficient of Det.:0.9992									
Out-of-Bag Score:0.9944									
Features	l2_tcm	l2_dcm	l2_tcm/l2_tca	l2_dcm/l2_tca	l2_dch	l2_icm	ib_bw	l2_tca	l2_ich
Importance	0.0003	0.0005	0.0009	0.0009	0.0025	0.0169	0.1976	0.3478	0.4326

Table A.42: Random forests predicting CPU time of workload in the presence of NiMC. (stream-cache, feature set:2)

Coefficient of Det.:0.9991								
Out-of-Bag Score:0.9937								
Features	l2_tcm/l2_tca	l2_tcm	l2_dcm	l2_dcm/l2_tca	l2_dch	l2_icm	l2_tca	l2_ich
Importance	0.0004	0.0005	0.0007	0.0008	0.0031	0.0185	0.4044	0.5715

Table A.43: Random forests predicting volume of RDMA traffic on target node. (stream-cache, feature set:2)

Coefficient of Det.:0.9962								
Out-of-Bag Score:0.9729								
Features	l2_tcm	l2_dcm/l2_tca	l2_dcm	l2_tcm/l2_tca	l2_dch	l2_icm	l2_ich	l2_tca
Importance	0.0026	0.0028	0.0078	0.0084	0.0130	0.0226	0.4097	0.5331

Table A.44: Random forests binary classification to detect NiMC. (stream-cache, feature set:2)

Coefficient of Det.:1.0								
Out-of-Bag Score:1.0								
Features	l2_dcm/l2_tca	l2_dch	l2_tcm/l2_tca	l2_tcm	l2_dcm	l2_ich	l2_tca	l2_icm
Importance	0.0069	0.0072	0.0261	0.0894	0.1152	0.2325	0.2465	0.2763

Table A.45: Random forests predicting CPU time of workload in the presence of NiMC. (stream-cache, feature set:3)

Coefficient of Det.:0.9987					
Out-of-Bag Score:0.9888					
Features	l3_tca	l3_tcm	l3_dca	l3_ica	ib_bw
Importance	0.0010	0.0047	0.0047	0.0597	0.9299

Table A.46: Random forests predicting CPU time of workload in the presence of NiMC. (stream-cache, feature set:3)

Coefficient of Det.:0.9976				
Out-of-Bag Score:0.9841				
Features	l3_tcm	l3_dca	l3_tca	l3_ica
Importance	0.0113	0.0269	0.0329	0.9289

Table A.47: Random forests predicting volume of RDMA traffic on target node. (stream-cache, feature set:3)

Coefficient of Det.:0.9937				
Out-of-Bag Score:0.9557				
Features	l3_tcm	l3_dca	l3_tca	l3_ica
Importance	0.0231	0.0342	0.0385	0.9041

Table A.48: Random forests binary classification to detect NiMC. (stream-cache, feature set:3)

Coefficient of Det.:1.0				
Out-of-Bag Score:0.9899				
Features	l3_tcm	l3_dca	l3_tca	l3_ica
Importance	0.0285	0.1203	0.3420	0.5092

Table A.49: Random forests predicting CPU time of workload in the presence of NiMC. (stream-dram, feature set:1)

Coefficient of Det.:0.9992					
Out-of-Bag Score:0.9931					
Features	l1_dcm	icy	tlb_im	tlb_dm	ib_bw	l1_icm
Importance	0.0001	0.0004	0.0506	0.0862	0.3042	0.5585

Table A.50: Random forests predicting CPU time of workload in the presence of NiMC. (stream-dram, feature set:1)

Coefficient of Det.:0.9981					
Out-of-Bag Score:0.984					
Features	l1_dcm	icy	tlb_im	tlb_dm	l1_icm
Importance	0.0001	0.0002	0.0843	0.1100	0.8053

A.5 Feature Importance of PMCs for STREAM-DRAM and Varying Prediction Criteria

Below are tables containing the feature importance values referenced in the work on detecting NiMC and predicting its impact on application performance. These tables contain the coefficient of determination, Out-of-Bag scores (OOB), and the feature importances for the listed features in binary classification or regression using the Random Forests package of scikit-learn in Python. There are four tables for each feature set, representing predictions of CPU-time (with and without the ib_bw feature), predictions of the volume of RDMA traffic, and binary classification of whether or not the workload is experiencing NiMC.

