Improving HPC Communication Library Performance on Modern Architectures

Matthew G. F. Dosanjh
University of New Mexico

Matthew G. F. Dosanjh
Candidate

Computer Science
Department

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

Approved by the Dissertation Committee:

Patrick G. Bridges , Chairperson

Ryan E. Grant

Anthony Skjellum

Dorian C. Arnold

Improving HPC Communication Library Performance on Modern Architectures

by

Matthew G. F. Dosanjh

B.S., Computer Science, University of New Mexico, 2010
M.S., Computer Science, University of New Mexico, 2013

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

December, 2017

Dedication

To my friends and family for helping me through this... uhh... grad school thing.

“Everything is perfect, but there is a lot of room for improvement.”

Shunryu Suzuki

Acknowledgments

There are a number of people without whom this document would not exist. First I'd like to acknowledge my advisor, Patrick Bridges. His guidance has proved essential in my development as a researcher and in the development of this dissertation. Through working with him, I've learned how to develop research ideas, test and explore these ideas, and communicate results to the scientific community. The opportunities I have now can be traced back seven years to when I first had the opportunity to work with Patrick, and I am forever grateful.

During the last four years of my PhD, I have been extremely fortunate to hold an internship at Sandia National Laboratories[1]. During the majority of that time, Ryan Grant has served as a mentor and de facto second advisor. The skills and knowledge I gained from working with him have been invaluable. The time and effort he contributed to help increase my understanding of the field and discuss and evaluate research ideas has impacted every part of this dissertation.

I would also like to acknowledge the other two members of my committee. Dorian Arnold has always been willing provide in-depth constructive criticism which proved extremely useful to create the best and most polished work of my career. Tony Skjellum has provided useful feedback through his experience and unique area knowledge that helped me contextualize my work. Their contributions have greatly helped in the work and writing of this dissertation.

I would like to thank all my fellow students and friends at the Scaleable Systems Laboratory. David DeBonis, Noah Evans, Taylor Groves, Aaron Gonzales, Dewan Ibtesham, Scott Levy, Oscar Mondragon, Whit Schonbein, Philip Soltero, and Hans Weeks have all helped by providing feedback, advice, and a platform for discussing ideas. I'd also like to thank some of my other peers from other research groups and institutions: Sridutt Bhalachandra, Jeffery Bowels, Carl Evans, and George Stelle.

There were a number of people that helped guide me before I got to the path that led to this dissertation. Joel Castelanos gave me a chance to become a teaching assistant, for a class I was clearly unprepared to teach, and helped solidify my path to grad school. Andree Jacobson, through the Cluster and Network Management Institute, taught me the basics of cluster computing which ultimately led me to HPC as a research path. Suzanne M. Kelly, my first mentor at Sandia, gave me an opportunity to work on my first HPC research project and introduced me to the research group I continue to work with.

[1]Parts of this research were supported by the Exascale Computing Project (ECP 17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. Additionally, this research was performed in part at Sandia National Laboratories, supported by the U.S. Department of Energy contractcontract DE-NA0003525

Finally, I would like to thank my family for all the love and support I needed over the years. My wife, Marie, has always been there for me and pushed me forward through the rough times where I wanted to give up. My daughter, Athena, has given me a reason to wake up at 5:30 in the morning, whether I want to or not. My parents have always been there for advice and support over the years and are directly responsible for getting me hooked on math at an early age.

Improving HPC Communication Library Performance on Modern Architectures

by

Matthew G. F. Dosanjh

B.S., Computer Science, University of New Mexico, 2010
M.S., Computer Science, University of New Mexico, 2013

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

Abstract

As high-performance computing (HPC) systems advance towards exascale (10^{18} operations per second), they must leverage increasing levels of parallelism to achieve their performance goals. In addition to increased parallelism, machines of that scale will have strict power limitations placed on them. One direction currently being explored to alleviate those issues are many-core processors such as Intel's Xeon Phi line. Many-core processors sacrifice clock speed and core complexity, such as out of order pipelining, to increase the number of cores on a die. While this increases floating point throughput, it can reduce the performance of serialized, synchronized, and latency sensitive code paths, such as traditional communication libraries.

In this thesis, I examine the impact of many-core processors on large-scale scientific applications and explore ways to improve performance for both future and legacy applications. I examine the effect by characterizing the performance and power tradeoffs for different core frequencies and network hardware. Then, I explore the viability of next-generation programming models by benchmarking the performance of communication libraries utilizing multi-threaded one-sided communication. Next, I improve communication library performance for legacy applications for many-core systems through optimizing the matching algorithm to leverage single instruction multiple data vectors and caching behavior. Finally, I explore two other matching algorithm optimizations targeted at next-generation processors and applications.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The drive to improve performance and capability in modern scientific computing has caused computational scientists to leverage increasing levels of parallelism in their applications. Traditional high performance computing (HPC) has focused on providing communication and data movement between computers (nodes) with fast, complex processors with limited parallelism on the chip. However, as clock speeds approach their theoretical and practical limits, new generations of processors are focusing on improving performance through parallelism and new hardware features. New processor architectures, such as the many-core Intel Xeon Phi, have opted to reduce core speed and complexity to enable even greater parallelism.

This poses a challenge to HPC middleware and communication libraries which have been designed and optimized for fast complex cores and limited parallelism. Expensive data structures in critical sections can decrease performance for latency sensitive regions of the code, keeping them from utilizing increased parallelism. In addition, proposed programing models have the potential to exacerbate the problem by redistributing work into smaller pieces, increasing synchronization and middleware overhead.

This chapter introduces the broader HPC concepts and research challenges for the use of communication libraries on many-core architectures. Section 1.1 presents advances in modern HPC hardware including many-core processors, network offload, and high bandwidth memory. Section 1.2 presents the current state of the art in HPC communication libraries, focusing on the most prevalent interface, MPI. Section 1.3 discusses the design challenges and constraints facing next generation HPC systems: power, memory performance, and increased parallelism. Section 1.4 presents the technical contributions of this dissertation. Finally, Section 1.5 presents an outline for the remaining chapters of this dissertation.

1.1 Modern HPC hardware

Current trends in HPC architectures have eschewed traditional processor clock speed for improving performance through other hardware optimizations. Many-core architectures, network offload, and high bandwidth memory are three advances proposed to improve performance for current and future HPC architectures. The rest of this section will discuss these hardware advancements in detail; 1.1.1 will discuss many-core architectures, 1.1.2 will discuss network offload, and 1.1.3 will discuss high bandwidth memory.

1.1.1 Many Lightweight Cores

Many-core systems sacrifice core complexity and speed to provide a large increase in the number of cores. For example, the Xeon Phi Knight's Landing (KNL) architecture has up to 72 cores on a chip compared to 22 in the Intel Xeon Broadwell. This comes at the cost of low clock speeds, maxing out at 1.5 GHz rather than the 3.7 GHz[1] on Broadwell, and decreased complexity, with a 3 step out-of-order pipeline compared to the 20 step out-of-order pipeline per-thread on Broadwell. This allows highly parallel codes to achieve higher computational throughput by leveraging all of the cores. It also has limitations for serial segments of code and hiding latency of operations such as memory lookups.

[1]This clock speed is based on a different model of Broadwell, the frequency for the 22 core model is 2.2 GHz.

1.1.2 Network Offload

Network offload is a hardware technique that utilizes a dedicated network protocol processor to reduce interference with application processes. This approach avoids costly context switches, system noise, and memory contention [49]. This approach can vary in the amount of a network processing operation done on the added hardware. This amount varies from Intel's Infinipath network, which offloads a subset of low level network operations, to Atos' BXI interconnect [28] which fully offloads network and HPC communication library processing. While this approach is becoming increasingly available and can become a solution for HPC communication libraries in the future, it presents an interesting challenge to HPC communication libraries as different networks diverge in utilized code-path and optimizations on one platform will not have the same impact on another.

1.1.3 High Bandwidth Memory

As processors continue to increase parallelism to increase throughput, performance bottlenecks have shifted to other parts of the system. Memory bandwidth is one of the more prominent limiting factors for many-core systems. To address this constraint, the Knight's Landing processor has introduced high bandwidth memory in the form of Multi-Channel DRAM (MCDRAM). MCDRAM is structurally different than standard memory, being stacked on top of the processor die. MCDRAM increases memory bandwidth to the processor but has limited capacity. While the MCDRAM can act as a cache, many applications see better performance when running exclusively out of MCDRAM [97]. This places a memory limitation on HPC communication libraries which need to maintain state for each process in the job. The memory constraint is expected to further reduce the amount of available memory per core in next generation systems.

1.2 The Message Passing Interface

The Message Passing Interface (MPI) is a network communication interface defined by the community. It has a number of widely-used implementations including Open MPI and MPICH. MPI is the most common communication library for HPC workloads and is utilized in parallel scientific applications. While alternatives like OpenSHMEM and Unified Parallel C exist, they do not see the same community adoption as MPI. Due to the ubiquity of MPI, this dissertation focuses on it as the primary HPC communication library. Subsection 1.2.1 describes the traditional two-sided and newer one-sided user interfaces. Subsection 1.2.2 presents some of the implementation details of message matching, a required component for MPI's two-sided interface. Finally, Subsection 1.2.3 presents the details of MPI's thread support.

1.2.1 Message Passing and Remote Memory Access

Traditional message passing has been one of the main focuses of MPI requiring both sides to interact with the library to manage data placement. The receiver of a message must provide an inbox buffer and identification data. The sender of a message must provide data and routing information to ensure that the data gets place properly. To properly place the data, the receiving process does message matching which is described in detail in section 1.2.2. RMA is another form of communication in MPI that attempts to avoid library processing steps like matching. It does this by placing data in a sender specified region and does not require interaction from the receiver. Section 2.2 has more details on MPI and RMA.

1.2.2 Message Matching

The message matching engine is one of the major internal computations in MPI. It matches incoming message data to a buffer specified in a receive function call. This matching uses two identifiers, a tag and the sender's rank. The matching engine is constrained by behavior guarantees in the MPI specification. First, the matching engine must ensure order. If a receive request matches two pending messages, it must match the first to enter the match list. Second, receives must support wild card identifiers. This allows for unexpected communication between two nodes. For example, AMG2013 [7] uses this to establish communication outside of its regular halo exchange.

Standard matching implementations utilize a pair of linked lists: a posted receive queue and an unexpected message queue. To post a receive, the library searches through the unexpected queue and, if no match is found, appends a new element to the posted receive queue. An incoming message must do the same for the other list. Due to the complexity of guaranteeing order and wild card usage, most multithreaded implementations treat the matching engine as a singe large critical section. This significantly limits MPI performance on many-core systems.

1.2.3 Thread Modes

The MPI specification defines four thread modes that provide optimizations based on user-guaranteed behavior. The most general is *MPI_THREAD_MULTIPLE*, which provides a thread-safe MPI interface. This mode has traditionally been implemented using coarse grained synchronization methods that attempt to acquire a lock upon entry to MPI. More recent implementations have been exploring fine grained synchronization to minimize the performance impact of critical sections. Unfortunately, the lack of a performant thread safe MPI limits the performance of programs that use multithreaded communications, like those expected to be needed to take advantage of an exascale many-core system. This is discussed in more depth in Section 2.2.2.

1.3 Design Challenges for Communication Libraries on Many-Core Systems

As most communication libraries are optimized for traditional systems, there are a few major challenges that arise for large scale HPC system with many-core processors. Three of the major challenges are power usage, memory bottlenecks, and parallelism.

1.3.1 Power

While systems have been increasing their computation capability, the power required per operation has been decreasing slowly. This has led to two major concerns among current high performance clusters: power caps and energy budgets. Power caps are a limit to the instantaneous wattage a cluster can draw from the power grid. This is primarily a concern for capability class supercomputers, where the expected increase to power requirements gets close and potentially exceeds the amount of power available. Another concern is energy budgets. With systems requiring more power, the expenses required to run machines have been steadily increasing. This has led to a desire to explore potential power saving techniques for network technologies such as offload network cards and network routers.

This is discussed in more detail in Section 2.4.2

1.3.2 Memory

One of the major sacrifices that many-core processors make is a reduced out-of-order pipeline which reduces the processor's ability to handle memory look-ups and prefetch data. The lack of deep out-of-order processing reduces the processor's ability to hide the latency of cache misses. This particularly impacts the use of data structures that are non-contiguous in memory, such as linked lists. For instance, the KNL architecture only supports 3 step out of order processing to hide the latency of memory lookups compared to the 20 step out of order pipeline of traditional cores. This can be even more problematic because of small cache sizes. The KNL's biggest level of cache is the 2 MiB L2 that is shared between two cores with a total of eight hyper-threads.

Memory latency has the potential for large impacts on the performance of non-contiguous data structures in HPC communication libraries, such as the matching engine. The lack of complexity causes a larger latency when fetching an entry from memory and the limited cache size causes these data structures to be evicted from cache between uses. This is particularly relevant to HPC communication libraries, as applications often process large amounts of data between communication phases.

1.3.3 Parallelism

One of the biggest challenges in many-core architectures is in fully utilizing the parallelism provided by the architecture. As the current generation of Xeon Phi can have 72 cores with a total of 288 hardware thread contexts, an application must utilize a large number of threads per node to fully utilize the available hardware resources. This is a constraint for MPI-everywhere legacy applications, because increasing the number of ranks causes MPI to require more time and resources to operate. It also creates an additional trade-off for hybrid applications, as users will have to determine how many processes per node and how many threads per processes are optimal on each new architecture.

HPC communication libraries must support application models that leverage this increasing parallelism. MPI+X is an attempt to couple two different parallelism models together. It usually consists of MPI to handle inter-node parallelism and a threading library, such as OpenMP, to handle the intra-node parallelism. The combination of MPI+X programing models and fine grained messaging techniques, such as communication and computation overlap, has driven the desire to have performant, multithreaded communication libraries.

1.4 Contributions

The goal of this dissertation is to demonstrate that successfully leveraging parallelism, offload, and caching behaviors will improve the performance of HPC communication libraries on modern hardware architectures.

The major contributions of this work are:

- An evaluation of the impact of many-core processors on modern communication systems including onload and offload network architectures
- A suite of benchmarks, RMA-MT, to explore the performance of next generation communication systems
- Studies using these benchmarks of the performance of both MPI RMA and shmem in hybrid programing contexts
- A vector matching architecture optimized for many-core processors that improves MPI performance by reducing cache cache misses and leveraging modern vector operations
- A multi-threaded matching architecture that reduces lock contention in current HPC communication systems
- An optimistic matching approach to reduce memory overheads in modern MPI implementations by utilizing lossy compression
- A software based explicit cache management technique that improves HPC communication performance on many-core and traditional processor architectures

1.5 Dissertation Outline

The rest of this dissertation is structured as follows: Chapter 2 presents an overview of the related work. Chapter 3 presents a exploratory study of how modern many-core architectures impact communication processing on onload and offload networks. Chapter 4 presents novel benchmarks and performance studies that explore the new parallelism model of one-sided communication in multi-threaded contexts. Chapter 5 presents techniques that leverage current state of the art hardware to improve MPI matching for traditional applications. Chapter 6 presents preliminary results for novel techniques for optimizing MPI matching for next generation hardware and applications. Finally, Chapter 7 summarizes the contributions of this dissertation and presents future directions in this work.

Chapter 2

Related Work

2.1 Introduction

This dissertation builds on three key research areas of improvements to HPC communication libraries: software techniques, hardware techniques, and alternative programing models. Section 2.2 presents background on MPI including message matching, threading, and RMA. Section 2.3 presents the state of the art in MPI matching research, including studies of matching performance and behavior, software techniques to improve message matching, and hardware techniques to offload message matching. Section 2.4 discusses the impact of modern architectures on MPI, specifically discussing multi-threaded MPI and power concerns. Section 2.5 presents background on other programing models for HPC systems, including one-sided communication and task based parallelism.

2.2 Message Passing Interface (MPI)

MPI [81] is an HPC communication library specification, the first version of which was published in November 1992. It is the most common HPC communication library on systems today. MPI's goal is to provide a performant and portable interface for tightly coupled scientific simulations. To accomplish this, the standard has defined interfaces for a number of different communication types, including point-to-point, collective, and one-sided communication. This communication diversity allows scientific applications to utilize cluster computation for different communication needs, such as mesh problem decomposition, global all-to-all communications, and scatter and reduce data dissemination.

Traditional MPI point-to-point communication is built on the concept of receiver-specified data placement. Each communication consists of two parts: a send and a receive. Messages are delivered based on two identifiers: tag and rank. Tag is a user-defined value that must match on both sides of the communication to identify the message. Rank is the MPI address assigned to each process, and each side of a communication must specify the other to properly route the message from sender to receiver. A send is given a buffer of data to transfer, an identifying tag, and a destination rank, while the receive takes a buffer in which to place the data, an identifying tag, and a source rank. This design imposes significant implementation requirements on the library but provides convenient communication semantics to users.

Since its inception, there have been many implementations, including MPICH [102], MVAPICH [103], Open MPI [104], FT-MPI [37], LA-MPI [6], LAM/MPI, Platform MPI, MPI Pro [30], ChaMPIon [29], and PACX-MPI [41]. There have been vendor-tuned implementations, such as Cray MPI, BG-MPI, and Intel MPI. Also, there have been implementations that have extended the standard to change functionality, such as MPI/RT [64] for realtime computation and FG-MPI [63] for task based programing models.

2.2.1 Matching

MPI matching works on three key elements: a source address, a matching tag, and a communicator. An MPI communicator is a special isolation mechanism that allows processes that belong to that communicator to send messages to each other. Each process in a communicator has a corresponding address, called a rank. Each process keeps track of the posted receives (messages that are expected to arrive) and unexpected messages (messages that have been received but did not match a posted receive). The unexpected messages are useful for cases where the receiving process is not ready for the incoming message or the communication is not predictable. MPI creates unexpected-message buffers to store the data until MPI receives a matching request with an application buffer to place the data in.

When a process wishes to receive a message, it calls MPI_Recv, which first searches the unexpected messages for a match. If a match is found, MPI moves the buffered message into the correct location or fetches it if it is not buffered. If no match was found in the unexpected messages, MPI must track a posted receive with the corresponding source, tag, and communicator.

MPI implementation support for matching is constrained by behavior guarantees in the MPI specification: First, the data structure must ensure order. Section 3.5 of the standard defines ordering of a point-to-point communication within a communicator [81]. The ordering requirements consists of two rules: If two receives both match the same message, the first must be fulfilled before the second can match said message. Similarly, if a process sends two messages to the same destination that match the same receive, the first message must be matched before the the second message can match said receive.

Second, receives must support wild-cards. The MPI standard supports two wild-cards, namely any source and any tag. These allow a receive to match against a wider range of tags and can be useful for application behavior such as unexpected communication between two nodes. For example, AMG2013 [7] uses this to establish communication outside of its regular halo exchange. There have proposals that would allow communicators to be created with out the wild-card constraint, as it constraint limits optimizations for application that do not use these features.

Traditional MPI Matching Implementation

```
Data: Request with Tag, Source, and Context ID
Result: Matched Element (null if not found)
lock ML;
foreach element in UMQ do
    if element matches Tag, Source, and Context ID then
        remove element from UMQ;
        unlock ML;
        return element;
    end
end
append new element to PRQ;
unlock ML;
return null;
```

Algorithm 1: Baseline MPICH CH3 - Post Receive

Algorithm 1 shows one of the most common matching engine implementations. It is based on MPICH CH3's matching engine [102], and is found in many derivatives of that codebase. It uses two large lists, an unexpected message queue (UMQ) and a posted receive queue (PRQ), to track requests. Because it does not separate the list based on communicator, it has three matching parameters: a user given tag, an expected source, and the communicator's identifier. With fine grained locks, a matching lock (ML) is locked upon entry and released on exit. The function then searches the UMQ, removes the first match and returns it or appends a new entry to the PRQ. For incoming messages, the process is similar, but the queues are switched.

MPI message matching is a key feature of the API that has helped contribute to its success over the last two decades. Having matched send/receive semantics makes programming easier and helps to clearly define communication isolation through source/destination information, MPI communicators, and user-defined tags. MPI message matching has been performant in MPI implementations since MPI's inception. As a result, it has not been optimized as heavily as other aspects of MPI libraries, particularly architectural optimizations for the matching lists.