Table A.51: Random forests predicting volume of RDMA traffic on target node. (stream-dram, feature set:1)

Coefficient of Det.:0.9955					
Out-of-Bag Score:0.9665					
Features	l1_dcm	icy	tlb_im	tlb_dm	l1_icm
Importance	0.0046	0.0051	0.0996	0.1398	0.7509

Table A.52: Random forests binary classification to detect NiMC. (stream-dram, feature set:1)

Coefficient of Det.:1.0					
Out-of-Bag Score:1.0					
Features	icy	l1_dcm	tlb_im	tlb_dm	l1_icm
Importance	0.0013	0.0534	0.2235	0.2635	0.4583

Table A.53: Random forests predicting CPU time of workload in the presence of NiMC. (stream-dram, feature set:2)

Coefficient of Det.:0.9995									
Out-of-Bag Score:0.997									
Features	l2_tcm	l2_dcm	l2_icm	l2_tca	l2_ich	l2_dch	l2_tcm/l2_tca	l2_dcm/l2_tca	ib_bw
Importance	0.0008	0.0011	0.0198	0.0229	0.0366	0.0604	0.0744	0.0853	0.6987

Table A.54: Random forests predicting CPU time of workload in the presence of NiMC. (stream-dram, feature set:2)

Coefficient of Det.:0.9957								
Out-of-Bag Score:0.973								
Features	l2_dcm	l2_tcm	l2_tca	l2_dch	l2_ich	l2_tcm/l2_tca	l2_icm	l2_dcm/l2_tca
Importance	0.0001	0.0007	0.0258	0.0700	0.1022	0.1178	0.1886	0.4949

Table A.55: Random forests predicting volume of RDMA traffic on target node. (stream-dram, feature set:2)

Coefficient of Det.:0.9969								
Out-of-Bag Score:0.9793								
Features	l2_dcm	l2_tcm	l2_tca	l2_dcm/l2_tca	l2_tcm/l2_tca	l2_icm	l2_dch	l2_ich
Importance	0.0023	0.0075	0.0337	0.0394	0.0453	0.0492	0.0982	0.7243

Table A.56: Random forests binary classification to detect NiMC. (stream-dram, feature set:2)

Coefficient of Det.:1.0								
Out-of-Bag Score:1.0								
Features	l2_dcm	l2_tcm	l2_dcm/l2_tca	l2_tcm/l2_tca	l2_dch	l2_tca	l2_ich	l2_icm
Importance	0.0082	0.0087	0.0447	0.0876	0.1372	0.2071	0.2158	0.2907

Table A.57: Random forests predicting CPU time of workload in the presence of NiMC. (stream-dram, feature set:3)

Coefficient of Det.:0.9989					
Out-of-Bag Score:0.9914					
Features	l3_dca	l3_tca	l3_tcm	l3_ica	ib_bw
Importance	0.0008	0.0010	0.0473	0.1088	0.8422

Table A.58: Random forests predicting CPU time of workload in the presence of NiMC. (stream-dram, feature set:3)

Coefficient of Det.:0.9985				
Out-of-Bag Score:0.9882				
Features	l3_dca	l3_tca	l3_tcm	l3_ica
Importance	0.0273	0.0429	0.2245	0.7053

Table A.59: Random forests predicting volume of RDMA traffic on target node. (stream-dram, feature set:3)

Coefficient of Det.:0.9962				
Out-of-Bag Score:0.9726				
Features	l3_tca	l3_dca	l3_tcm	l3_ica
Importance	0.0259	0.0276	0.3163	0.6303

Table A.60: Random forests binary classification to detect NiMC. (stream-dram, feature set:3)

Coefficient of Det.:1.0				
Out-of-Bag Score:0.995				
Features	l3_tca	l3_dca	l3_tcm	l3_ica
Importance	0.0313	0.0599	0.4404	0.4684

References

[1] FCI Leap On-Board Transceiver Datasheet. `http://portal.fciconnect.com/Comergent/fci/documentation/datasheet/opticalinterconnect/oi_leap_obt.pdf`. accessed: 2016-05-11.

[2] Hasan Abbasi, J. Lofstead, Fang Zheng, K. Schwan, M. Wolf, and S. Klasky. Extending I/O through high performance data services. In *IEEE Conf. on Cluster Computing*, pages 1–10, Aug 2009.

[3] Jung Ho Ahn, Nathan Binkert, Al Davis, Moray McLaren, and Robert S Schreiber. HyperX: topology, routing, and packaging of efficient large-scale networks. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 41. ACM, 2009.

[4] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, pages 63–74, New York, NY, USA, 2008. ACM.

[5] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: incorporating long messages into the LogP model–one step closer towards a realistic model for parallel computation. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, SPAA '95, pages 95–105, New York, NY, USA, 1995. ACM.

[6] M. Alonso, S. Coll, J.M. Martnez, V. Santonja, and P. Lpez. Power consumption management in fat-tree interconnection networks. *Parallel Computing*, 48(0):59 – 80, 2015.

[7] Marina Alonso, Salvador Coll, J-M Martinez, Vicente Santonja, Pedro López, and José Duato. Dynamic power saving in fat-tree interconnection networks using on/off links. In *Parallel and Distributed Processing Symp., 2006. IPDPS 2006. 20th Intl.*, pages 8–pp. IEEE, 2006.

[8] Marina Alonso, Juan Miguel Martinez, Vicente Santonja, and Pedro Lopez. Reducing power consumption in interconnection networks by dynamically adjusting link width. In *Euro-Par 2004 Parallel Processing*, pages 882–890. Springer, 2004.

[9] Brian W Barrett, Ron Brightwell, Ryan E Grant, Simon D Hammond, and K Scott Hemmert. An evaluation of MPI message rate on hybrid-core processors. *Intl. Journal of High Performance Computing Applications*, 28(4):415–424, 2014.

[10] Brian W Barrett, Simon D Hammond, Ron Brightwell, and K Scott Hemmert. The impact of hybrid-core processors on MPI message rate. In *Proceedings of the 20th European MPI Users' Group Meeting*, pages 67–71. ACM, 2013.

[11] A. Bhatele, A.R. Titus, J.J. Thiagarajan, N. Jain, T. Gamblin, P.-T. Bremer, M. Schulz, and L.V. Kale. Identifying the culprits behind network congestion. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 113–122, May 2015.

[12] Abhinav Bhatele, Kathryn Mohror, Steven H Langer, and Katherine E Isaacs. There goes the neighborhood: performance degradation due to nearby jobs. In *Proc. of SC13: Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*, page 41. ACM, 2013.

[13] Leo Breiman. Out-of-bag estimation. Technical report, Citeseer, 1996.

[14] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[15] Marc Casas and Greg Bronevetsky. Active measurement of memory resource consumption. In *Parallel and Distributed Processing Symp., IEEE 28th Intl.*, pages 995–1004. IEEE, 2014.

[16] Cy Chan, Didem Unat, Michael Lijewski, Weiqun Zhang, John Bell, and John Shalf. Software design space exploration for exascale combustion co-design. In *Supercomputing*, volume 7905 of *Lecture Notes in Computer Science*, pages 196–212. Springer Berlin Heidelberg, 2013.

[17] IP Simple Network Management Protocol (SNMP) Causes High CPU Utilization. `http://www.cisco.com/image/gif/paws/7270/ipsnmphighcpu.pdf"`.

[18] Phillip Colella. Defining software requirements for scientific computing. 2004.

[19] S Conner, Sayaka Akioka, Mary Jane Irwin, and Padma Raghavan. Link shutdown opportunities during collective communications in 3-D torus nets. In *IEEE Intl. Parallel and Distributed Processing Symp., 2007. IPDPS 2007.*, pages 1–8. IEEE, 2007.

[20] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 1–12, New York, NY, USA, 1993. ACM.

[21] William J. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 39(6):775–785, 1990.

[22] John T Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303–312, 2006.