2.2.2 Threading

In 1994, Chowdappa et al. proposed a thread safe version of MPI [25]. While early versions of the standards defined a 'thread compliant' term for MPI, it was not until MPI 2.1 that the current thread safety requirements were added to the standard. The MPI specification defines four thread modes that provide optimizations based on user-guaranteed behavior. The most general is *MPI_THREAD_MULTIPLE*, which provides a thread-safe MPI interface. This mode has traditionally been implemented using coarse grained synchronization methods that attempt to acquire a lock upon entry to MPI. As described in Section 2.4.1, more recent implementations have explored fine grained synchronization to minimize the performance impact of critical sections.

MPI_THREAD_SERIALIZED is used in most applications. It does not provide thread safety but places no other constraints on the application. This allows for applications to call MPI from multiple threads as long as the calls are not concurrent. *MPI_THREAD-_SINGLE* and *MPI_THREAD_FUNNLED* allow for potential optimizations beyond *MPI_THREAD_SERIALIZED*. *MPI_THREAD_SINGLE* is for applications that will only use a single thread and *MPI_THREAD_FUNNELED* is for applications that will have multiple threads and only call MPI from a single thread. While these could theoretically be used to optimize memory operations, in practice MPI implementations will simply run using the *MPI_THREAD_SERIALIZED* mode.

2.2.3 One-Sided Communication

MPI 2.0 introduced remote memory access (RMA) to the specification. RMA is an implementation of one-sided communication and attempts to reduce the demand and overhead of MPI implementations by using requester data placement semantics. RMA transfers are done by calling put or get at a specified offset within a exposed memory window. These calls are typically non-blocking and are only guaranteed to be complete once a synchronization call is made. This matches closely with underlying network architectures and removes the need for MPI operations, such as matching. However, it places a larger burden on users to manage the distributed memory windows, who must ensure that the data location is known by both peers and prevent overlap between RMA transfers.

Several MPI benchmark suites support measuring MPI-3 RMA performance. The OSU Benchmark Suite from Ohio State University [84] supports several different measurements associated with MPI-3 RMA operations, including different window creation and synchronization methods. It also includes 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 [59] also has several benchmarks for measuring MPI-3 RMA performance and allows for measuring the impact of the different MPI thread levels, but it 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 [106]. 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 [105], and a proposal for endpoints in MPI [98] 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 [51] networking programming interface. Several of the issues with extracting more concurrency from the networking hardware and software stack were explored by Ibrahim et al. [57].

While both MPI RMA and multithreaded MPI have been explored in the past, the benchmarks described in Chapter 4 of this dissertation are the first to explore the combination of the two in depth.

2.3 MPI Matching

2.3.1 Performance Studies

Several papers have examined MPI message matching performance. For example, Barrett et al. [9] examines many-core matching rates. Underwood and Brightwell [108] introduce some microbenchmarks with the aim of analyzing the behavior of MPI implementation in the presence of long priority (PRQ) and unexpected (UMQ) message queues. Vetter and Yoo [113] analyze the scalability and performance characteristics of some scientific applications and show that communication overhead is one of the important limiting factor in some MPI applications.

Other studies have focused on evaluating message match list lengths for applications. For example, Keller and Graham [67] study the characteristics of UMQ of three MPI applications (GTC, LSMS and S3D). These characteristics include the size of UMQ, the required time for searching the UMQ and the length of time such messages spend in these queues. The evaluation results show that message queue matching is one of the scalability bottlenecks for some applications. On the other hand, others [19, 17, 18] investigate the impact of latency and message queue length on the performance of applications. Ferreira et al. [39] leverage simulation to analyze matching statistics. Klenk et al. [68] use traces of MPI applications and proxy apps to characterize matching performance, unexpected message rate, use of wild cards, and blocking vs non-blocking calls.

While MPI is the primary example of a matching algorithm, other matching and matching-like use cases exist. For example, Grant et al. [45, 89] explore a mechanism for the IWARP interface over RDMA that performs a matching-like step to issue a notification for the completion status of transfers.

2.3.2 Data Structures

With the approach of exascale, there have been new efforts to improve MPI matching. Newer approaches, like CH4 in MPICH, use more than one list, separating matching entries by communicator [101]. Open MPI [104] extends this by creating a separate list for each peer in the communicator and a wild card list. Each communicator has an array of linked lists that can be indexed by the communication peer. A search must check both elements in both the specific list for specified peer and the wild card list, compare the order of any matches and return the first match. The cost of this approach is memory overhead when dealing with a large number of ranks, which this dissertation will explore further in Chapter 6.

There have been several more complex proposals to accelerate the matching engine without incurring a memory penalty. Zounmevo and Afsahi [115] proposed a 4-dimensional matching engine that scales in both performance and memory. This approach attempts to avoid chunks of the match list by turning the search into a look-up in a 4-dimensional table.

Rodrigues et al. [91] explored the use of wide ALU operations to process multiple MPI match operations in a single ALU operation. Unlike the matching work presented in this dissertation, the study utilizes different data structures for a Processing-in-Memory processor.

Flajslik et al. [40] propose a hash-bucket approach. In this approach, the match lists are fixed-size hash buckets that use matching data as a key to separate the linked lists. The number of linked lists and the hash function are configurable parameters. The evaluation done in this paper compares this data structure with the linked list approach and shows that the proposed design with 256 bins significantly reduces the number of match attempts per message. Bayatpour et al. [14] extend the hash-table approach by creating a dynamic runtime approach to swap between hashing and traditional matching when appropriate. The selection of each approach is done at runtime. This approach shows up to a 2x speed-up.

In contrast to the message matching approaches that have been designed based on CPU architecture, Klenk et al. [69] introduce a new message matching algorithm taking advantage of GPU features. This algorithm is designed with two phases, scan and reduce to leverage the available threads on a GPU. The evaluation results show that the proposed algorithm could provide 10x to 80x improvement in matching rate. This approach has two major caveats: it is dependent on the existence of a GPU-style accelerator card and having a sufficiently long matching engine to make use of all the threads in a GPU warp.

The matching approaches presented in Chapters 5 and 6 differ from those described above in that they utilize the CPU cache, SIMD vector unit, and threading features to accelerate matching. In these techniques, memory overheads are directly dependent on the number of match list entries, unlike prior work like Open MPI where overheads are related to the number of processes in a job. These techniques are also likely to be compatible with other approaches. For instance, utilizing the vectorization technique described in Chapter 5 could reduce the number of bins required for a performant hash table structure, and the parallel matching technique described in Chapter 6 can be used to parallelize the hash tables independently.

2.3.3 Matching Hardware

Most interconnects used in large-scale HPC systems today incorporate some offload capability. IBM's Blue Gene/Q [24] and PERCS [5] networks both support offloading of MPI collective operations. Likewise, Cray's Gemini [2] and Aries [36] networks support MPI collective communication offload. With the ConnectX-2 [42] product, InfiniBand network adapters from Mellanox also began supporting MPI collective communication offload. However, these networks do not offload the more complex tag matching and queue traversal mechanisms needed to handle MPI point-to-point communication operations. These networks rely on the MPI process running on host processors for this capability. Techniques like Cray's Core Specialization [87] provide a mechanism for dedicating host processor cores to running an MPI progress thread. This technique has also been used to improve the performance of TCP/IP protocol stack processing [95]. Many offload schemes also include collective offloading, which requires a limited version of matching, often at both the NIC and switch level, to support communication within a collective [42, 16, 109, 92, 93, 58, 23].

MPI match list processing has been proposed as an addition to offload hardware, with the Portals networking API [10] enabling such offloads. This includes studies into hardware designs that can efficiently perform MPI message matching with wild cards [110]. There have been a number of implementations of Portals including a reference implementation [86], a simulated hardware design [100], and a commercial product [28]. MPICH implementations are in the process of adding a new channel CH4 in order to help address scalability issues and better handle hardware-supported message matching [47].

Recently hardware has been released with offloaded message matching. This includes Intel's OmniPath PSM2 [15] with a subset of the network cards capable of offloading matching, and Atos-Bull's BXI interconnect [28], which performs MPI-style message matching entirely in hardware. Such solutions will not benefit from software MPI matching list improvements. However, in the PSM2 case, there are still models that use the software matching implementation. MPI message matching hardware is typically complex and large in terms of overall die area on the custom ASICs used.

2.4 MPI for Modern Architectures

2.4.1 Multi-threading

Another relevant area of research to this dissertation is HPC communication library support for hybrid parallel programing models, such as MPI+X. These programing models reduce the number of MPI processes, while maintaining the same level intra-node parallelism. Additionally, these models can take advantage of thread level parallelism features that are unavailable to MPI-everywhere applications. For instance, most MPI-everywhere applications do not utilize hardware thread contexts to avoid resource contention during highly synchronous code paths, such as communication phases.

Recently, $MPI\_THREAD\_MULTIPLE$ has been explored in depth. Gropp et al. [48] discus the implications and requirements of $MPI\_THREAD\_MULTIPLE$ to MPI implementations. There are a number of non-trival thread safety details, including how to support receiving messages of an unknown size [55]. In particular, using `MPI_Probe` to get the size of a message before posting a receive creates a critical section that encompasses the two calls.

There has been work to improve thread-safe MPI implementations. While most current implementations have used a global synchronous lock, however Balaji et al. [8] apply fine grained locking to MPI and show a performance improvement over global locking. By reducing the critical sections in MPI, they reduce the time spent waiting on locks. To further mitigate the cost of locking, Amer et al. [3] propose a new synchronization arbitration to provide better fairness guarantees. This avoids potential starvation issues where one thread can be constantly blocked by other threads to the point if it falls behind in computation. Vaidyanathan et al. [111] propose implementing a software offload mechanism to support thread multiple. They dedicate a core to do MPI processing and intercept the MPI calls to have a dedicated progress thread execute them. This is particularly useful for non-blocking calls as the calling thread can queue up a request and do other computation while the offload thread progresses the communication.

There has been work to explore how to change the MPI programing model to better suit hybrid applications on highly parallel architectures. Stark et al. [99] and Barrett et al. [12] focus on exploring MPI+X task models that allows over-decomposition of a problem in to a number of tasks that can then can be run by different MPI processes. Similarly, FG-MPI [63] is a execution model that extends the MPI to enable task based parallelism at a process level. Endpoints [31, 98] is a proposal to allow MPI processes to separate some of the MPI state and processing by turning threads into addressable endpoints with a sub-rank. This could allow threads to access MPI in parallel for the parts of the processing that have been isolated. Finepoints [46] is another proposal that creates a new method of messaging that allows threads to specify a portion of a message in an MPI call. The underlying implementation can choose to aggregate and send this data to the receiver or can split the data transfers. This allows a process to complete a subset of the transfer while waiting on other threads to complete.

2.4.2 Power and Energy

Limits on power usage have the potential to reduce HPC communication library performance. They could, for example, throttle systems to a lower processing speed, slowing network and middleware processing. Complicating this problem, today's hardware can have significant variability in power usage. This has been an active area of research in the past few years, as power caps become increasingly likely for next generation systems. The PowerAPI [70, 44] has been a community-driven effort to create a standard interface for handling power measurements and constraints in HPC systems. Grant et al. [43] examines application power characteristics when run on the Xeon Phi and various frequencies. Leon et al. [72, 73] examine power characteristics of applications at a finer granularity, focusing on how different optimizations of a single application, LULESH [65], effect its power and performance characteristics. For the purposes of this dissertation, it is important to note that power and energy efficiency of the interconnect has become an important consideration for large-scale data centers [75, 21] and HPC systems [54, 107], providing a new perspective on the offload-versus-onload debate. Other works have previously studied the impact that power-efficient cores have on MPI message rate [11].

Chapter 3 of this dissertation focuses on the power and performance tradeoffs of different network hardware. This work controlled for variations in processor power usage by ensuring the only device changed between tests was the network card. With deviations in power usage of components, this also has the potential to slow processes in an MPI job unevenly, which causes delays at synchronizations points. These challenges also affect the viability of certain solutions to improve HPC communication library performance. Complex dedicated network processing hardware may become undesirable as it increases the power requirements of each node. Other techniques such as using the CPU for network processing while the main computation is being performed on an accelerator card and increasing clock frequency during serial execution have power usage implications that may render them unviable.

2.5 Other HPC Programming Models

2.5.1 One-Sided Programming

One-sided programming models have been proposed as an alternative to MPI as their semantics allow these models to avoid some of challenges that MPI faces for many core systems. These models try to improve performance by mapping directly to the underlying RDMA protocol and by avoiding the processing related to data placement, as the placement is determined by the sender. There are generally two types of one-sided HPC communication libraries: globally addressed and message based. GASNet and UPC are of the former category, allowing access to variables within a global address space. This has some semantic advantages but can be difficult for users to optimize because it abstracts data locality information. MPI-RMA and SHMEM are messaging based one-sided communication interfaces.

This requires explicit data transfers to the destination process but does not provide complex global address space abstractions.

SHMEM

SHMEM and the open source OpenSHMEM are particularly relevant to this dissertation. SHMEM is an HPC communication library focused on providing one-sided communication semantics, similar to MPI's RMA. In addition to one-sided point-to-point communication, modern implementations also support a variety of collective communication calls. The first implementation of SHMEM was introduced in 1993, four years before MPI-RMA appeared in MPI 2.0. Because the semantics are similar to MPI's RMA, Chapter 4 explores hybrid SHMEM in addition to hybrid MPI's RMA.

Luecke et al. [77] compared the performance of SHMEM with MPI-2 RMA on an SGI Origin 2000 and Cray T3E system. Unlike this work, they used a single threaded approach and the MPI-2 RMA interface was the only one available. Since that time, significant improvements have been made to the MPI RMA interfaces for MPI-3, and OpenSHMEM [22] has emerged as a standard, with matching implementations.

OpenSHMEM benchmarking has been explored before with work from Pophale et al. [85]. Further work has been done looking at OpenSHMEM on specific networks [61]. Several works have examined hybrid MPI+OpenSHMEM codes for mini-applications [74] and Graph500 [60].

Performance evaluation and benchmarks for the original SHMEM interface (inspiration for the OpenSHMEM interface) have been done since the inception of SHMEM in the early 1990s [38]. Evaluations on shared memory systems [76, 35] have been popular SHMEM performance comparison targets. Comparisons of performance between SHMEM and MPI for interfaces like Portals [108], have been done as well as MPI/SHMEM comparisons [56] even implementations of MPI over SHMEM [20]. Some work has also been done in implementing a SHMEM interface using MPI-3 RMA operations [50]. Other studies have compared MPI, SHMEM, and UPC [78]. Remote memory approaches, such as ARMCI [83], have also addressed remote direct memory communication.

Chapter 3 in this dissertation explores SHMEM as an alternative for the multithreaded one-sided programming modes by converting the RMA-MT benchmark suite and is one of the first studies explore the performance of multi-threaded SHMEM on an application level.

2.5.2 Many Task Programming

In addition to traditional threading models, such as Pthreads [82] and OpenMP [27], task based parallelism models are proposed. Task based programing models define parallelism in terms of tasks, often with dependencies, that can be run in parallel on a system. Often these models map better to problem over-decomposition where parallel work is split into more parts than can be processed simultaneously. This allows the system to utilize work balancing and overlapping communication with computation. Examples include Legion [13] and Charm++ [62].

2.6 Chapter Summary

This chapter surveyed the state-of-the art of HPC communications libraries in terms of MPI matching, MPI for modern architectures, and alternative programing models. All of these were presented in the context of many core processors. In general, while the subjects in this dissertation, such as matching, $MPI\_THREAD\_MULTIPLE$, vectorization, caching, and one-sided, have been studied before, these studies have examined these in isolation. Chapter 4 examines the combination of one-sided communication and multithreading. Chapters 5 and 6 examine matching in combination with caching, vectorization, and multithreading.

Chapter 3

The Power and Performance Effects of Many Core Processors

3.1 Introduction

As described in Chapter 1, many-core processors have a potentially dramatic impact on HPC communication performance. Little research has quantified this impact, however, and it is expected to depend on the complexity of the network hardware. Additionally, quantitive analysis needs to address both the power and performance concerns of exascale machines. This chapter quantifies the performance impact of many-core CPU architectures on communication libraries. It also explores the current state of the art for HPC communication library performance on many-core HPC platforms. This includes a study of the effects of processor speed changes on the communication performance of both onload and offload HPC systems. This work was done collaboratively with Ryan Grant; Ryan performed the fine grained evaluation with the micro-benchmarks and I conducted the in-depth application study. This work has been previously published in the proceedings of the International Conference on Cluster Computing [33].

The contributions of this chapter are the following:

- An evaluation of the impact of many-core processors on modern communication systems including onload and offload network architectures.
- A study of the power and performance trade-offs of onload and offload InfiniBand network architectures.
- An in-depth study of the effects of CPU speed on HPC communication performance in an application context.

This chapter is structured as follows: We first discuss our experimental setup and methodology in Section 3.2 and present and analyze the results of these experiments in Section 3.3. Section 3.4 presents an analysis of these results and the implications for HPC communication libraries on many-core systems. Finally, we present our conclusions and describe directions for future work in Section 3.5.

3.2 Experimental Methodology and Setup

To evaluate the performance and power trade-offs between onload and offload networking approaches, we conducted experiments with both Mellanox offload and QLogic onload InfiniBand cards. These cards were placed in systems instrumented for power collection, and host CPU power consumption was controlled to understand the impact of CPU speed on network performance and power consumption in different applications. In the remainder of this section, we provide additional details on our experimental setup, the hardware system on which these results were gathered, and the microbenchmarks and applications used.

3.2.1 Hardware and Data Collection Setup

The evaluation of the onloaded vs. offloaded networking approaches was performed on 4 nodes of a cluster at Sandia National Laboratories. Each node has a 3.8 GHz AMD Fusion APU, 16 GB of memory, and Linux kernel version 2.6.32 (RHEL 6). For onload experiments, we installed a QLogic 4X QDR InfiniBand HCA in each node, while we used a Mellanox ConnectX-3 4X QDR InfiniBand HCA for offload experiments. In both onload and offload cases, a Qlogic 12200 36-port InfiniBand switch connected the InfiniBand HCAs. The APU has CPU and GPU components; we used only the CPU.

Power measurements for the experiments were collected using the PowerInsight [71] measurement system installed in the cluster. PowerInsight is an out-of-band measurement device that collects fine grained samples for multiple system components through the use of a mother measurement board and risers on system components. They enable the in-line reading of system power on a per component basis without impacting the performance or power consumption of the node. The components PowerInsight measures include CPU, memory, motherboard, network cards, and fans. All power information output by PowerInsight uses a separate out-of-band network to deliver the information to a central collection node that was not participating in the testing.

For the purposes of this study, we collected power data at 10 Hz for NetPIPE microbenchmarks and 1 Hz for the application benchmarks. To accurately represent their power usage, the microbenchmarks required a higher resolution. However, as the application benchmarks had runtimes that were all greater than 5 minutes, a lower resolution was sufficient for the comparison.

3.2.2 Benchmarks and Applications

To compare the onloaded vs. offloaded networking approaches, we analyzed the performance and power comparisons on benchmarks and applications. In particular, we compared onloaded and offloaded runs in the MILC application [26] and the LULESH [66] and NetPIPE [96] benchmarks. Furthermore, we ran profiling runs using mpiP to determine why these applications react to the network cards differently.

The NetPIPE microbenchmark suite is a tool designed to test the bandwidth and latency of a network. We ran the streaming, streaming without cache effects, and send-recv ping-pong tests over different message sizes ranging from two bytes to one megabyte. The MIMD Lattice Computation (MILC) application was the first application benchmark we used [26]. It was developed to study quantum chromodynamics and uses four dimensional lattice computation using a halo exchange communication pattern. We used an input deck based on the weak scaling NERSC 6 acceptance benchmarks [4]. In particular, each node has lattice of size 8x8x8x9. The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) application is the second large benchmark we used [65]. LULESH is designed to be a representative application for larger hydrodynamics codes. For all of our tests, we ran a 120^3 problem for 130 iterations. There is a constraint on the number of MPI ranks used for this code; it has to equal a cube of an integer. Because of this, we fixed the number of MPI ranks at 8, adding an extra OMP thread to each rank in the four node case.

mpiP is a lightweight profiling layer for MPI. When running, mpiP collects statistics including number of calls, type of calls, and time spent in function calls. This information is separated by each call written in the source code. We used this information to analyze communication patterns.

All three benchmarks were run three times for combinations of the following variables: InfiniBand card, number of nodes, and CPU frequency. We used the InfiniBand cards mentioned in section 3.2.1 to test both onload and offload. The number of nodes varied between 1, 2, and 4 for MILC, 2 and 4 for LULESH, and was not a variable for NetPIPE. The different number of nodes allowed us to compare the performance differece with variying levels of inter-node communication. The CPU frequency was modified using DVFS to 1.4GHz, 1.9GHz, 2.4GHz, 2.9GHz, 3.4GHz, and 3.8GHz. This allowed us to relate this work to a range of processors, by changing the performance from consumer-grade to the equivalent of server-grade and many-core cores. We collected the overall runtime and power statistics of these applications as well as MPI profiling information on a couple of separate runs.

3.3 Experimental Results

3.3.1 Microbenchmark Evaluation

The NetPIPE microbenchmarks [96] were used to examine the impact CPU frequency has on both the power draw and performance of the different networking approaches. Figures 3.1 and 3.2 show the stream bandwidths along with the power consumption of both onloaded and offloaded networks. Aside from the obvious protocol switching points (MPI eager to rendezvous) causing plateaus and in some cases dips in performance between messages sizes, the important observation to make from these figures is the spread in performance between the highest CPU frequencies and the lowest. The onloaded method loses some performance when CPU frequency is lowered, resulting in an near halving of bandwidth between the 3.8GHz and 1.4 GHz frequencies. For the offloaded network, the reduction in CPU frequency impacts network performance by a much smaller degree. The only noticeable difference in behavior occurs when the lowest CPU frequency is used. There is some variance in the bandwidth curve than the other scaling points, which can be observed at the 256KiB message size. This suggests the microbenchmark may not be able to keep up with the network events at this speed. The performance gaps between the CPU frequencies remains relatively similar in terms of percentage of performance loss for all message sizes, including the smaller message size results.

Removing caching effects from the results as shown in Figures 3.3 and 3.4 show has little impact on the offloaded case, except for a slightly reduced throughput. This results in a smaller gap between the slowest speed (1.4GHz) and the other clock speeds. The impact

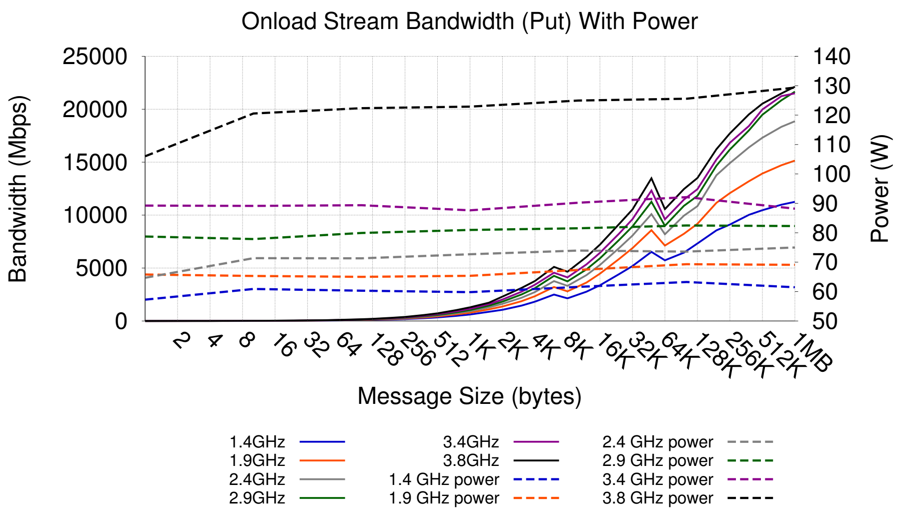

Figure 3.1: Onload stream with varying CPU frequencies

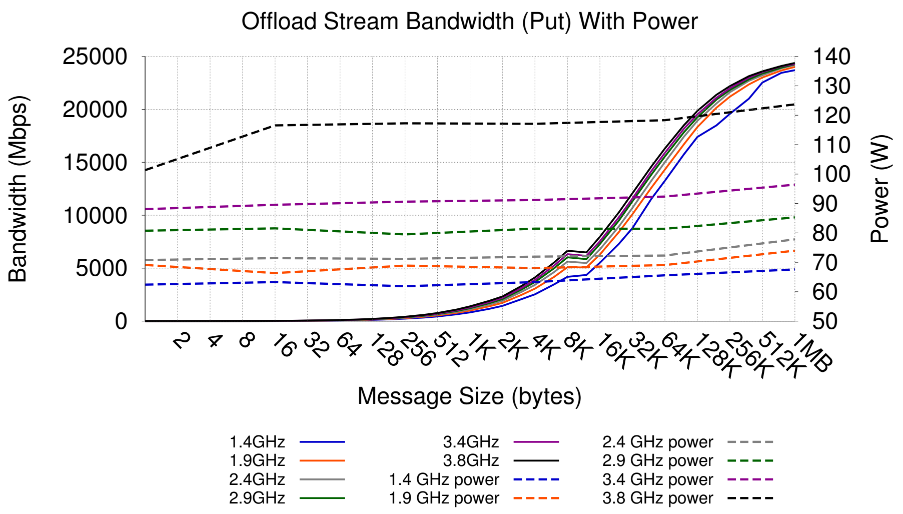

Figure 3.2: Offload stream with varying CPU frequencies

on the onloaded case is similar for large messages. However, there are differences for small and medium sized messages. The drop occurring after 8KiB message sizes is caused by the eager-rendezvous protocol switch-over in MPI. The drop occurring at 64KiB message sizes is caused by the virtual maximum transmission unit (VMTU) maximum of 64KiB, necessitating multiple calls to the onloaded networking stack.

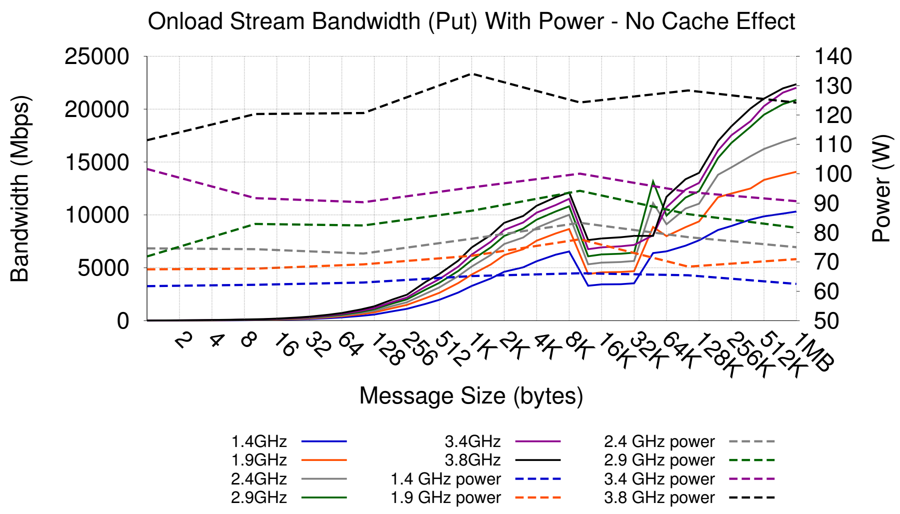

Figure 3.3: Onload stream with varying CPU frequencies without cache effects

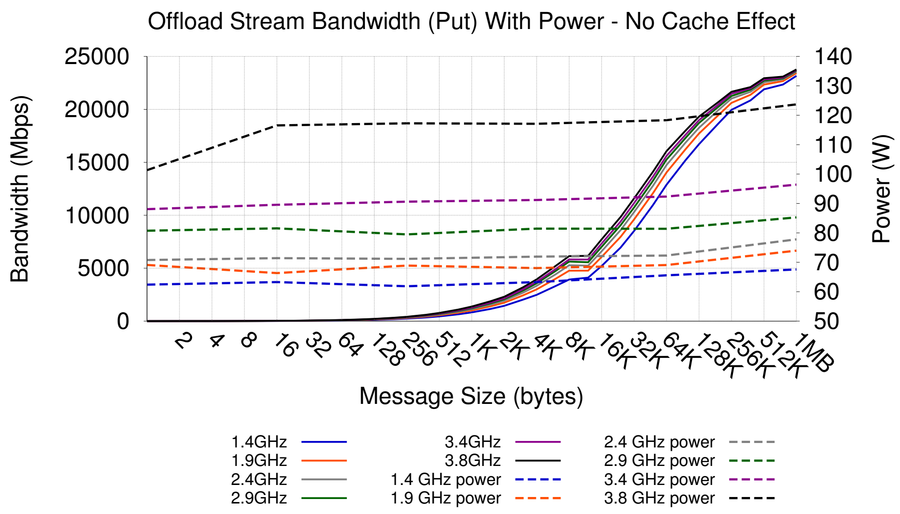

Figure 3.4: Offload stream with varying CPU frequencies without cache effects

Examining send-recv performance with bi-directional ping pong (as opposed to the previous unidirectional streams), Figures 3.5 and 3.6 show that the results are similar to the stream results with cache effects. The increase in throughput is due to the bi-directional nature of the test, but generally aligns with the unidirectional results, in that they are reasonably within twice of the unidirectional throughput.

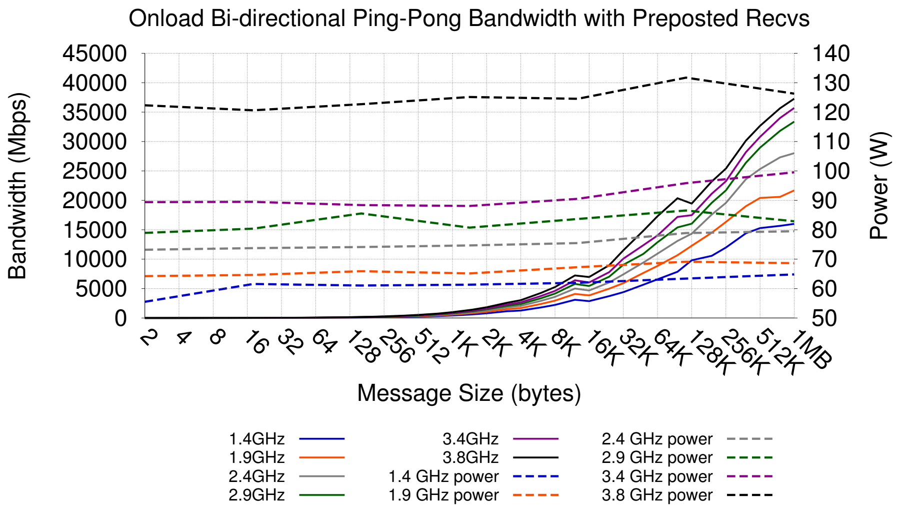

Figure 3.5: Onload bi-directional ping-pong with send-recv and preposted recvs

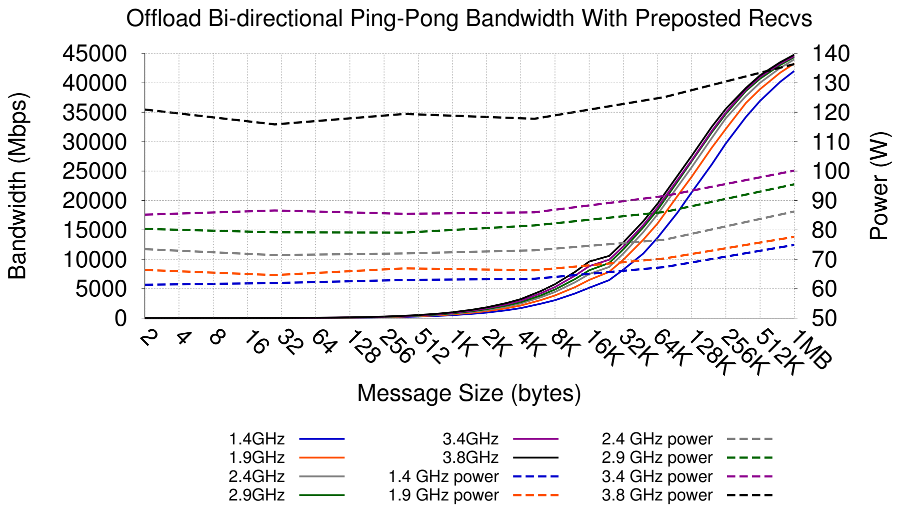

Figure 3.6: Offload bi-directional ping-pong with send-recv and preposted recvs

A key observation from these results is the relatively small trade-off in throughput performance from transitioning between CPU frequencies for both onload and offload at 2.9GHz and higher. This observation indicates that, at this speed and above, the network bottleneck is no longer protocol processing. At this point, the bandwidth and latency is more impacted by the limits of the hardware. We concentrate on the results including cache effects for the purpose of this analysis, but the percentages are similar for the case in which cache effects have been removed. For the onload case, 36.4% of the power consumption can be saved while only losing 2.5% of throughput, while for the offloaded case 22.5% of power can be saved while only impacting performance by 0.5%. The offloaded network provides better results in scaling frequency back below the 2.9GHz level, providing power consumption savings of approximately 30.5% while impacting performance by only 1.5% when switching from a 3.8GHz clock rate to 1.9GHz. For the onloaded case, this is impractical, as using a lower frequency such at 1.9GHz would result in a performance loss of 35.1%. This emphasizes the potential issues that may arise when using many-core systems with slower and less powerful compute cores. It also highlights that if such network onload approaches are to be practical on future many-core systems, parallelism for communication will be a key component in achieving performant network throughput.

Finally, Figures 3.7 and 3.8 show the latency impacts from slower CPU frequencies on the onloading and offloading approaches. The latency penalties associated with lower CPU frequencies occur for both onloaded and offloaded networking. However, the offloaded networking approach leads to convergence of latencies for successively lowering CPU frequencies at smaller message sizes, and all CPU frequencies eventually converge at 1MiB message sizes. For small messages under 512 bytes, the offloaded networking approach has a flat latency curve, while the onloaded case has a upward slope at smaller message sizes.

3.3.2 Application Benchmark Evaluation

We used application benchmarks to examine the performance and power tradeoffs in realistic workloads. Using MILC and LULESH, we measured the runtime and power usage at different node counts and CPU speeds to compare onload and offload. Then we ran MPI profiling tests to compare the results of the two applications. It should be noted that there was not much variance in either the runtime or power of the application benchmarks; the standard deviations of 80% of the runs were below 1% of the mean, only 2.5% of the runs had a

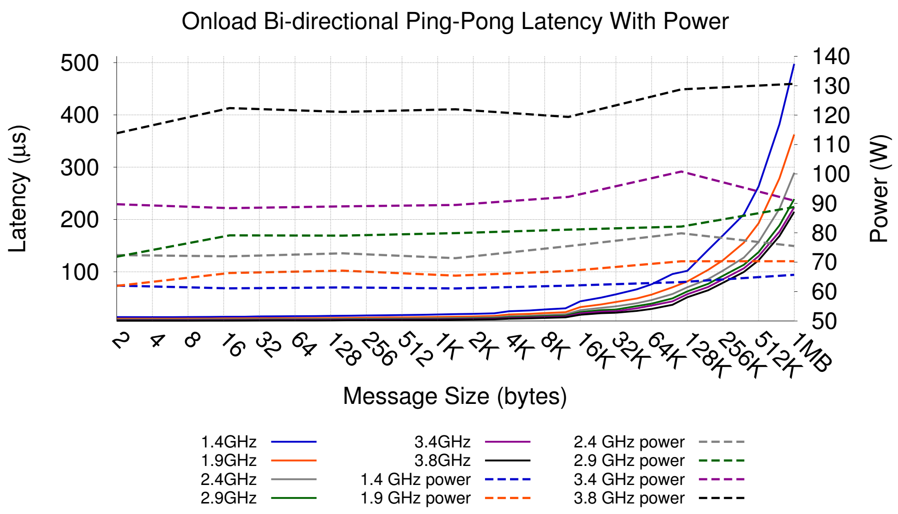

Figure 3.7: Onload bi-directional ping-pong latency

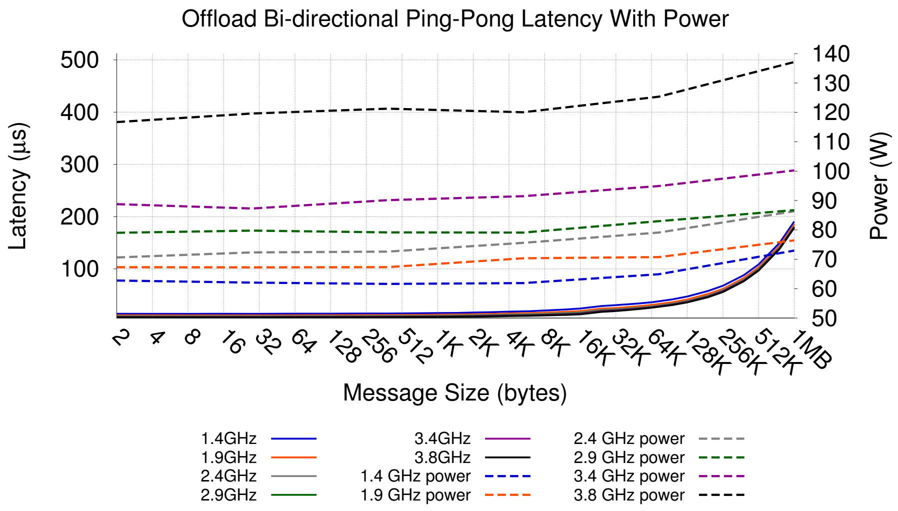

Figure 3.8: Offload bi-directional ping-pong latency

standard deviation greater than 2%, and the maximum standard deviation was 2.81%.

Runtimes and Power

Figures 3.9 and 3.10 show the power and performance results of MILC. The tests show the runtime change over the CPU frequency for each of the different node counts. Because the problem size is scaled to the number of nodes, the runtime increases when adding nodes, for instance the four node, onloaded case at 1.4GHz takes over 17 minutes more than the one node, onloaded case. This can be attributed to a combination of the extra computation at the boundary and the communication time between the four nodes.

In all of the cases we measured, the offload version takes less time than its onload counterpart. For four nodes, it ranges from a 7.7% to a 10.6% difference between the two, for two nodes, the range is from 5.3% and 7.8%, and even the single node case had a small but consistent performance benefit, ranging from 0.9% to 3.1%. These differences all steadily decline when we increase the clock speed. This shows that while the offload cards have a significant effect on most of the test cases, they have a more significant performance impact on low frequency cores. The power usage of MILC has less significant differences. The offload HCAs used between 1.1% and 3.2% more power than their onload counterpart.

Figures 3.11 and 3.12 show the power and performance results of LULESH. The tradeoffs are less distinctive here. The difference between runtime and power usage fluctuate around 0%. The change in performance ranges from 0.6% in favor of offload and 0.6% in favor of onload. The power differences are similarly low. Since LULESH is not significantly affected by the InfiniBand card used, it interestingly contrasts with MILC.

It is important to note that the impact of decreasing CPU frequency is significant to the performance of the compute portion of the proxy-applications under study. The goal of these experiments is to examine systems known to have little process variation induced performance impact while limiting the differences between the systems to solely the networking hardware. By using the same switch for both onload and offload approaches, we have isolated other potential performance and power related impacts due to factors not of interest to this study. In order to confirm the results, we conducted some MILC experiments on the exact same hardware with the network hardware swapped. These experiments confirmed that process variation between the servers used for the experiments was negligible.

Profiling the Applications

An interesting trend emerges when comparing these results. Offloading has a measurable performance benefit on MILC but LULESH is nearly indistinguishable from the onloaded environment. To understand the difference between these two applications, we profiled the two applications using mpiP [112]. These tests were run at 3.8GHz on four nodes with the QLogic onloaded InfiniBand cards. The pieces of information we gathered are percentage of time the application spent in MPI, the distribution of time within MPI, and the number and distribution of function calls.