[23] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[24] Branimir Dickov, Miquel Pericas, Paul Carpenter, Nacho Navarro, and Eduard Ayguadé. Software-managed power reduction in Infiniband links. In *Proceeding of the 2014 Conf. on Parallell Processing, Intl., ICPP'14*. IEEE Computer Society, 2014.

[25] James Dinan, Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, and Rajeev Thakur. An implementation and evaluation of the MPI 3.0 one-sided communication interface. *Concurrency and Computation: Practice and Experience*, 2013.

[26] James Dinan, Ryan E Grant, Pavan Balaji, David Goodell, Douglas Miller, Marc Snir, and Rajeev Thakur. Enabling communication concurrency through flexible MPI endpoints. *International Journal of High Performance Computing Applications*, 28(4):390–405, 2014.

[27] DOE. Energy Website. DOE.GOV. `http://www.eia.doe.gov/energyexplained/index.cfm?page=electricity_use`. accessed: 2011-04-08. (Archived by WebCite at http://www.webcitation.org/).

[28] Doug Doefler and Brian W. Barrett. Sandia MPI microbenchmark suite (SMB). Technical report, Sandia National Laboratories, 2009.

[29] Matthew G F Dosanjh, Ryan E Grant, Patrick G Bridges, and Ron Brightwell. Re-evaluating network onload vs. offload for the many-core era. In *IEEE Intl. Conf. on Cluster Computing*. IEEE, 2015.

[30] Matthew G.F. Dosanjh, Taylor Groves, Ryan E. Grant, Patrick Bridges, and Ron Brightwell. RMA-MT: A benchmark suite for assessing mpi multi-threaded RMA performance. In *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2016)*, 2016.

[31] David Duarte, Yuh-Fang Tsai, Narayanan Vijaykrishnan, and Mary Jane Irwin. Evaluating run-time techniques for leakage power reduction. In *Proc. of the 2002 Asia and South Pacific Design Automation Conf.*, page 31. IEEE Computer Society, 2002.

[32] David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. Bandwidth bandit: Quantitative characterization of memory contention. In *Intl. Symp. on Code Generation and Optimization (CGO)*, pages 1–10. IEEE, 2013.

[33] Christian Engelmann and Thomas Naughton. A network contention model for the extreme-scale simulator. Technical report, Oak Ridge National Laboratory (ORNL), 2015.

[34] ExaCT Co-Design Center. `https://ccse.lbl.gov/ExaCT/index.html` 4/1/15.

[35] ASCAC Subcommittee for the Top Ten Exascale Research Challenges. Top ten exascale research challenges. Technical report, U.S. DoE, ASCR, Feb 2014. `http://science.energy.gov/~/media/ascr/ascac/pdf/meetings/20140210/Top10reportFEB14.pdf`.

[36] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118. ACM, 1978.

[37] Matthew I. Frank, Anant Agarwal, and Mary K. Vernon. Lopc: Modeling contention in parallel algorithms. In *Proc. of the SIXTH ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Las Vegas*, pages 276–287, 1997.

[38] Joshua D Goehner, Taylor L Groves, Dorian C Arnold, Dong H Ahn, and Gregory L Lee. An optimal algorithm for extreme scale job launching. In *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1115–1122. IEEE, 2013.

[39] Ryan E Grant and Ahmad Afsahi. Improving energy efficiency of asymmetric chip multithreaded multiprocessors through reduced os noise scheduling. *Concurrency and Computation: Practice and Experience*, 21(18):2355–2376, 2009.

[40] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, SIGCOMM '09, pages 51–62, New York, NY, USA, 2009. ACM.

[41] Taylor Groves, Dorian Arnold, and Yihua He. In-network, push-based network resource monitoring: scalable, responsive network management. In *Proc. of the Third Intl. Workshop on Network-Aware Data Management*, page 8. ACM, 2013.

[42] Taylor Groves, Ryan E Grant, and Dorian Arnold. NiMC: Characterizing and eliminating network-induced memory contention. In *30th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2016)*, 2016.

[43] Taylor Groves, Ryan E. Grant, Scott Hemmert, Simon Hammond, Michael Legenhagen, and Dorian Arnold. (SAI) stalled, active and idle: Characterizing power and performance of large-scale dragonfly networks. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, 2016.