Both applications spent a fair amount of time in MPI; MILC spent 15% of its runtime in MPI, while LULESH spent 12%. However, the number of MPI calls was significantly different. MILC called MPI 4,011,216 times over its runtime, which ran for 1382.45 seconds on average. Comparatively, LULESH made substantially fewer calls, with 42,904 MPI calls over it's runtime, which ran for an average of 81.64 seconds. This equates out to 2901.5 MPI calls per second for MILC and 525.5 MPI calls per second for LULESH. Table 3.1 shows the distribution of MPI calls to specific functions. The notable differences are MILC has larger percentage of wait calls and the lower percentage of Allreduce calls, compared to LULESH.

Table 3.2 shows the distribution of time within MPI calls. The differences here are stark; MILC spends a reasonable amount of time in Allreduce, Isend, and Wait; however, LULESH spends almost no time in Isend, mainly spending time in Allreduce and Wait. The amount of Irecv and Isend calls in MILC illustrates that it is performing more significant point-to-point communication operations than LULESH. MILC is a memory bound code that can be sensitive to network performance, as such it is not surprising that the performance of MILC is impacted by lowering CPU frequency in an onloaded networking situation. LULESH

Call	MILC	LULESH
Allreduce	1.15%	2.22%
Irecv	24.71%	30.34%
Isend	24.71%	30.34%
Wait	49.42%	30.34%
Waitall	0.00%	6.73%
Other	0.00%	0.04%

Table 3.1: Distribution of MPI Calls

Call	MILC	LULESH
Allreduce	29.86%	42.68%
Irecv	1.71%	0 %
Isend	13.99%	0.2%
Wait	54.43%	54.02%
Other	0.01%	0.4%

Table 3.2: Distribution of Time Within MPI on a Standard Run

is primarily dependent on the performance of Allreduce for good networking performance. These results show that Allreduce performance does not change significantly between the onloaded and offloaded networking approaches.

These results indicate that the performance benefit of offloading is seen primarily in codes that have a large number of small communication calls, rather than a few larger calls. This indicates that the network protocol processing overhead is more a function of the number of messages, rather than the amount of data being sent. MILC and LULESH spent similar percentages of their runtimes in MPI, but MILC relies on a large number of small point to point and collective operations and LULESH focuses on small number of large collectives that make up most of its time in MPI.

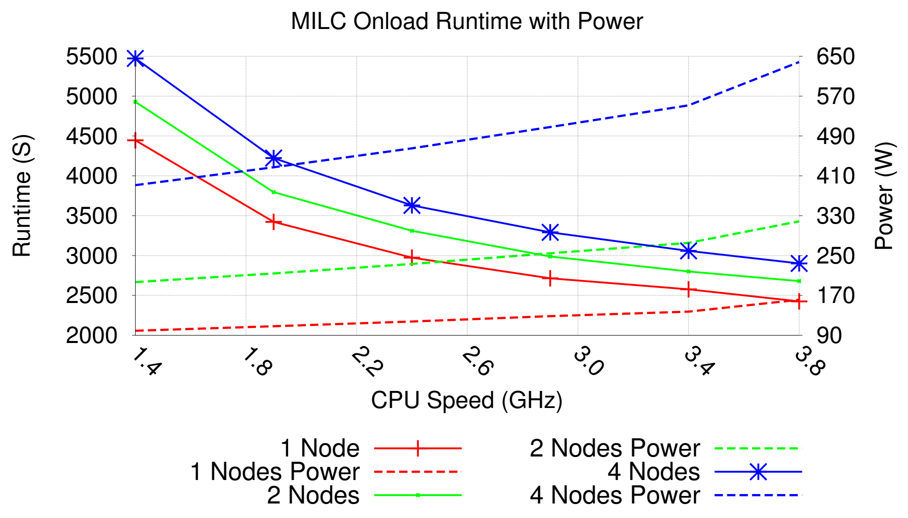

Figure 3.9: Onload runs of the MILC application

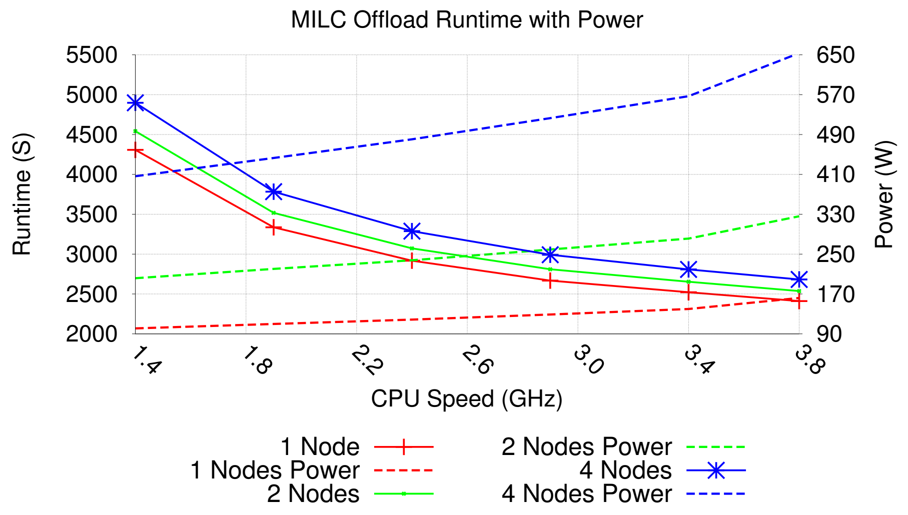

Figure 3.10: Offload runs of the MILC application

3.4 Analysis

The work in this chapter shows that optimizing and adapting HPC communication libraries to many-core processors is important to the next generation of applications running on exascale supercomputers. Figure 3.1 shows that reduced clock speeds have a large negative

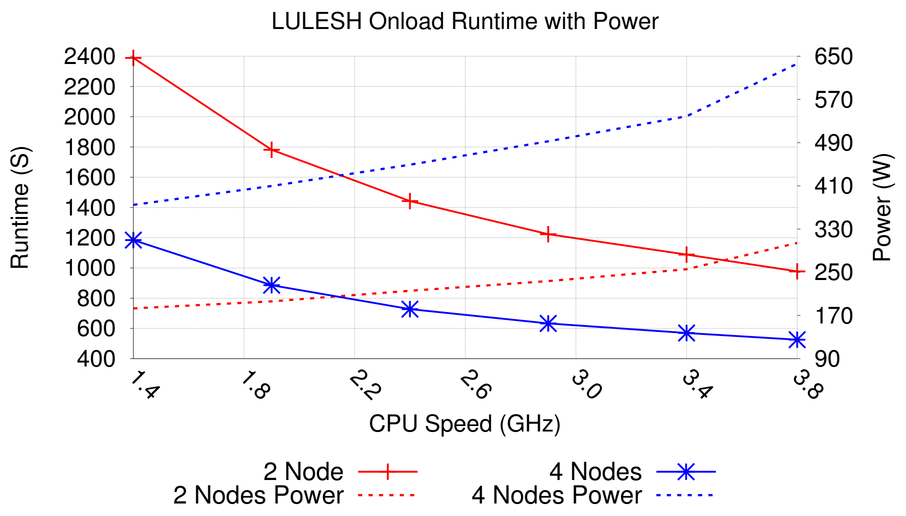

Figure 3.11: Onload runs of the LULESH application

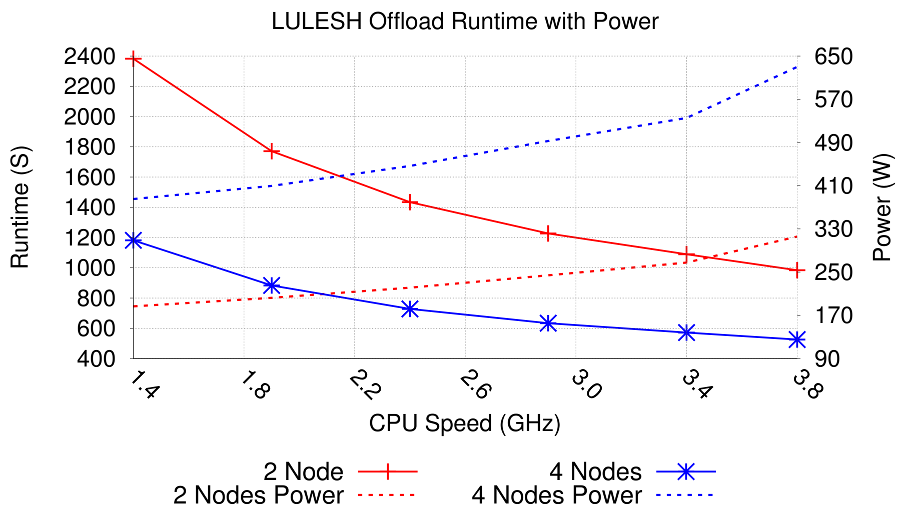

Figure 3.12: Offload runs of the LULESH application

impact on onload networks. Offload is an important tool to help mitigate these increased costs. This does not account for more complex communication patterns where HPC communication libraries become more processing intensive.

Offloaded networks architectures may reduce this impact. Figure 3.2 shows that Infiniband level offload mitigates the effect on microbenchmarks. Due to the range of different offload networks, each doing a different amount of the network processing on the dedicated ASIC, it is difficult to generalize how much of the reduction will be. It is also difficult to quantify the power trade-offs of future hardware that does more complex network offload, like Atos's BXI network. It is also unclear how offload networks will perform in an MPI-everywhere legacy application. The offload network card would have to maintain the HPC communication library state for each process on the node and be able to switch between them.

Figures 3.9, 3.10, 3.11, and 3.12 demonstrate the importance of evaluating the power usage of hardware techniques for different applications. While the offload cards used in this study only increase power usage by 2%, it is likely that more complex hardware will have a larger power draw. For applications like MILC, that call into MPI often, there is likely a significant benefit. The results we saw for MILC showed up to a 10.6% improvement in performance, for a small 2% increase in power. This indicates an application where hardware solutions to HPC communication library performance are beneficial. A more complex offload approach would likely increase power and performance. Alternatively, LULESH does not see a performance benefit from offload. Increasing the offload complexity in this case will likely only increase power usage. This could be a limiting factor for the use of offload in an general purpose exascale machine with a power cap. This is indicative of the importance of software-based onload and partial offload solutions to HPC communication library performance. They don't increase the maximum power consumption in a node and thus might be a relevant technique to maximize throughput on power capped systems.

This work shows an estimate of the impact that many-core processors can have on HPC communication libraries. It is important to note that this study, while an estimate, is likely underestimating the impacts. The use of DVFS provides an environment that is similar to the reduced speed of many-core systems, but does not account for the negative performance effects of the reduced complexity. With this in mind, many-core processes will have a large and quantifiable impact on onload network performance, as shown in the micro-benchmark results. This can have significant application performance implications, as seen in MILC, if there is frequent communication. As the amount of communication calls we see is likely to grow, as application begin to use fine grained messaging patterns to support communication and computation overlap, multi-threading, and asynchronous algorithms.

3.5 Conclusions

The key contribution of this study is an analytic, quantitative study of the tradeoffs of network onload and offload. Using PowerInsight and DVFS, we examined network processing performance in detail under a number of different conditions. In examining the differences between onloaded and offloaded networks for varying host CPU frequencies, an offloaded networking approach provides approximately equivalent or superior performance at lower frequencies.

The microbenchmark results clearly illustrate the potential benefits of network offloading, with power savings in the 20.5% range with only 0.5% performance loss and good performance down to 1.5% performance loss and power reductions of 30.5%. While onloading can reap greater power consumption drops, performance at the 1.4GHz level shows that onloading results in a loss of over half of the available throughput at higher CPU frequencies. This demonstrates the tradeoffs in single thread communications performance that would occur on systems with lowered CPU frequencies, and can reasonably be expected to be even lower if more simplified little-cores are used.

While the high-performance computing networking community has pre-supposed that such outcomes were likely, this is the first, to the authors' knowledge, analytical evaluation of the performance difference. As such, these results provide the foundation for further evaluations of the merits of onload versus offload for next generation systems. For example, these results clearly demonstrate that single-thread performance of onloaded networking solutions are restrictive in emerging many-core architectures. While multi-threaded approaches could alleviate some of the negative performance implications that this study exposes, current multi-threaded MPI implementations perform worse than their single-threaded counter parts and often involve course-grained, high-contention locks.

Chapter 4

Next Generation Parallel Programming Models

4.1 Introduction

As described in Chapter 1, one-sided communication and multithreading are programming paradigms proposed for exascale. This chapter addresses the research challenge of evaluating new parallel programing techniques on many-core architectures.

In this chapter, I present the RMA-MT benchmark suite and three studies that utilize it. I developed the RMA-MT suite collaboratively with Taylor Groves and Ryan Grant. Taylor was responsible for the latency and bandwidth benchmarks, Ryan converted HPCCG, and I converted message rate and the other two mini-applications. The analysis and writing for the initial study was done collaboratively between the three of us. The work has been previously published in the proceedings of CCGrid [34]. Section 4.7 presents collaborative work with Hans Weeks where we translated the benchmarks to Cray's SHMEM programming system. This work was previously published in the proceedings of the OpenSHMEM workshop [114].

The contributions of this chapter are the following:

- A new suite of benchmarks to explore the performance and correctness of MPI RMA in multi-threaded modes.
- A study of RMA-MT performance at small scale on unoptimized implementations.
- A study using the benchmarks modified to explore multithreaded SHMEM implementations.
- A study of RMA-MT performance on an optimized MPI implementation.

This chapter is structured as follows: Section 4.2 discusses the miniapps and microbenchmarks in depth, detailing their design and the considerations taken into account during the design process. Section 4.3 expounds on our experimental design and environment for running the evaluation section of this paper. Sections 4.4 and 4.5 show the results of testing with the micro-benchmarks on a large production system. Section 4.6 presents the an application study of RMA-MT with an assessment of three mini-apps: HPCCG, miniFE, and miniMD. Section 4.7 presents the results of the multithreaded SHMEM study. Section 4.8 presents an analysis of the impact of one-sided communication performance on optimizing HPC communication libraries for modern processors. Finally, Section 4.9 draws conclusions and discusses plans for future work.

4.2 Benchmarks and Mini-applications

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.

4.2.1 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 operations due to the overhead of synchronization and that latency tests measure the round trip time of a single message. 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 [32]. Single threaded, two-sided versions of the SMB's have been used in past work [9, 11]. 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 recvs are posted. For the RMA-MT versions of the halo exchange tests, only one version was needed to map to RMA, since RMA doesn't have an unexpected message equivalent.

4.2.2 Miniapps

This subsection presents the modifications made to a subset the Mantevo Suite [52]. 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 ranks 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.

4.3 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 used many hundreds of iterations within the test. Each test was run 10 times. The results presented are the average of those 10 runs in each figure in Sections 4.4 and 4.5. All figures shown include vertical 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.

4.4 Single Threaded Results

Single threaded comparisons between one-sided and two-sided communication in MPI have been explored before. We primarily include the results of figures 4.1-4.6 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

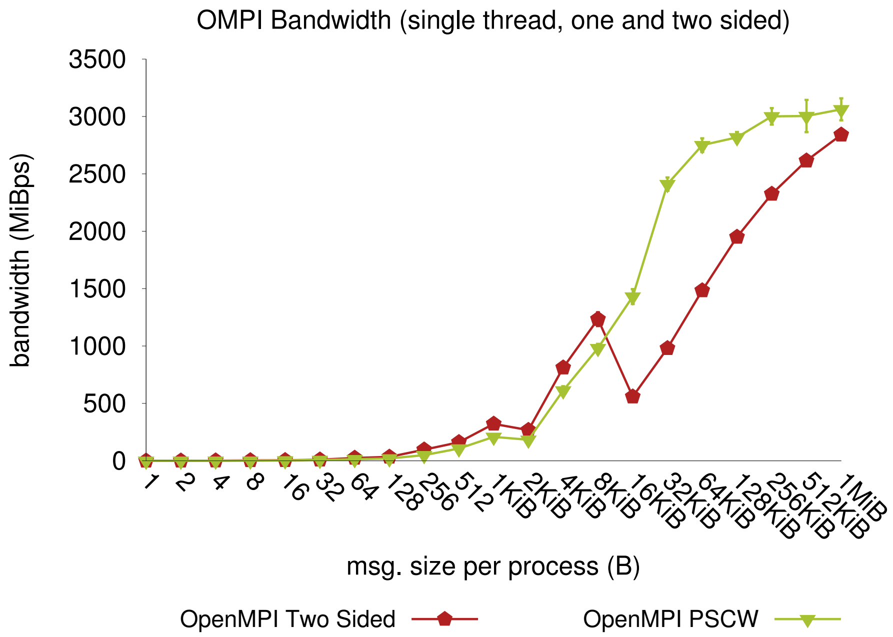

Figure 4.1: Single threaded one-sided and two sided bandwidth Open MPI

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 (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 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 [106], which lack samples between 1 and 16

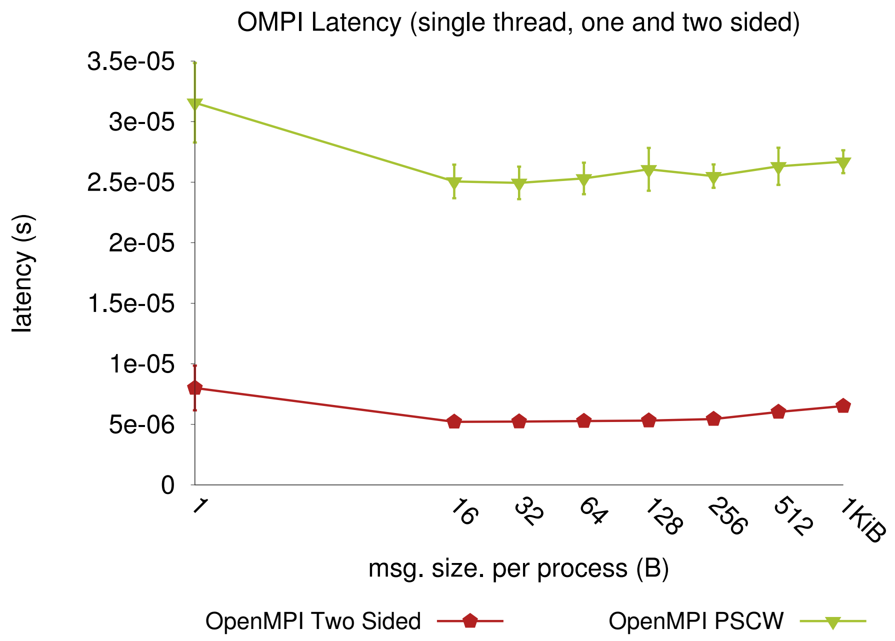

Figure 4.2: Single threaded one-sided and two sided latency of Open MPI

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 size. This is also observed in Section 4.5 and we will discuss it further there. For messages smaller than 2KiB, MVAPICH provided one of the few instances of RMA message rate performance exceeds the two sided baseline. For those message sizes, MVAPICH Lock does an average of 21.4% better than the

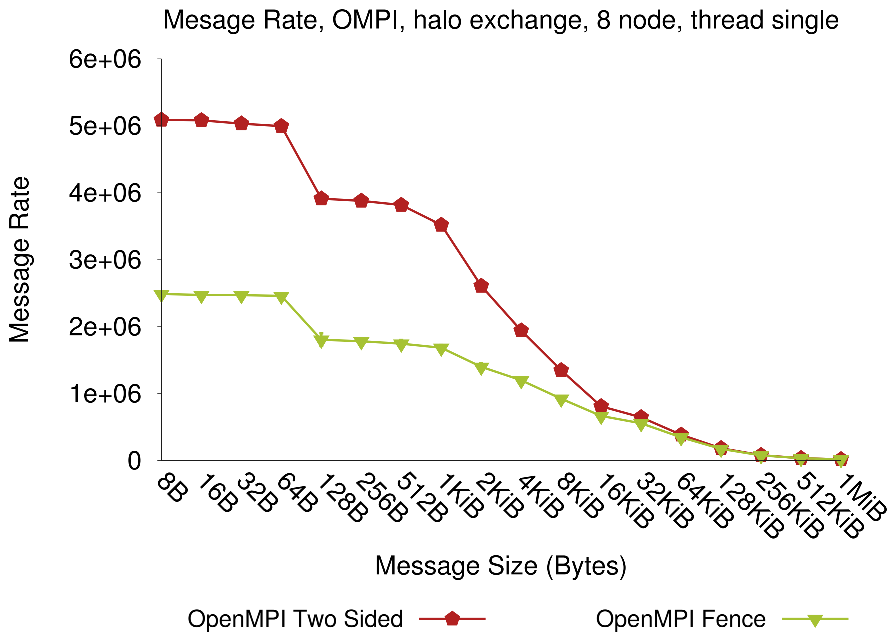

Figure 4.3: Single threaded one-sided and two sided message rate of Open MPI

baseline.