[44] Taylor Groves, Samuel K. Gutierrez, and Dorian Arnold. A LogP extension for modeling tree aggregation networks. In *2015 HPCMASPA in association with IEEE International Conference on Cluster Computing*, pages 666–673, Sept 2015.

[45] Q. Gu and A. Marshall. Network management performance analysis and scalability tests: Snmp vs. corba. In *Network Operations and Management Symposium, 2004. NOMS 2004. IEEE/IFIP*, volume 1, pages 701–714 Vol.1, 2004.

[46] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. Bcube: a high performance, server-centric network architecture for modular data centers. *SIGCOMM Comput. Commun. Rev.*, 39(4):63–74, August 2009.

[47] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. Dcell: a scalable and fault-tolerant network structure for data centers. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, pages 75–86, New York, NY, USA, 2008. ACM.

[48] Hadoop. `http://hadoop.apache.org`, 2013.

[49] Hadoop file system. `http://hadoop.apache.org/docs/stable/hdfs_design.html`, 2013.

[50] Jeff R Hammond, Sayan Ghosh, and Barbara M Chapman. Implementing OpenSHMEM using MPI-3 one-sided communication. In *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools*, pages 44–58. Springer, 2014.

[51] Nathan Hanford, Vishal Ahuja, Mehmet Balman, Matthew K. Farrens, Dipak Ghosal, Eric Pouyoul, and Brian Tierney. Characterizing the impact

of end-system affinities on the end-to-end performance of high-speed flows. In *Proceedings of the Third International Workshop on Network-Aware Data Management*, NDM '13, pages 1:1–1:10, New York, NY, USA, 2013. ACM.

[52] Paul Hargrove. GASNet-EX collaboration. `https://sites.google.com/a/lbl.gov/gasnet-ex-collaboration`, 2015.

[53] Hbase. `http://hbase.apache.org/`, 2013.

[54] Brandon Heller, Srinivasan Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. ElasticTree: Saving energy in data center networks. In *NSDI*, volume 10, pages 249–264, 2010.

[55] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, 2009.

[56] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm. LogfP - a model for small messages in InfiniBand. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 6 pp., april 2006.

[57] Torsten Hoefler. Software and hardware techniques for power-efficient HPC networking. *Computing in Science & Engineering*, 12(6):30–37, 2010.

[58] Torsten Hoefler, Torsten Mehlan, Andrew Lumsdaine, and Wolfgang Rehm. Netgauge: A network performance measurement framework. In *High Performance Computing and Communications*, pages 659–671. Springer, 2007.

[59] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Characterizing the influence of system noise on large-scale applications by simulation. In *Proc. of the Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[60] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. LogGOPSim: simulating large-scale applications in the LogGOPS model. In *Proc. of the 19th ACM Intl. Symp. on High Performance Distributed Computing*, pages 597–604. ACM, 2010.

[61] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. LogGOPSim: simulating large-scale applications in the LogGOPS model. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 597–604, New York, NY, USA, 2010. ACM.

[62] Ming-yu Hsieh, Arun Rodrigues, Rolf Riesen, Kevin Thompson, and William Song. A framework for architecture-level power, area, and thermal simulation

and its application to network-on-chip design exploration. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):63–68, 2011.

[63] Steve Huss-Lederman, Bill Gropp, Anthony Skjellum, Andrew Lumsdaine, Bill Saphir, Jeff Squyres, et al. MPI-2: Extensions to the message passing interface. *University of Tennessee, available online at http://www. mpiforum. org/docs/docs. html*, 1997.

[64] Khaled Z. Ibrahim, Paul H. Hargrove, Constin Iancu, and Katherine Yelick. An evaluation of one-sided and two-sided communication paradigms on relaxed-ordering interconnect. In *IEEE International Parallel and Distributed Processing Symposium*, May 2014.

[65] Infiniband Roadmap. `http://www.infinibandta.org/content/pages.php?pg=technology_overview`. accessed: 2016-05-31.

[66] Fumihiko Ino, Noriyuki Fujimoto, and Kenichi Hagihara. LogGPS: a parallel computational model for synchronization analysis. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, PPoPP '01, pages 133–142, New York, NY, USA, 2001. ACM.

[67] Intel. Intel MPI benchmarks 4.0. `https://software.intel.com/en-us/articles/intel-mpi-benchmarks`, 2015.

[68] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L Chamberlain, Jonathan Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, et al. Exploring traditional and emerging parallel programming models using a proxy application. In *Proc. of the 27th IEEE Intl. Symp. on Parallel and Distributed Processing (IPDPS)*, pages 919–932. IEEE, 2013.

[69] Eun Jung Kim, Ki Hwan Yum, Greg M Link, Narayanan Vijaykrishnan, M Kandemir, Mary Jane Irwin, M Yousif, and Chita R Das. Energy optimization techniques in cluster interconnects. In *Proc. of the 2003 international symposium on Low power electronics and design*, pages 459–464. ACM, 2003.

[70] John Kim, Wiliam J. Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. *SIGARCH Comput. Archit. News*, 36(3):77–88, June 2008.

[71] John Kim, William J Dally, and Dennis Abts. Flattened butterfly: a cost-efficient topology for high-radix networks. *ACM SIGARCH Computer Architecture News*, 35(2):126–137, 2007.

[72] Jong Yong Kim and John Shawe-Taylor. Fast string matching using an n-gram algorithm. *Software: Practice and Experience*, 24(1):79–88, 1994.

[73] Peter Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, W Carson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, et al. Exascale computing study: Technology challenges in achieving exascale systems. 2008.

[74] James H Laros, Pavel Pokorny, and David DeBonis. Powerinsight-a commodity power measurement capability. In *Green Computing Conference (IGCC), 2013 International*, pages 1–6. IEEE, 2013.

[75] James H Laros III, Kevin T Pedretti, Suzanne M Kelly, Wei Shu, and Courtenay T Vaughan. Energy based performance tuning for large scale high performance computing systems. In *Proc. of the 2012 Symp. on High Performance Computing*, page 6. Society for Computer Simulation Intl., 2012.

[76] A. Lastovetsky and M. O'Flynn. A performance model of many-to-one collective communications for parallel computing. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1 –8, march 2007.

[77] A. Legrand, L. Marchal, and Y. Robert. Optimizing the steady-state throughput of scatter and reduce operations on heterogeneous platforms. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 176, april 2004.

[78] Feihui Li, Guilin Chen, M Kandemir, and M Karakoy. Exploiting last idle periods of links for network power management. In *Proc. of the 5th ACM international conference on Embedded software*, pages 134–137. ACM, 2005.

[79] Jian Li, Wei Huang, Charles Lefurgy, Lixin Zhang, Wolfgang E Denzel, Richard R Treumann, and Kun Wang. Power shifting in thrifty interconnection network. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th Intl. Symp. on*, pages 156–167. IEEE, 2011.

[80] Ning Liu, Christopher Carothers, Jason Cope, Philip Carns, and Robert Ross. Model and simulation of exascale communication networks. *Journal of Simulation*, 6(4):227–236, 2012.

[81] Priya Mahadevan, Puneet Sharma, Sujata Banerjee, and Parthasarathy Ranganathan. A power benchmarking framework for network devices. In *NETWORKING 2009*, pages 795–808. Springer, 2009.

[82] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[83] Jeffrey C Mogul, Jayaram Mudigonda, Nathan Binkert, Parthasarathy Ran-

ganathan, and Vanish Talwar. Using asymmetric single-isa cmps to save energy on operating systems. *IEEE micro*, (3):26–41, 2008.

[84] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R De Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[85] Alessandro Morari, Roberto Gioiosa, Robert W Wisniewski, Francisco J Cazorla, and Mateo Valero. A quantitative analysis of os noise. In *Proc. of the Parallel and Distributed Processing Symp. (IPDPS)*, pages 852–863. IEEE, 2011.

[86] Csaba Andras Moritz and Matthew I. Frank. Logpc: modeling network contention in message-passing programs. In *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '98/PERFORMANCE '98, pages 254–263, New York, NY, USA, 1998. ACM.

[87] MPI Forum. MPI: A message-passing interface standard version 3.0. Technical report, University of Tennessee, Knoxville, 2012.