4.5 Multi-Threaded Benchmarks

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 dis-

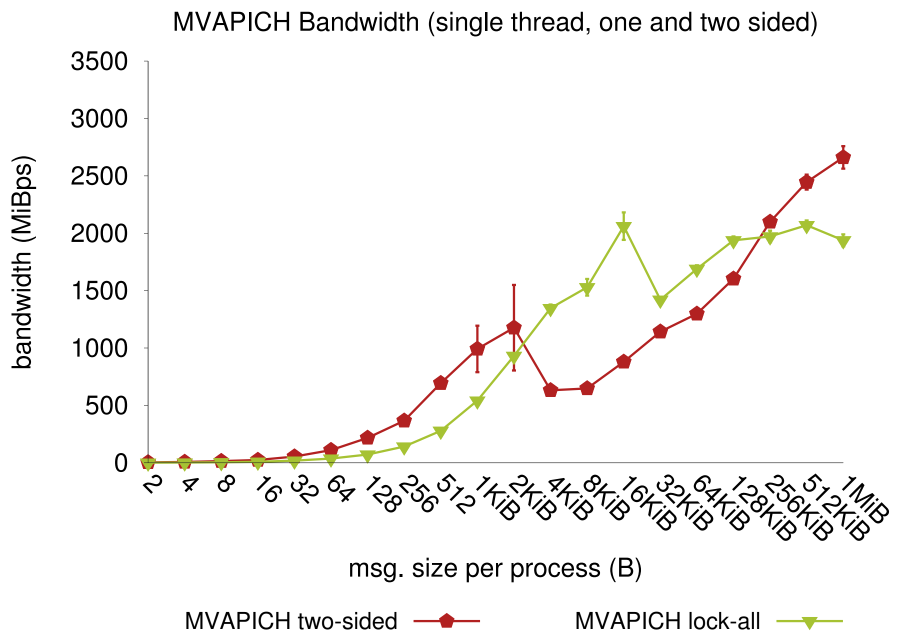

Figure 4.4: Single threaded one-sided and two sided bandwidth of MVAPICH

cussion 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.

4.5.1 MVAPICH Release

Bandwidth Our results from MVAPICH only reported minor differences in throughput when comparing different synchronization methods. We focus the plot for the Lock_All

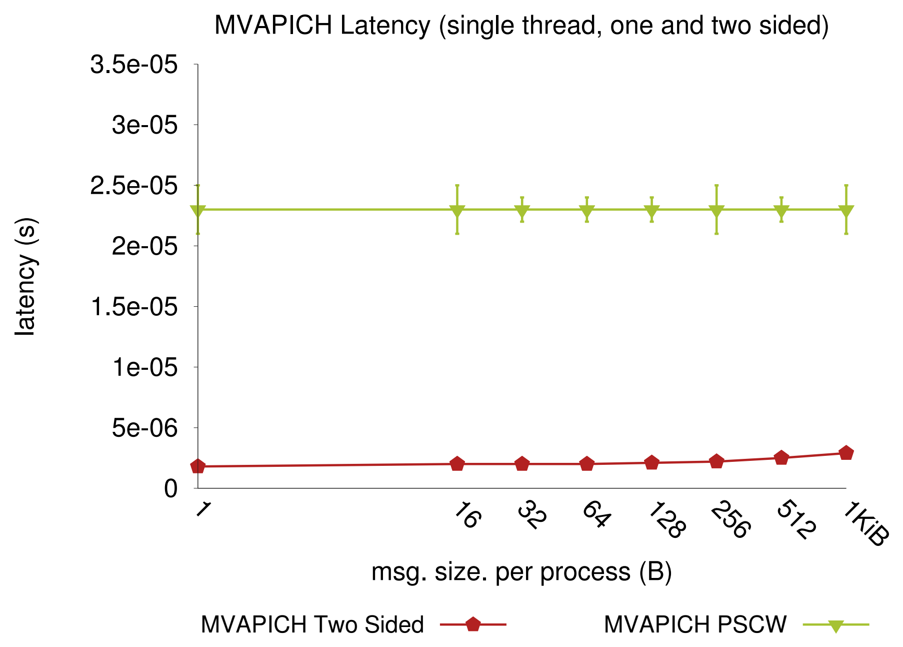

Figure 4.5: Single threaded one-sided and two sided latency of MVAPICH

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 4.7 the results require close examination; 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.

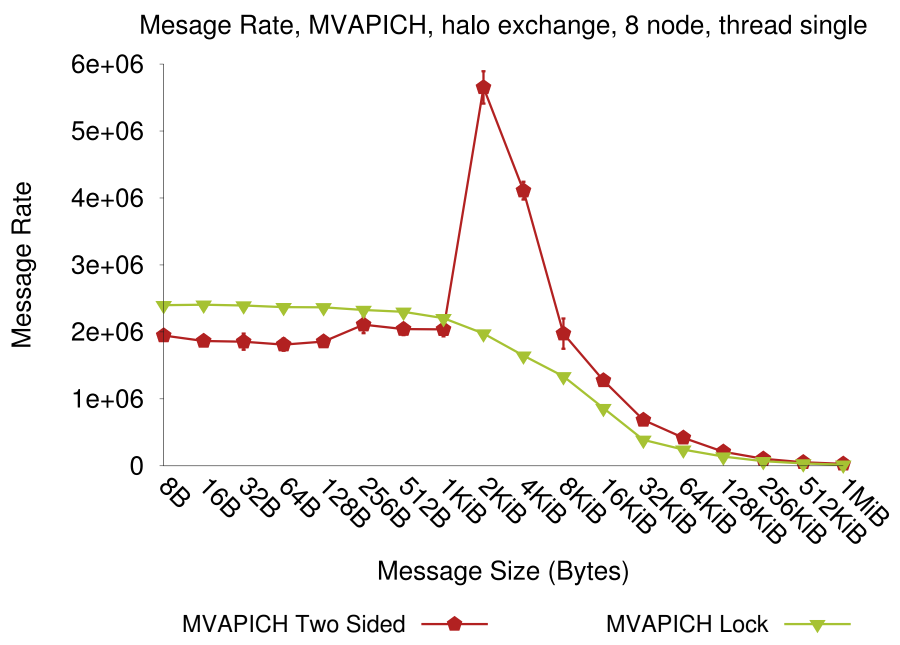

Figure 4.6: Single threaded one-sided and two sided message rate of MVAPICH

Latency

The results of Figures 4.8 and 4.9 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.

Figures 4.10 and 4.11 present the results of the halo exchange message rate benchmark when run under MVAPICH. This figure shows the total message throughput of the benchmark for

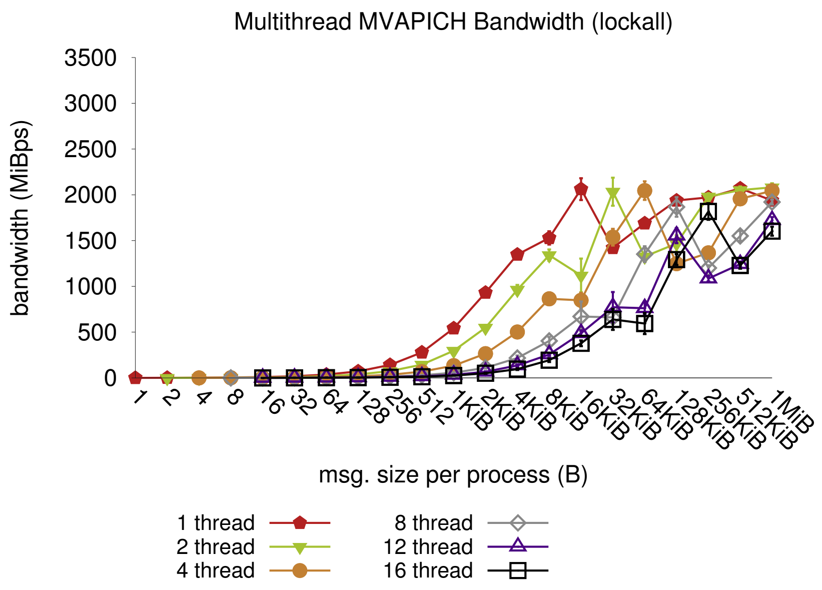

Figure 4.7: MVAPICH bandwidth results for one-sided (lockall) communication

each of the synchronization methods. 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 multithreading has been turned off at compile. This was done to compare RMA-MT to a current day implementation.

In Figure 4.10 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 didn't 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 4.11 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.

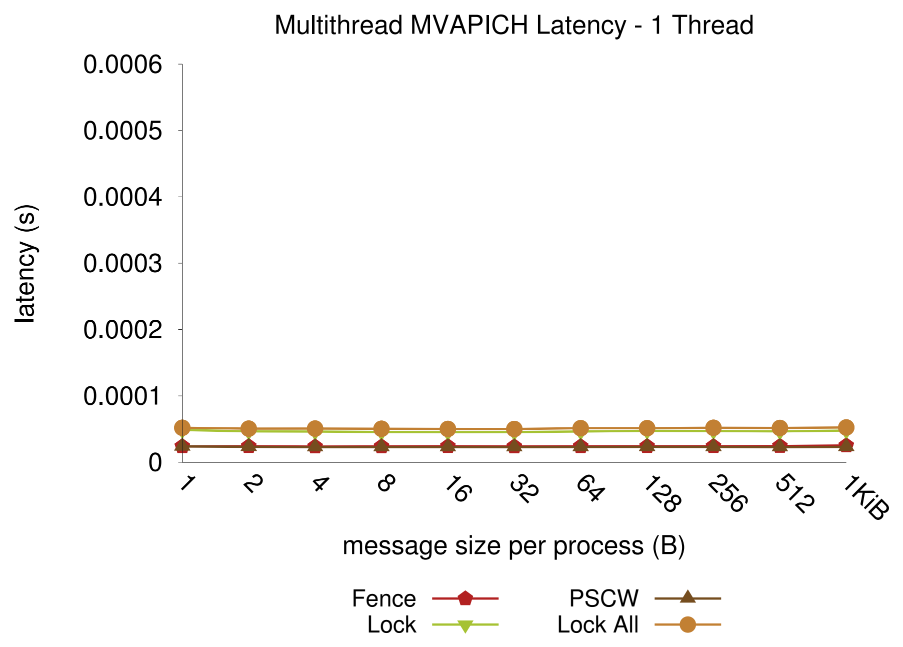

Figure 4.8: MVAPICH latency results for 1 thread

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].

4.5.2 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

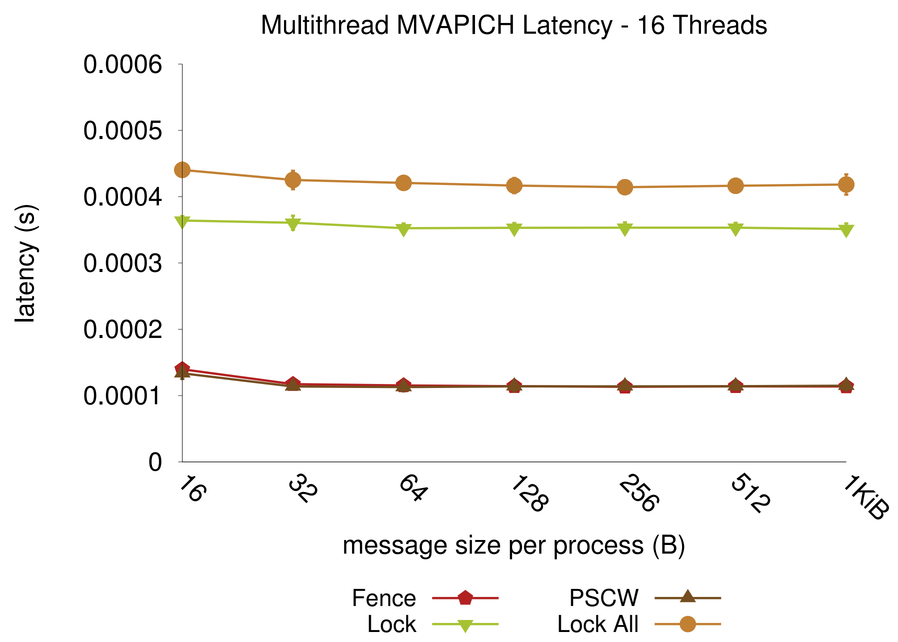

Figure 4.9: MVAPICH latency results for 16 threads

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 4.12-4.13, it is evident that for single threads, the 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%.

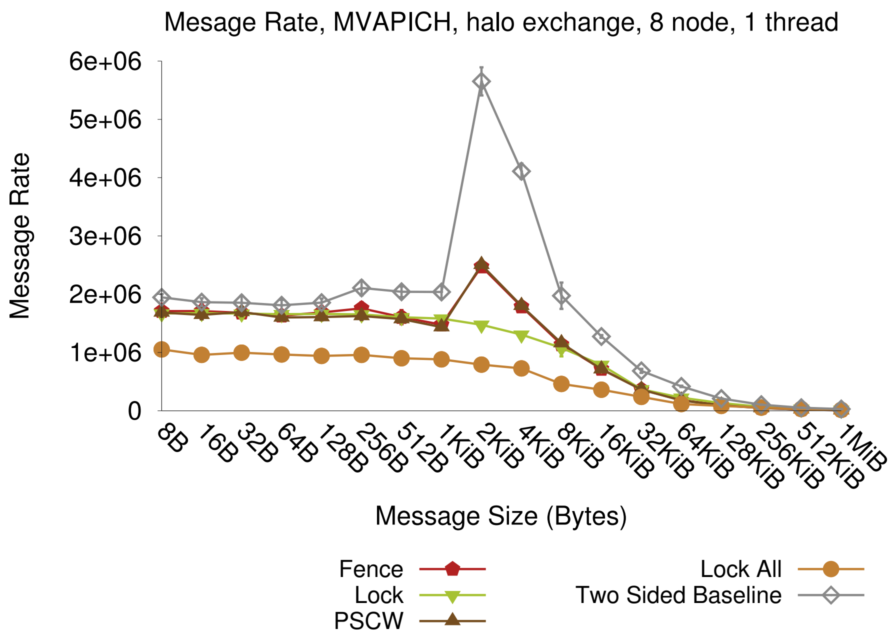

Figure 4.10: MVAPICH message rate results for 1 thread

Latency

The results of Open MPI latency (shown in Figure 4.14-4.15) 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

Figures 4.16-4.17 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

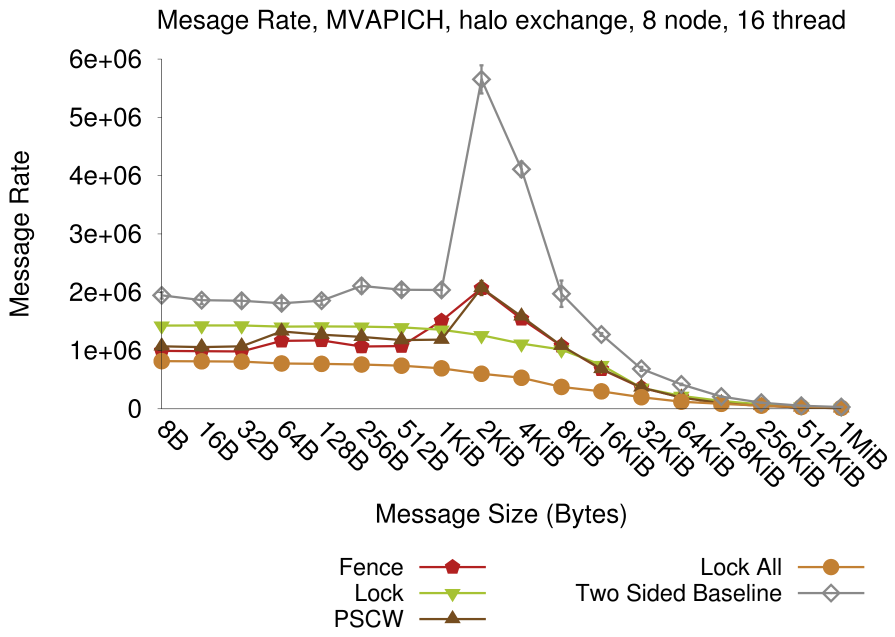

Figure 4.11: MVAPICH message rate results for 16 threads

a special case, as it is run under a single threaded instance of MPI where the multithreading 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 4.17 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.

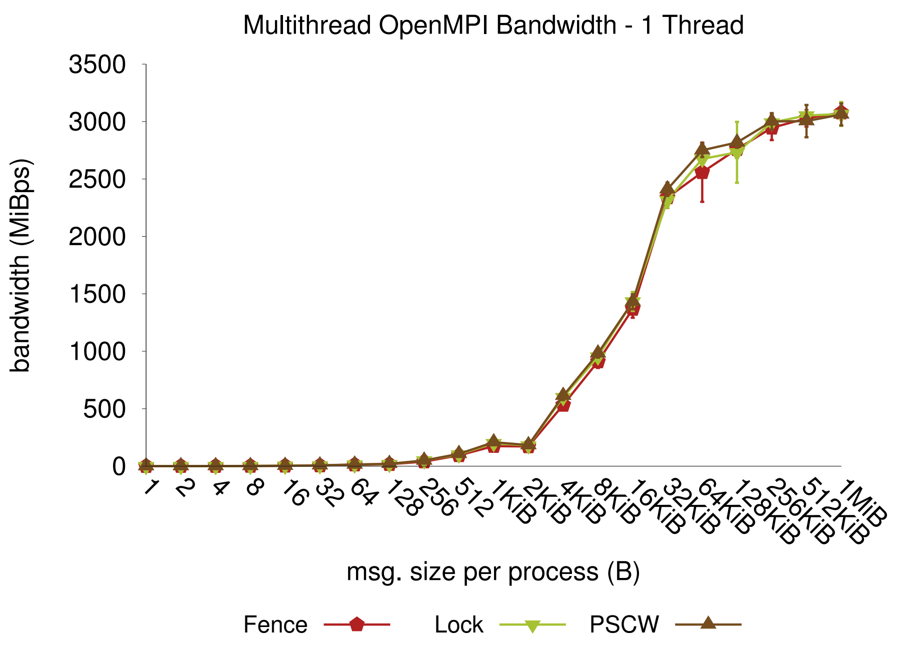

Figure 4.12: Open MPI bandwidth results for 1 thread

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 hundreds of one-sided communications across 20 different message sizes. Because we were using a development branch of Open MPI we had a number of runs where errors were detected. These errors were limited to multithreaded 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,

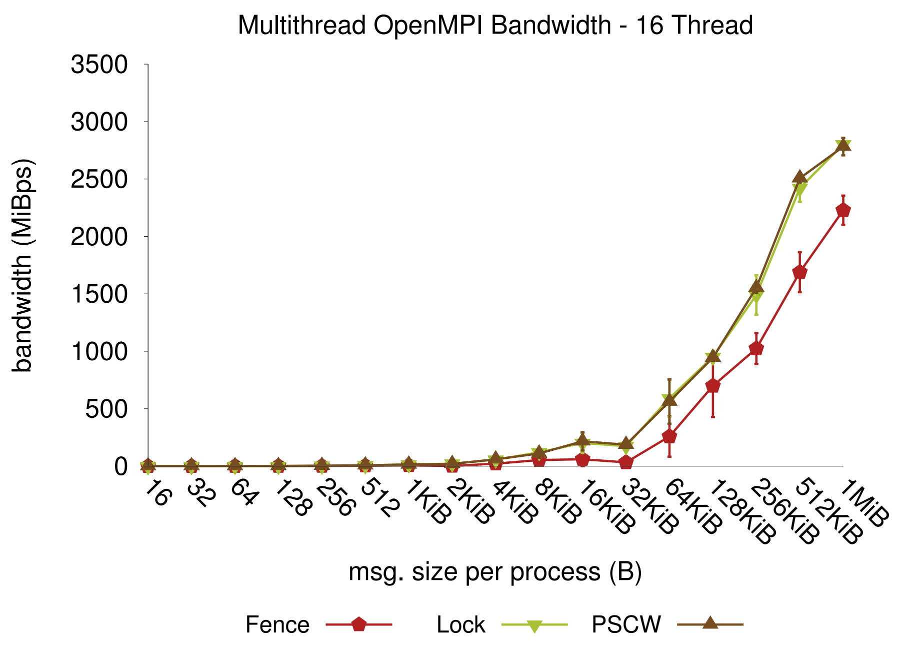

Figure 4.13: Open MPI bandwidth results for 16 threads

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.

4.5.3 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.

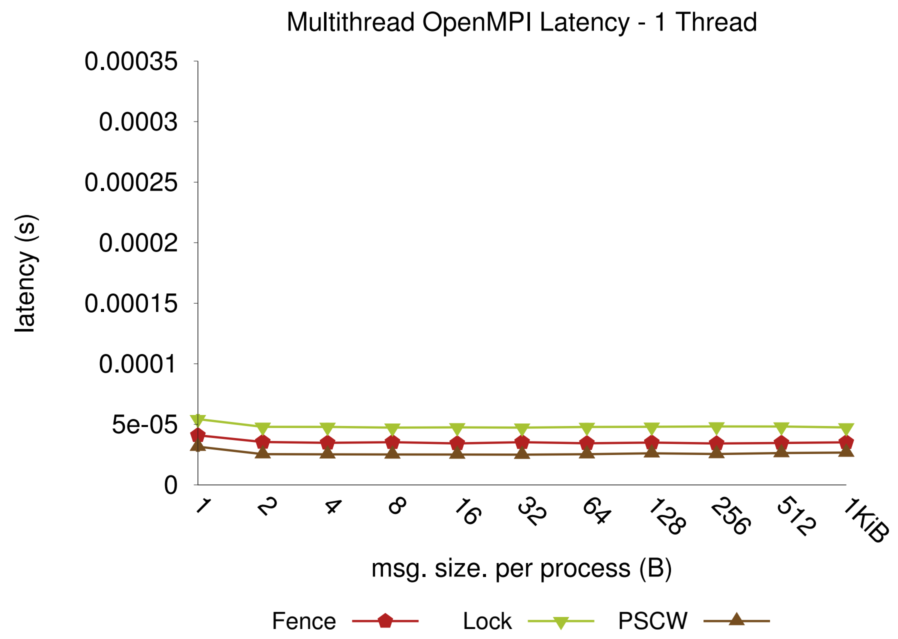

Figure 4.14: Open MPI latency results for 1 thread

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 [49].

4.6 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 4.18 graphs the performance of our tests normalized compared to the performance of the original non-RMA version.

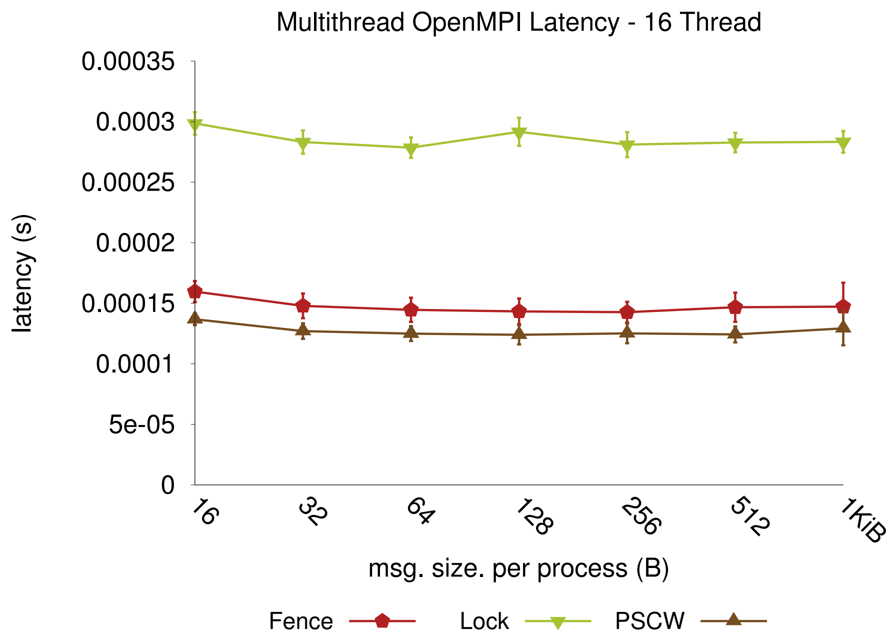

Figure 4.15: Open MPI latency results for 16 threads

4.6.1 HPCCG

For HPCCG, the results in Figure 4.18 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.

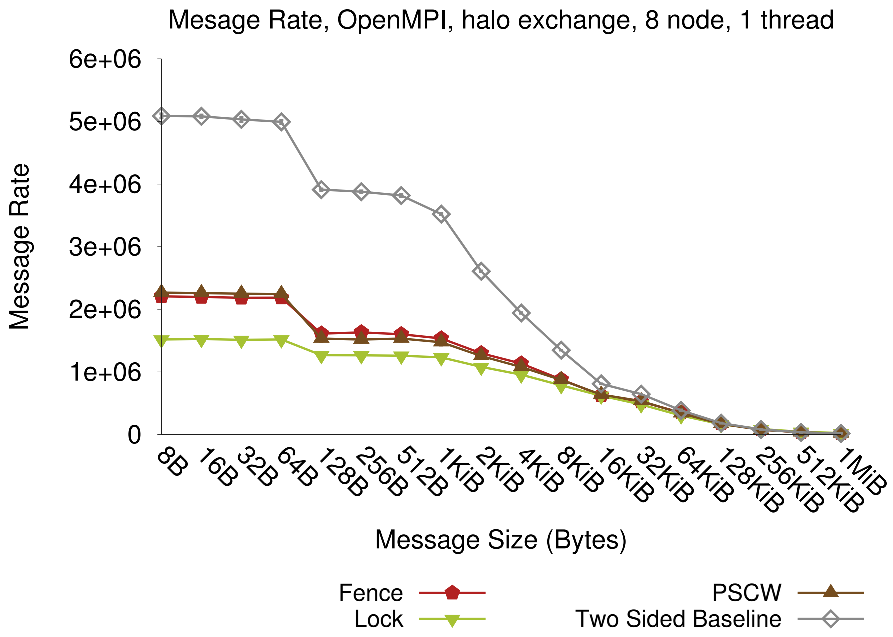

Figure 4.16: Open MPI message rate results for 1 thread

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.

4.6.2 MiniMD

For MiniMD, the results in Figure 4.18 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

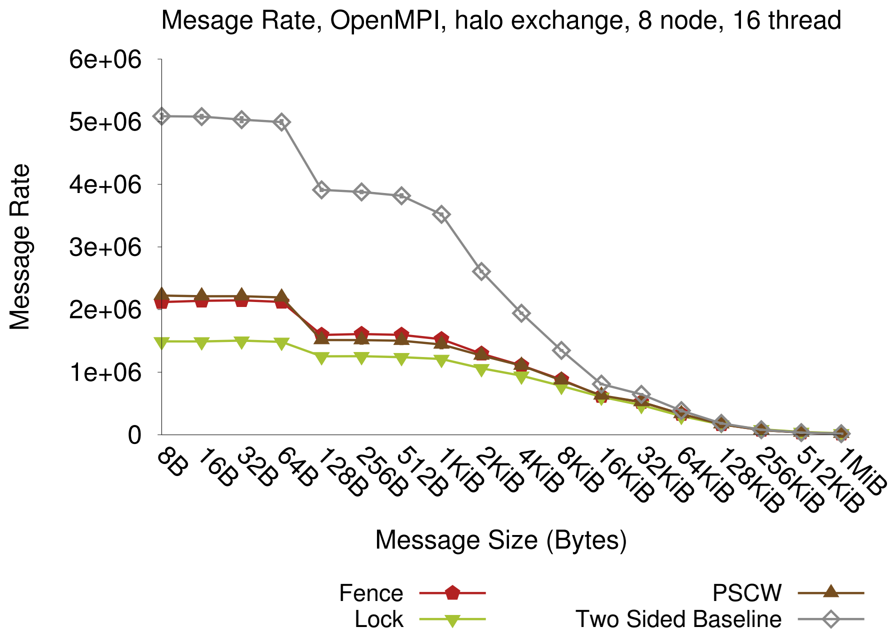

Figure 4.17: Open MPI message rate results for 16 threads

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.

4.6.3 MiniFE

Finally, Figure 4.18 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

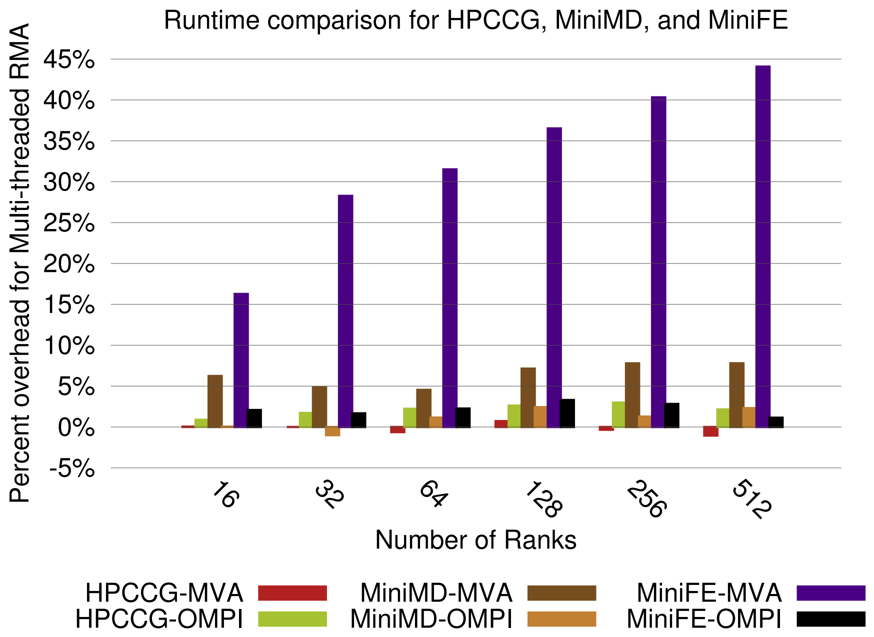

Figure 4.18: RMA-MT mini-app run time overhead compared to the regular version

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.

4.7 SHMEM Results

In this section we present initial results using this benchmark suite on a Cray XE30m cluster. Each node has two Xeon Ivy Bridge 2.4 GHz 12-core processor with hyper-threading enabled, 32 GB of memory per node, and a Cray Aries network interface. SHMEM-MT benchmarks were compiled using the Cray compiler suite and Cray shmem version 7.3.2. Each data point in this section is an average of 10 runs with each run performing 10,000 iterations, in the case

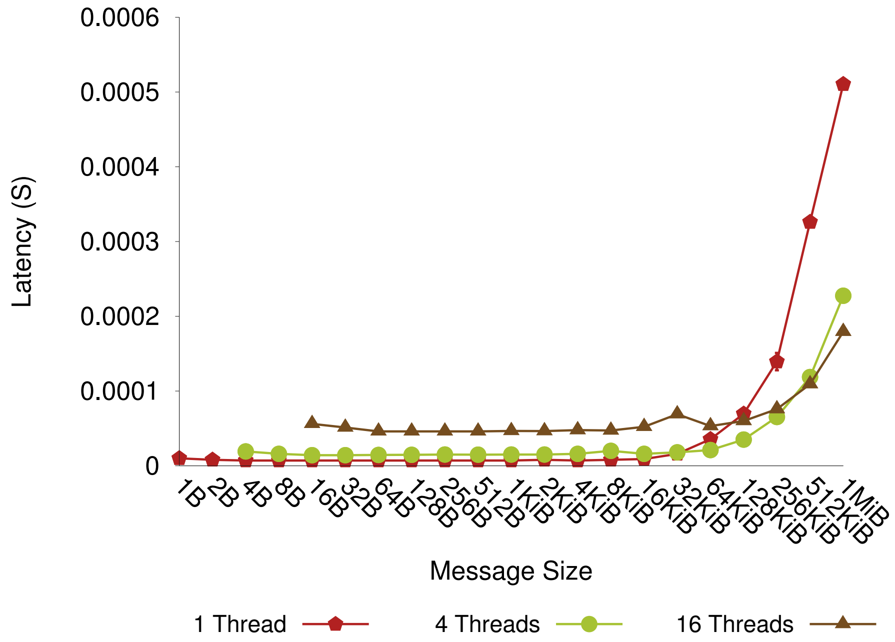

Figure 4.19: SHMEM-MT Latency Performance - Get

of the messaging benchmarks. Each point is plotted with error bars showing the standard deviation of the 10 runs; in a large number of cases, the standard deviation of the ten runs was small enough not to show up on the plots.

4.7.1 Latency

Figures 4.19 and 4.20 show the results from the multi-threaded latency tests using get and put operations. In this graph, the message size is used is the total message size sent every iteration and each thread sends a piece of that message. For instance, a message size of sixteen results in 1 16-byte put or get for the one thread case, 4 4-byte puts or gets for the 4 thread case, and 16 1-byte puts or gets for the 16 thread case; an iteration completes only when operations by all threads are completed after joining the threads that issue the operation and finishing a call to SHMEM Quiet.

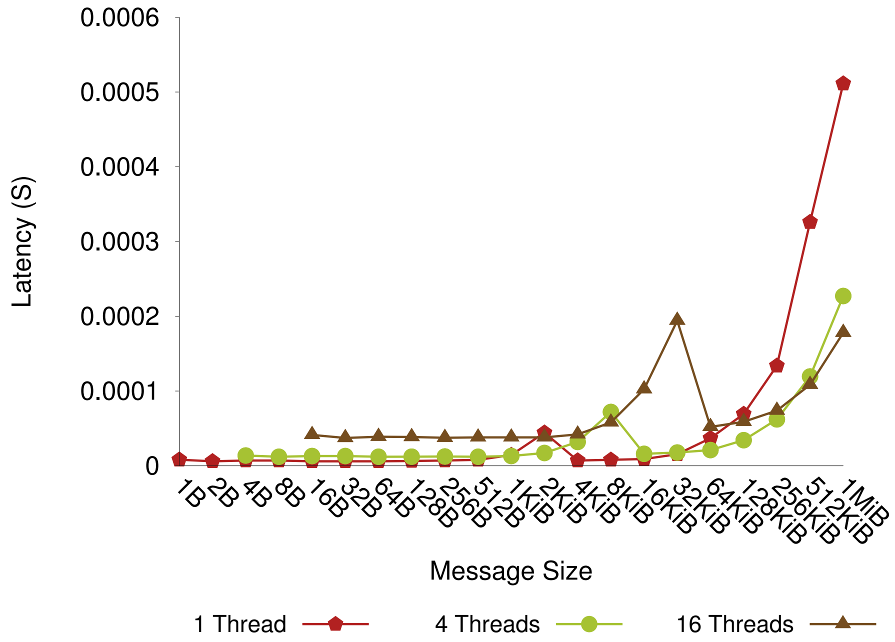

Figure 4.20: SHMEM-MT Latency Performance - Put

As expected, for small messages, splitting a single transfer into smaller transfers increases latency, even accounting for the protocol switch when individual messages reach 2KiB in size. This holds true until about 32 KiB for four threads and 128 KiB for sixteen threads. For larger messages, however, it becomes advantageous to split the data and use multiple threads. With a 1 MiB total transfer size, for example, a threaded transfer of 16 64-KiB messages has a latency 35% of that of a single 1-MiB message. We believe that this is due to multiple threads being able to exploit hardware-level concurrency in the NIC and fabric; the need to use multiple threads or ranks to maximize messaging performance on the Aries network is consistent with previous results on this system [53].

4.7.2 Bandwidth

Figures 4.21 and 4.22 show the results from the bandwidth tests using get and put respectively. This test is setup similarly to latency; as the number of threads increases each

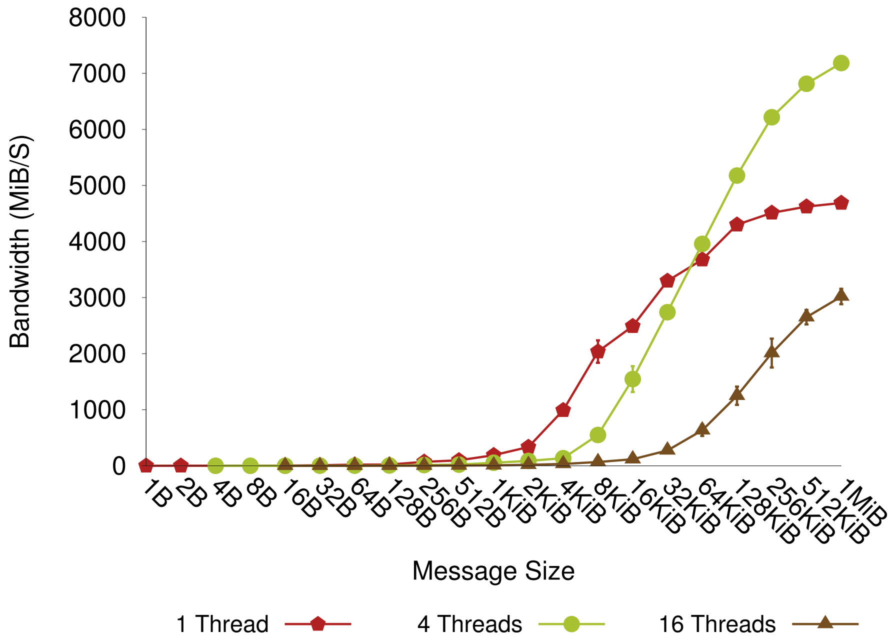

Figure 4.21: SHMEM-MT Bandwidth Performance - Get

message is split into smaller equal pieces for each thread to send. We can see that for small messages, specifically less than 32KiB, Cray SHMEM achieves best bandwidth performance when using a single thread. After 32 KiB, 4 threads sending portions of the message appear to outperform the 1 thread case. For the message sizes used in this test, 16 threads always performs worse than the other cases, most likely because there is not sufficient hardware-level concurrency to amortize the increased synchronization overheads.

4.7.3 Message Rate

Figures 4.23 and 4.24 show shows the results of the message rate halo exchange benchmark, using either get or put operations to exchange messages. This benchmark splits the work among threads differently than latency and bandwidth, keeping the same number of similarly sized transfers per iteration and simply changing the number of messages sent by each thread in each per iteration. As a result, the number and size of messages sent per iteration is

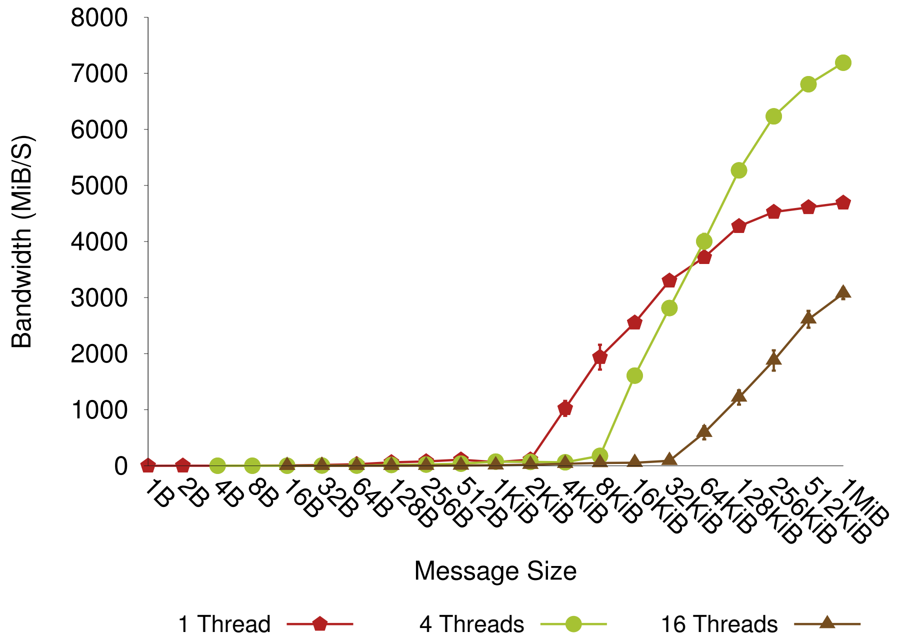

Figure 4.22: SHMEM-MT Bandwidth Performance - Put

constant at a given message size. As a result, these tests evaluate the potential benefits of threading and overheads of contention when accessing the communication library and underlying hardware.