[88] MPI Forum. MPI: A message-passing interface standard version 3.1. Technical report, University of Tennessee, Knoxville, 2015.

[89] GitHub repository for MRNet. `https://github.com/dyninst/mrnet`, 2016.

[90] Misbah Mubarak, Christopher D Carothers, Robert B Ross, and Philip Carns. Using massively parallel simulation for mpi collective communication modeling in extreme-scale networks. In *Proceedings of the 2014 Winter Simulation Conference*, pages 3107–3118. IEEE Press, 2014.

[91] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. *SIGCOMM Comput. Commun. Rev.*, 39(4):39–50, August 2009.

[92] NOAA Website. NOAA.GOV. `http://www.noaanews.noaa.gov/stories2016/011116-noaa-completes-weather-and-climate-supercomputer-upgrades.html`. accessed: 2016-09-26.

[93] Ohio State University. OSU micro-benchmarks 4.4.1. `http://mvapich.cse.ohio-state.edu/benchmarks/`, 2015.

[94] GitHub repository for OpenMPI. `https://github.com/open-mpi/ompi`, 2016.

[95] OpenFabrics. `https://www.openfabrics.org/` (visited Oct. 2014).

[96] Open time series database (opentsdb). `http://opentsdb.net`, 2013.

[97] Tcollector. `http://opentsdb.net/tcollector.html`, 2013.

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

[99] Swann Perarnau, Marc Tchiboukdjian, and Guillaume Huard. Controlling cache utilization of HPC applications. In *Proc. of the Intl. Conf. on Supercomputing*, pages 295–304. ACM, 2011.

[100] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 337–350, New York, NY, USA, 2012. ACM.

[101] Fabrizio Petrini, DK Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proc. of the Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 55–55. IEEE, 2003.

[102] Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of computational physics*, 117(1):1–19, 1995.

[103] Download site for RMA-MT benchmarks. `http://www.cs.sandia.gov/smb/rma-mt.html`, 2016.

[104] Arun F Rodrigues, K Scott Hemmert, Brian W Barrett, Chad Kersey, Ron Oldfield, Marlo Weston, R Risen, Jeanine Cook, Paul Rosenfeld, E CooperBalls, et al. The structural simulation toolkit. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):37–42, 2011.

[105] Brian M Rogers, Anil Krishna, Gordon B Bell, Ken Vu, Xiaowei Jiang, and Yan Solihin. Scaling the bandwidth wall: challenges in and avenues for cmp scaling. In *ACM SIGARCH*, volume 37, pages 371–382. ACM, 2009.

[106] Philip C. Roth, Dorian C. Arnold, and Barton P. Miller. Mrnet: A software-based multicast/reduction network for scalable tools. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, pages 21–, New York, NY, USA, 2003. ACM.

[107] Karthikeyan P Saravanan, Paul M Carpenter, and Alex Ramirez. Power/performance evaluation of energy efficient ethernet (eee) for high perfor-

mance computing. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE Intl. Symp. on*, pages 205–214. IEEE, 2013.

[108] J. Schonwalder, A. Pras, M. Harvan, J. Schippers, and R. van de Meent. Snmp traffic analysis: Approaches, tools, and first results. In *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on*, pages 323–332, 2007.

[109] Martin Schulz, Jim Galarowicz, Don Maghrak, William Hachfeld, David Montoya, and Scott Cranford. Open— speedshop: An open source infrastructure for parallel performance analysis. *Scientific Programming*, 16(2-3):105–121, 2008.

[110] Li Shang, Li-Shiuan Peh, and Niraj K Jha. Dynamic voltage scaling with links for power optimization of interconnection networks. In *Proc.. The Ninth Intl. Symp. on High-Performance Computer Architecture, HPCA.*, pages 91–102. IEEE, 2003.

[111] Vassos Soteriou, Noel Eisley, and Li-Shiuan Peh. Software-directed power-aware interconnection networks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(1):5, 2007.

[112] Vassos Soteriou and Li-Shiuan Peh. Exploring the design space of self-regulating power-aware on/off interconnection networks. *Parallel and Distributed Systems, IEEE Transactions on*, 18(3):393–408, 2007.

[113] Srinivas Sridharan, James Dinan, and Dhiraj D. Kalamkar. Enabling efficient multithreaded MPI communication through a library-based implementation of MPI endpoints. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2014.

[114] GitHub repository for SST. `https://github.com/sstsimulator`, 2016.

[115] Rajesh Subramanyan, José Miguel-Alonso, and José A. B. Fortes. A scalable snmp-based distibuted monitoring system for heterogeneous network computing. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[116] Nathan R. Tallent, Abhinav Vishnu, Hubertus Van Dam, Jeff Daily, Darren J. Kerbyson, and Adolfy Hoisie. Diagnosing the causes and severity of one-sided message contention. In *Proc. of the 20th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 130–139, New York, NY, USA, 2015. ACM.

[117] Monika ten Bruggencate, Duncan Roweth, and Steve Oyanagi. Thread-safe SHMEM extensions. In Stephen Poole, Oscar Hernandez, and Pavel Shamis,

editors, *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools*, volume 8356 of *Lecture Notes in Computer Science*, pages 178–185. Springer International Publishing, 2014.

[118] Rajeev Thakur and William D. Gropp. Test suite for evaluating performance of MPI implementations that support MPI_THREAD_MULTIPLE. In Franck Cappello, Thomas Herault, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4757 of *Lecture Notes in Computer Science*, pages 46–55. Springer Berlin Heidelberg, 2007.

[119] Clark D Thompson. Area-time complexity for vlsi. In *Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 81–88. ACM, 1979.

[120] Ananta Tiwari, Anthony Gamst, Michael A Laurenzano, Martin Schulz, and Laura Carrington. Modeling the impact of reduced memory bandwidth on HPC applications. In *Euro-Par 2014 Parallel Processing*, pages 63–74. Springer, 2014.

[121] Top 500 Supercomputer Sites. `http://www.top500.org/` (visited August 2016).

[122] Ehsan Totoni, Nikhil Jain, and Laxmikant V Kale. Power management of extreme-scale networks with on/off links in runtime systems. 2013.

[123] Ehsan Totoni, Nikhil Jain, and Laxmikant V Kale. Toward runtime power management of exascale networks by on/off control of links. In *2013 IEEE 27th Intl. Parallel and Distributed Processing Symp. Workshops & PhD Forum (IPDPSW)*, pages 915–922. IEEE, 2013.

[124] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *PHYSOR 2014*.

[125] Christian Trott, Courtney Vaughan, Sikandar Mashayak, Mike Brown, Carl Ponder, Paul Crozier, and Fiona Reid. Lammps benchmarks. http://lammps.sandia.gov. 2015-10-12.

[126] Bogdan Marius Tudor, Yong Meng Teo, and Simon See. Understanding off-chip memory contention of parallel programs in multicore systems. In *Proc. of the Intl. Conf. on Parallel Processing (ICPP)*, pages 602–611. IEEE, 2011.

[127] Keith D Underwood, Michael Levenhagen, and Arun Rodrigues. Simulating red storm: Challenges and successes in building a system simulation. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–10. IEEE, 2007.

[128] U.S. DOE. Fact sheet: Collaboration of Oak Ridge, Argonne, and Livermore (CORAL). http://energy.gov/, December 2014.

[129] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[130] Yahoo! Launches World's Largest Hadoop Production Application. `http://developer.yahoo.com/blogs/hadoop/posts/2008/02/yahoo-worlds-largest-production-hadoop/`, 2008.

[131] Felix Zahn, Pedro Yebenes, Steffen Lammel, Pedro J Garcia, et al. Analyzing the energy (dis-) proportionality of scalable interconnection networks. In *2016 2nd IEEE International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB)*, pages 25–32. IEEE, 2016.

[132] R. Zamani, A. Afsahi, Ying Qian, and V.C. Hamacher. A feasibility analysis of power-awareness and energy minimization in modern interconnects for high-performance computing. In *Cluster Computing, 2007 IEEE Intl. Conf. on*, pages 118–128, Sept 2007.

[133] R. J. Zerr and R. S. Baker. SNAP: SN (Discrete Ordinates) Application Proxy: Description. *Los Alamos National Laboratories, Tech. Rep. LA-UR-13-21070*, 2013.