As we can see from the data, the message rates are largely identical with changing numbers of threads, little benefit or overhead for the majority of the graph. The primary exception is for messages of 128 KiB and larger. In these cases, using four threads results in increased messaging performance. For example, four threads with 1 MiB messages results in 47% higher rate than with 1 thread; again, this is likely due to multiple threads being better able to utilize concurrency in underlying network resources.

While our results do not show as large of a performance difference based on the numbers of threads used, they do show a large difference between using Put and Get operations. For small messages, the results show a significantly higher message when using put rather than get for small messages. We believe that this is due to the synchronous nature of get

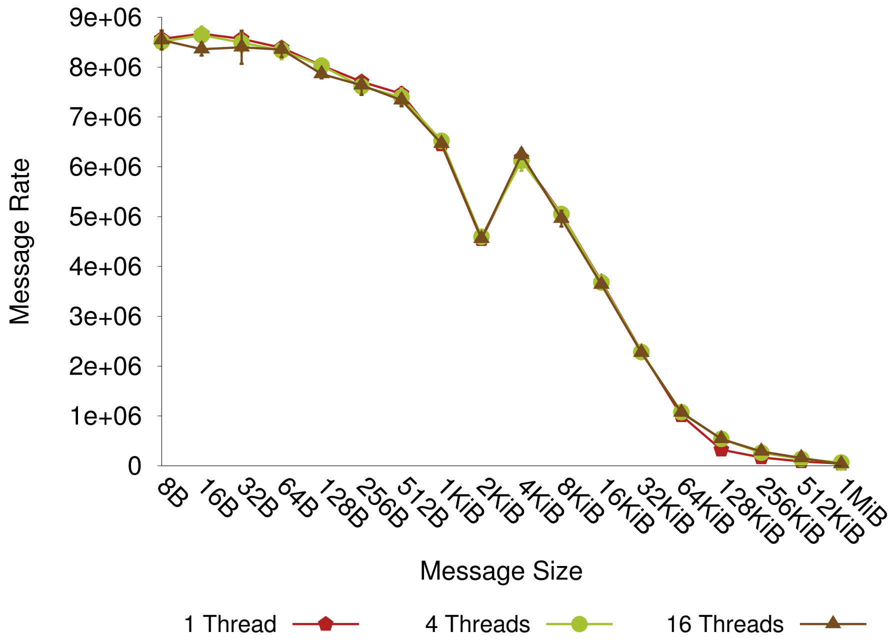

Figure 4.23: SHMEM-MT Message Rate for Halo Exchange - Get

operations in OpenSHMEM as opposed to put operations. We also notice that after the protocol switch at 2KiB, the performance difference between Put and Get operations largely disappears.

4.7.4 MiniApp Results

Figure 4.18 shows the runtime of the HPCCG and MiniFE mini-applications when run with 24 ranks per node on up to 32 nodes in a weak scaling configuration. In particular, the problem size for HPCCG was set to 100^3 rows per PE, using 24 processes per node or one for each core while the MiniFE problem size was set at $(330 * nodes^{1/3})^3$. Note that, as described earlier, these mini-applications do not yet include full threading support. In this case, messaging concurrency results from running multiple OpenSHMEM PEs per node, in this case one PE per core.

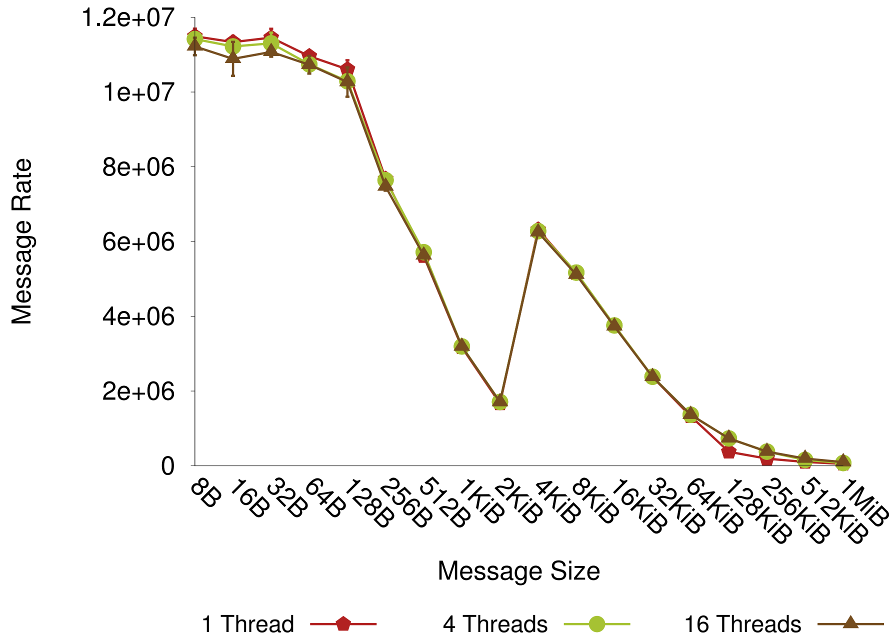

Figure 4.24: SHMEM-MT Message Rate for Halo Exchange - Put

In both cases, performance is largely constant as expected, particularly for MiniFE. HPCCG runtimes begin to increase slightly with increased scale, but to a level that is generally expected for this mini-application. Note that both mini-applications also include solution verifications provided from the original Mantevo versions that complete successfully.

4.8 Analysis

This chapter shows that one-sided communication is a viable, performant option for new exascale codes. A simple, vendor optimized, dedicated one-sided communication library in the form of Cray's SHMEM implementation is able to both fully utilize the network and run hybrid codes with minimal synchronization overhead. This contrasts with the baseline of two-sided MPI implementations where data structures and shared state like the matching engine show large overheads when run under $THREAD\_MULTIPLE$. One-sided communication

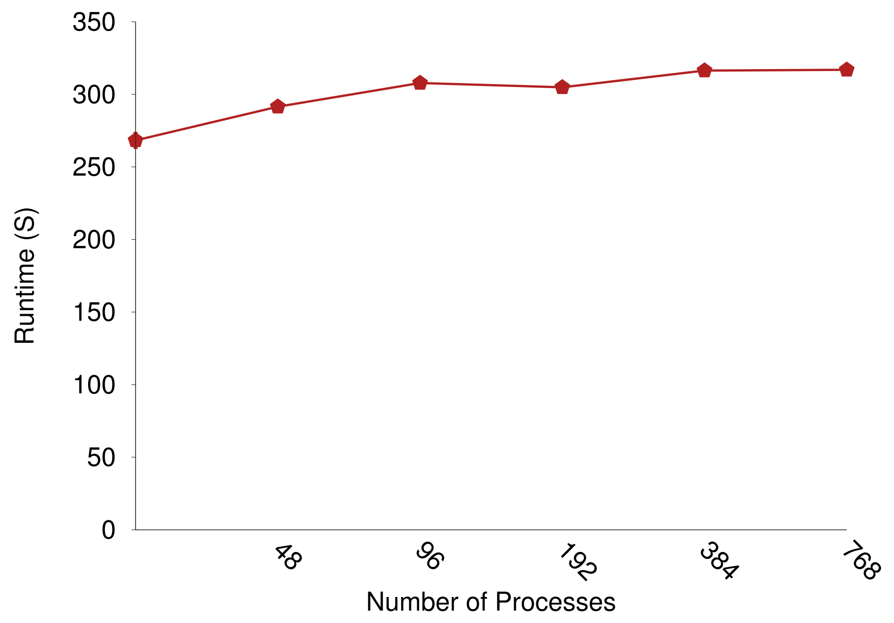

Figure 4.25: SHMEM-MT Mini-application Weak-scaling Runtime - HPCCG

can avoid most of these data structures and is therefore better suited for extreme scale hybrid applications. One-sided communication is also able to better support fine grained communication strategies due to the aggregation of communication synchronization. At the time of these experiments, MPI was only starting to support these features. This is likely why the results in Section 4.5 were underwhelming compared to Cray's SHMEM implementation.

The are a few notable limitations to one-sided communication as defined in these libraries. First, there is a large overhead to converting legacy application over to this model of parallelism. Changing one time use buffers into persistent memory windows, managing memory epochs, and managing data placement are some of the challenges when dealing with this model. It will likely be challenging to architect an application around and difficult to adapt a legacy application to adapt for these changes. Secondly, performance is not as portable as traditional two-sided MPI; application developers often use the default installed MPI implementations which can be very outdated. For example, an application using RMA with

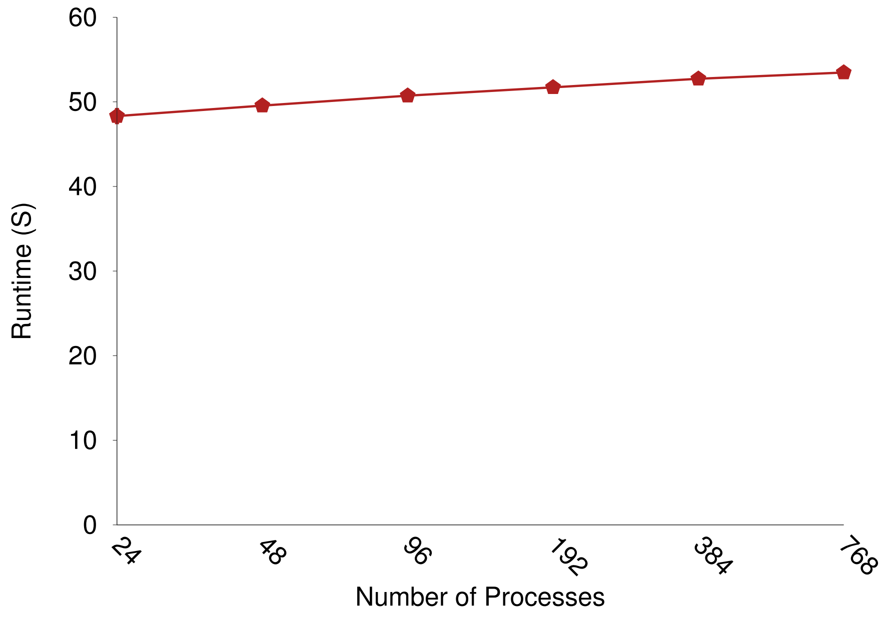

Figure 4.26: SHMEM-MT Mini-application Weak-scaling Runtime - MiniFE

Open MPI 1.10, the default for some systems, will result in a decrease in performance, even if the implementation improves with later versions. Finally, there are some communication patterns that are not easily handled by one-sided communication. For example, AMG 2013 has occasional long-range communication that is not expected by the receiver. In two sided MPI, this application uses a probe operation to check for unexpected messages and then dynamically allocates a buffer to receive that data. This is difficult in one-sided communication as there are challenges with placing the data into a pre-allocated window, avoiding memory that has been written to by another communication, and notifying the receiver to processes the message.

4.9 Conclusions

In this chapter, I described two benchmark suites that explore new parallel programing paradigms, both using one-sided communication and multithreading. RMA-MT show that while MPI RMA in combination with multithreading shows promise, it is still very preliminary. Recent work has made great strides in improving these code paths. Both the Open MPI and MPICH teams have utilized the RMA-MT benchmark suite to implement, test, and optimize these code paths. SHMEM-MT explored the possibilities of one-sided communication using an alternate communication library. As a newer, light-weight, one-sided centric communication library, it is unsurprising that these tests showed better performance and functionality with SHMEM than with MPI RMA. In certain circumstances, such as the bandwidth test, multithreaded SHMEM was able to leverage more of the network than the single threaded variant.

This work contributed both a suite of benchmarks, RMA-MT, to explore the performance of next generation communication systems and studies using said benchmarks of the performance of both MPI RMA and shmem in hybrid programing contexts. This answers the research question of 'How do we evaluate new parallel programing techniques on many-core architectures?' by not only providing an evaluation of these two programing models, but also a framework for throughly testing new communications models, especially for hybrid contexts.

Chapter 5

Matching Engine Architectures For Modern Processors

5.1 Introduction

Chapter 1 described a number of different parallel programing paradigms for exascale computing. One of the largest challenges is to adapt legacy codes to modern systems. Often these codes will be run as-is as there is not a drive to update these large applications. Thus, to make these codes performant on modern architectures we must adapt the underlying system software to their behaviors. The research question this chapter aims to answer is 'How do we optimize performance of current and future communication libraries for many core architectures?'.

In this chapter, I present three new matching architectures that target many core systems such as the Xeon Phi. This work was the result of a collaboration between S. Mahdieh Ghazimirsaeed, Whit Schonbein, Micheal J. Levenhagen, and myself. Mahdieh contributed an initial implementation of the linked list of arrays architecture, a Knight's Corner prototype of the vector approach. Whit contributed match engine profiling data. Micheal contributed the data for the communication pattern match list depths. Finally, my contribution was the hot caching architecture and implementation, the final design and optimizations for the linked list of arrays architecture and implementation, a Knight's Landing vector architecture, implementations that combine the approaches in both MVAPICH and Open MPI, and a performance study of the architectures.

The contributions of this chapter are the following:

- Three matching engine architectures for modern architectures
- Implementations in both MVAPICH and Open MPI.
- A performance study study of the architectures and combinations of the different approaches.

This chapter is structured as follows: Section 5.2 presents an exploration of list length of common MPI communication patterns. Section 5.3 presents the different architectures and details of the implementations. Section 5.4 presents the experiments including the exploration of match list length and a performance study of the different architectures.

5.2 Background and Motivation

An important metric to study prior to examining any message matching queue optimizations is the typical match queue length of common communication patterns, particularly at scale. In order to study general patterns and enable larger scale evaluations, we used a modified version of the SST simulation system [90]. We used message queue data from SST to create histograms of match list behavior for these communication patterns. SST contains motifs that describe common communication patterns for a number of categories of applications. We examined three patterns at large scale: an adaptive mesh refinement pattern, AMR, at 64K processes, a 3D sweep pattern at 128K processes, and a Halo-exchange pattern at 256K processes.

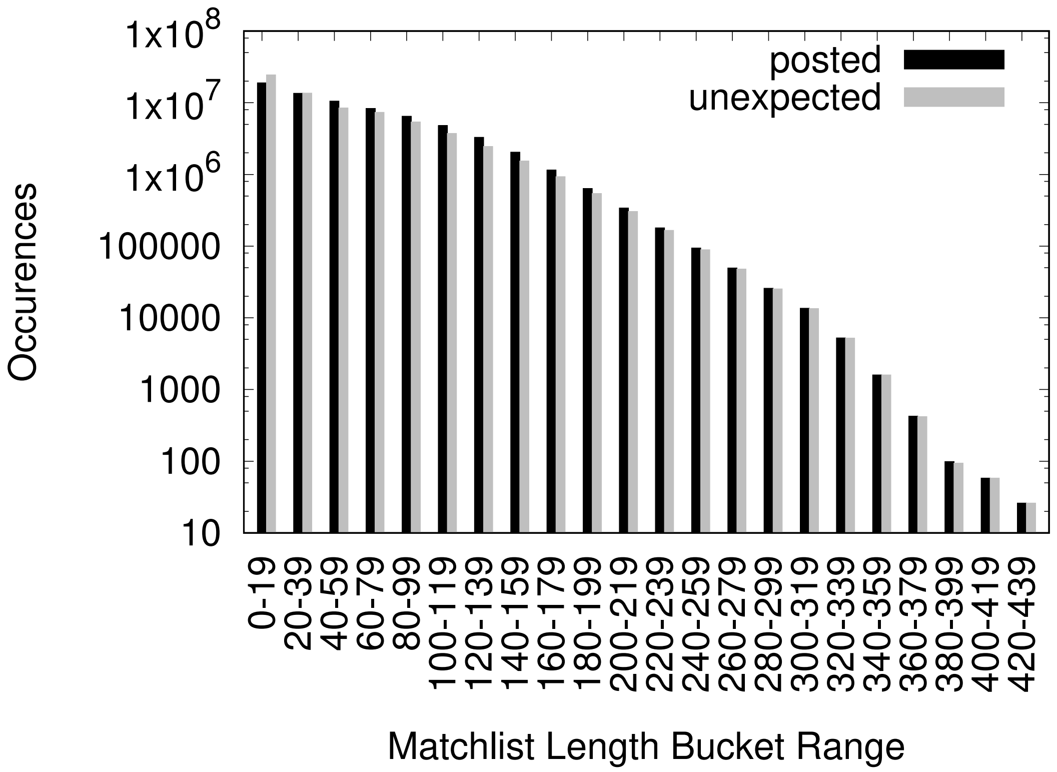

Figure 5.1: AMR match list sizes - 64K

Figure 5.1 shows the number of items in a given process' match list on the x-axis and the number of times that such a sample occurred throughout the simulation on the y-axis. Samples are taken during each communication phase in simulation, such that all list additions and deletions are captured. AMR shows that most list lengths maintain zero to mid-hundreds of elements for the majority of the application run; however, extremes do occur out to the mid 400s. Therefore we can conclude from these results that the most important queue lengths occur in the mid 100s as they are abundant and intensive to search. It is also important to consider the performance ramifications of several hundreds of list elements at a time to capture performance drop off with regards to longer lists.

Sweep3D patterns in Figure 5.2 show similar results to AMR, with the exception of the length of exceptionally long queues. Sweep3D needs short list length performance with good

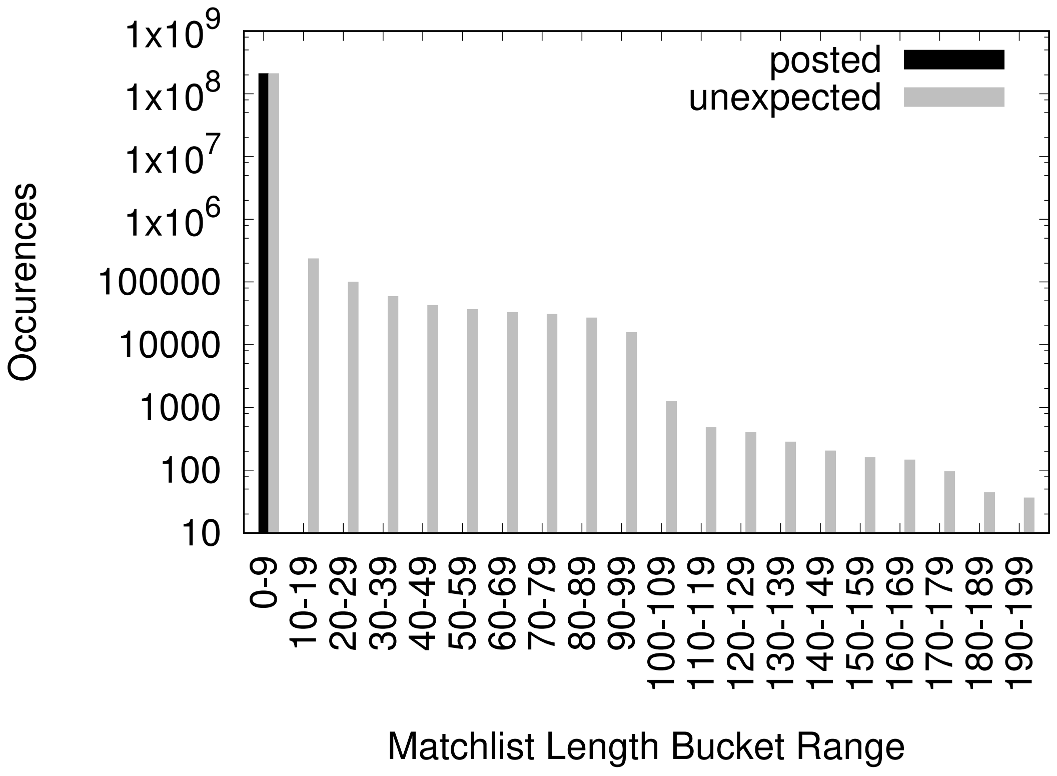

Figure 5.2: Sweep3D match list sizes - 128K

performance for queue lengths into the low hundreds of elements.

Halo3D shows the expected queue lengths for a neighbor halo exchange, few elements in the queue and many very small queue length operations. As expected, queue lengths do not grow to beyond 27 elements, as this is the maximum number of send/recv operations that can be processed in between communication barriers.

5.3 MPI Matching Improvements

In this section we present three novel techniques for improving message matching. Section 5.3.1 presents a linked list of arrays matching architecture that attempts to better

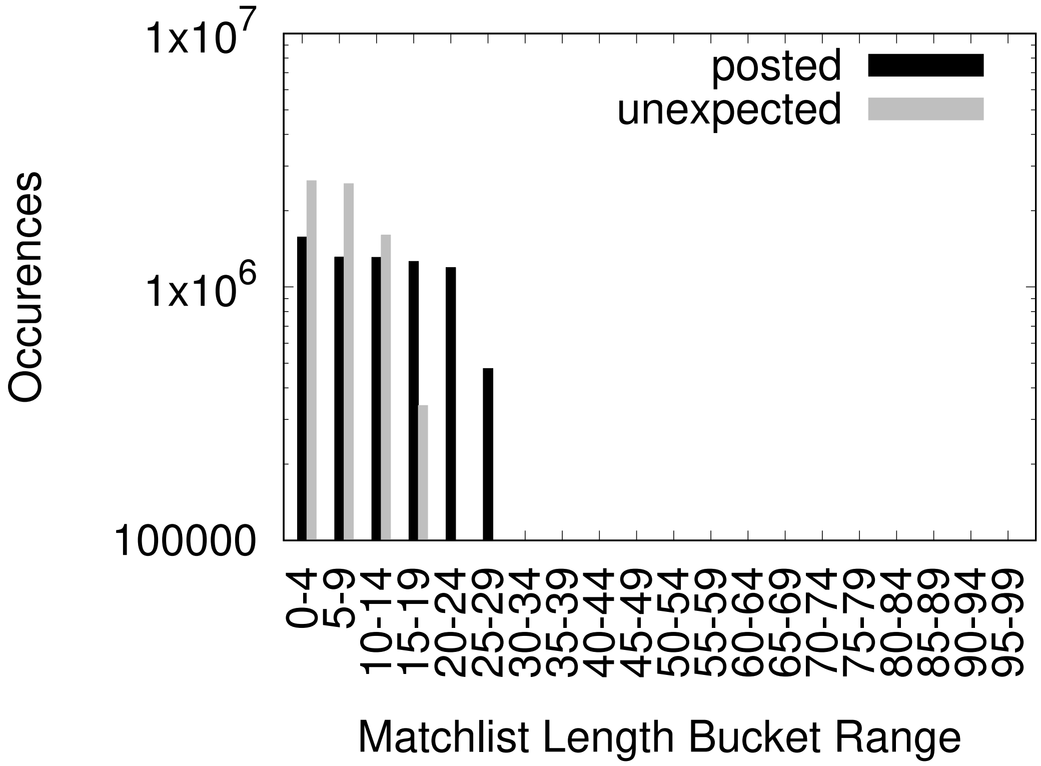

Figure 5.3: Halo3D match list sizes - 256K

interact with the memory subsystem by combining multiple matching elements into a single linked list element. Section 5.3.2 expands on this idea by rearranging data into AVX vectors. Finally, Section 5.3.3 presents *hot caching*, a technique to ensure data remains in cache.

These techniques utilize auxiliary data structures, and can be used while maintaining the underlying match lists. This makes it feasible to utilize these techniques when it improves performance and, if they are no longer the performant path, revert to using the original linked list structure. They are also complementary, with the vector approach extending the structure of the linked list of arrays and both data structures benefiting from cache miss reductions from hot caching.

5.3.1 Linked List of Arrays

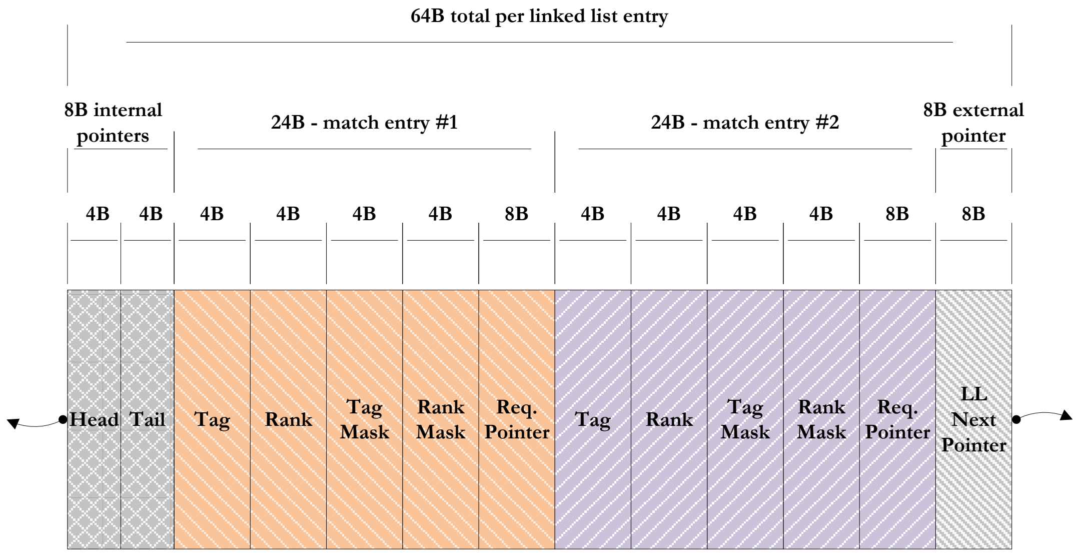

Figure 5.4: Packing data structures into 64 byte cache lines

Our linked list of arrays structure utilizes contiguous memory regions to better interact with the caching behavior and pre-fetching. Each queue element for the posted receive queue contains 24 bytes of information, 4 bytes each for the tag and rank, eight bytes of bit masks for matching, and an 8 byte pointer to the request. The unexpected message queue does not require masks, so it only requires 16 bytes per entry. There are also 3 per array items that are stored: a pointer to the next array and indexes to the array indicating the start and end of the used section. We manage holes in the array (from deletions in the middle of the list) by ensuring tags and sources are invalid and all bitmask fields are set.

This data structure allows a variety of size options that generally fit into two approaches. The first approach provides a long array that interacts well with the prefetcher. Iterating over contiguous memory is a predictable pattern, and modern architectures are designed to take advantage of this pattern. There is a potential caveat to this approach, as elements that have been matched in the middle of an array must still be checked during subsequent searches. This approach is ideal for applications that frequently empty the match list.

The second approach, illustrated in Figure 5.4, optimizes the efficiency of each memory look-up. For the posted receive queue, we fit two entries and the linked list overhead into a single cache-line. This effectively doubles the memory efficiency of the matching operations as it only requires $N/2$ memory operations to fully traverse a list of size N, whereas the traditional method requires N operations.

5.3.2 Intrinsics for AVX Vectors

We then extended the linked list structure described above to take advantage of vector instructions (e.g. Intel's AVX intrinsics) by reorganizing the data to be grouped per field into a vector rather than being grouped by a per element structure. For the posted receive queue we need to store 4 vectors of 4 byte integers for tags, tag masks, ranks, and rank masks. This requires that each linked list element contains 16 entries to fill a 512 bit vector. We also need to separately store pointers to the request structures, head and tail indices, and a pointer to the next linked list element. Our current implementation also stores intermediary arrays for ease in updating the vectors; however, these are not essential to the technique.

When searching the posted receive queue, we use the `set1` function to immediately initialize two target vectors for tag and rank. These can be used for the entire search. For each linked list element, we apply the vector bitmask using a vector bitwise `and` operation and then use a vector equality operation to test for a match. This gives us a 16 field bitmask of match information. After we use this technique for both tag and rank, we check the context_id of any potential matches. The overhead in modifying the vectors is limited to inserting or removing an element.

There are a couple of caveats to this approach. First, similar to the linked list of arrays approach, it preforms best on a dense match-list with very few holes. Secondly, its availability

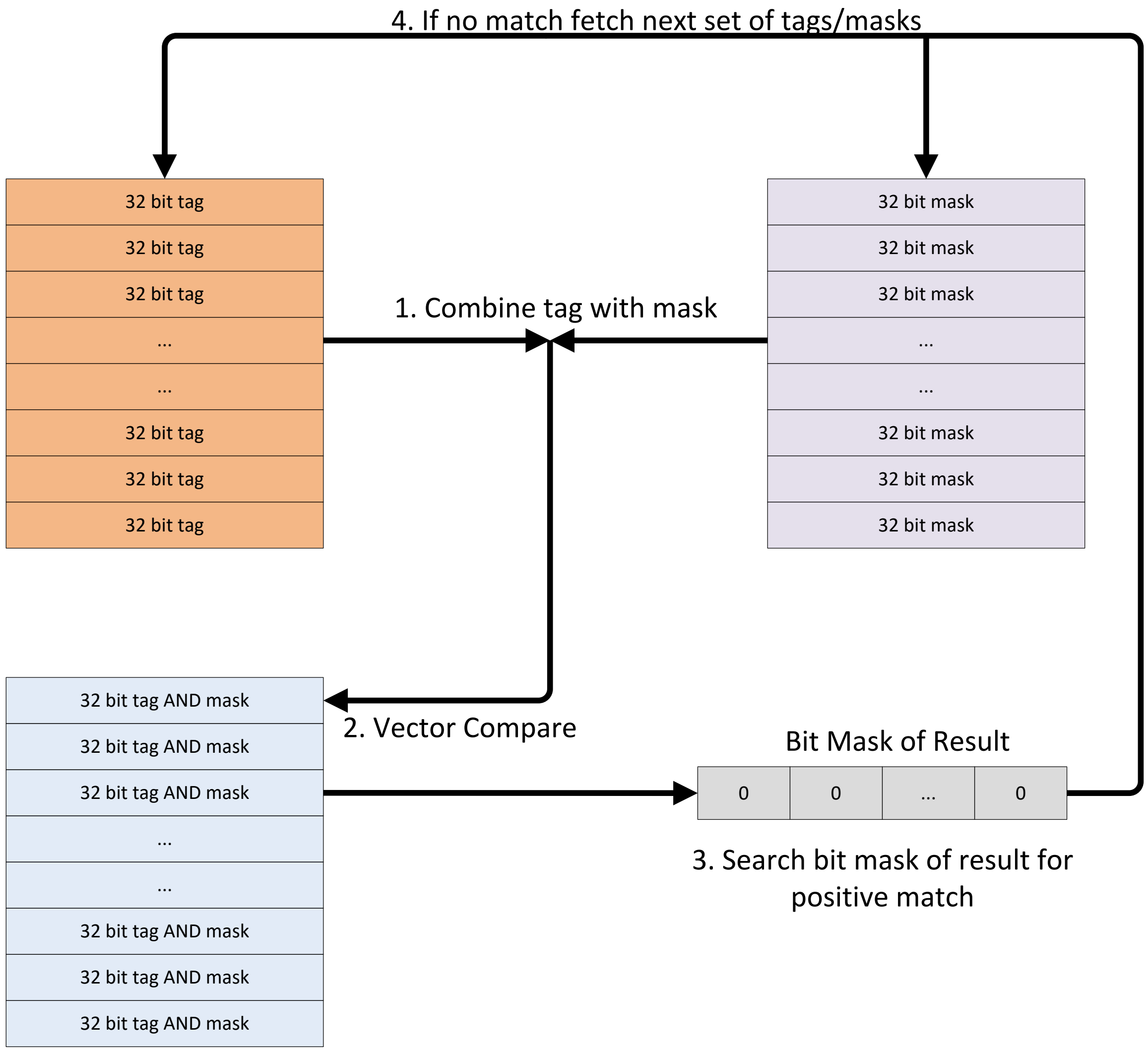

Figure 5.5: Vector matching logic

is architecture-dependent. In particular, matching requires an integer equality check operation, which was introduced in AVX2. We implemented this in AVX-512 for the Knight's Landing architecture, and AVX2 for experiments on a Broadwell system.

This technique can use existing CPU vector units. However, it motivates us to demonstrate that a very simple wide vector unit capable of integer equality checks and bit masks could be designed to accelerate network message matching at relatively low cost in terms of die area and increased complexity.

5.3.3 Hot Caching

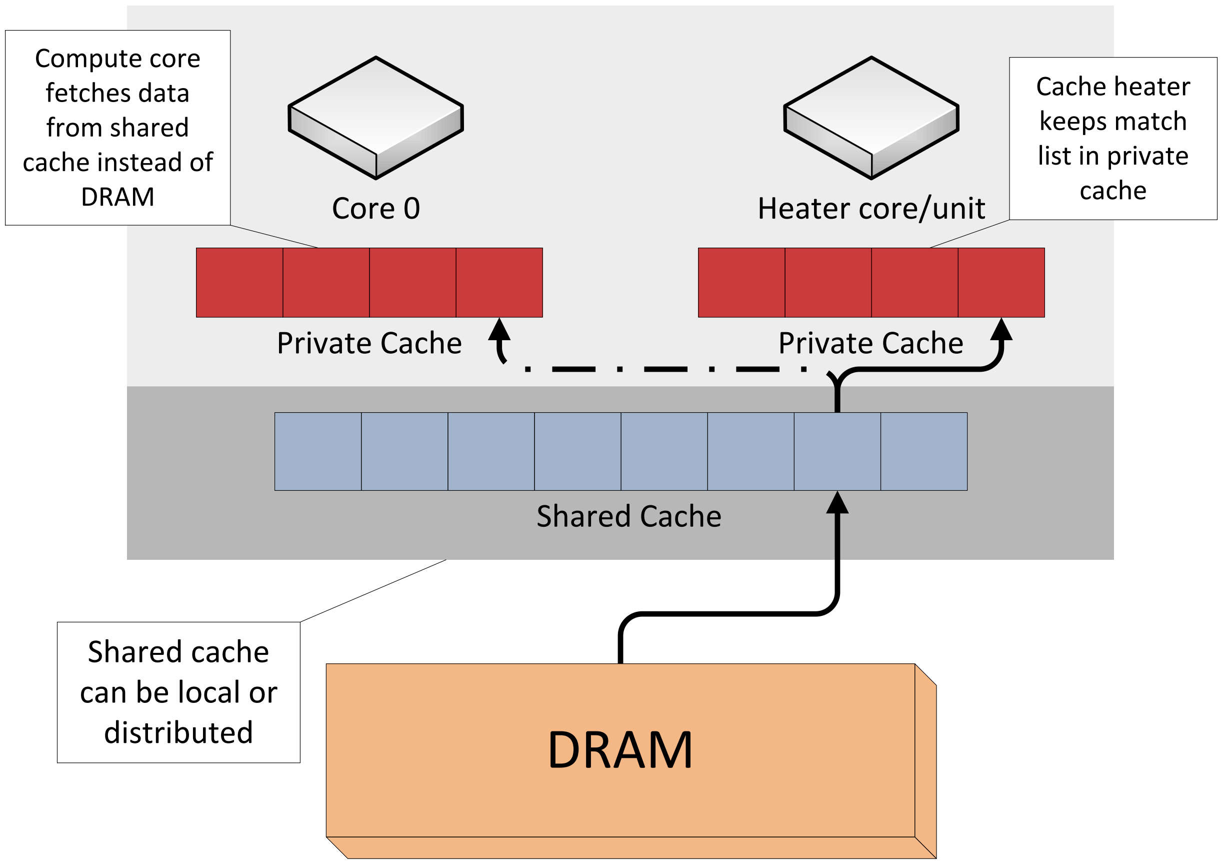
Compute core fetches data from shared cache instead of DRAM
Core 0
Heater core/unit
Cache heater keeps match list in private cache
Private Cache
Private Cache
Shared Cache
Shared cache can be local or distributed
DRAM

Figure 5.6: Hot Caching

We developed an additional technique, termed *hot caching*, to explicitly keep MPI match list entries loaded in cache. MPI typically uses all of the cores on a system, but it is not uncommon to have idle hyperthread contexts available for use. The technique of hot caching ensures a specified region of memory stays in cache by running a thread that periodically accesses said region. Our basic implementation is a thread that iterates through a list of regions, specified as a virtual memory address and a size, and adds the first four bytes of each cache line to a throwaway summation variable. It then sleeps for an arbitrary number of nanoseconds and repeats the process.

There are a number of challenges in implementing this technique. First, when running alongside an MPI implementation it is necessary to pin the extra thread to a core that shares a level of cache with the communication process. Sandy Bridge and Broadwell processors have a shared L3 cache. This allows us to pin the thread to any other core on the socket. The Knight's Landing architecture only has latency efficient cache sharing on a per tile basis. Since each tile only has two cores, it is important to identify and pin the heater process to the other core on the tile.

The second challenge with cache heating is lock contention. When first using this technique with MVAPICH, we realized that the hot caching region list created a large critical section. Removals, particularly with the intent to deallocate the memory region, could cause a segmentation fault if the heater is using the list at the same time. Enforced mutual exclusion through a spin lock is problematic for performance when the region queue is long and insertions and removals are frequent. Instead, we use utilize auxiliary data structures for the list search that re-uses list elements and does not remove entries from the heater.

The third challenge is reducing application interference. The hot caching thread utilizes processor resources, occupying both cycles on a core and lines in cache. This has the potential to affect the computation phases of some applications. While our initial implementation focuses on verifying benefits of hot caching, there are a number of potential mitigation strategies. First, the heater can collaborate with the application to pause when needed. The challenge with this approach is to resume the heater in time to ensure the match list is in cache before the first access in a communication phase; this should be easily achievable in current generation bulk synchronous algorithms, where communication phases are explicitly separated from computation. Another option assuming is to gain access access to defective cores on the die that still have the potential to load data from memory into a shared cache; this would allow the heater to leverage otherwise unutilized hardware. In such a case, we need a core that is turned off for yield purposes, that is still capable of load/store operations but might otherwise be faulty, for example a bad FPU or bad register. Alternatively, this could also be supported with relative ease by device manufacturers by adding a small 1-2KiB network specific cache to the core design.

5.4 Experimental Results

In this section we present the results of several studies. First, we describe our experimental setup. Next, experimental results for all three optimizations using several different generations of processors are presented with microbenchmarks, a Sandy Bridge system, a Broadwell system, and a Knight's Landing Xeon Phi. Finally, an application study provides a detailed look at an application that makes heavy use of the match list and consequently benefited greatly from our proposed optimizations.

5.4.1 Experimental Setup

To test our proposed optimizations performance with respect to match engine length, caching behavior, and memory subsystem behavior, we identified and modified a few well known benchmarks. In particular, we tested using the OSU microbenchmarks [84] for MPI bandwidth and latency, the Mantevo mini-application [52] miniFE, the APEX benchmark AMG-2013 [7], and the SPEC benchmark Fire Dynamic Simulator (FDS) [79]. The microbenchmarks were modified in four ways: First, we added an MPI barrier to ensure that recvs were preposted. This allowed us to test the fast path of the underlying MPI implementation. Second, we cleared the cache between each iteration. This simulates a computation phase in a bulk synchronous application and allows us to do fine grained performance evaluation in that context. Third, we pinned the master thread to a specified core. This allowed for experiments utilizing shared caching behaviors on our target platforms. Finally, we added unmatched entries to the queue to evaluate performance with different receive queue lengths. The mini-apps were modified to test different receive queue lengths to test our optimizations' impacts on future communication patterns that utilize finer grained messaging. We expect the use of finer grained messaging in the future for asynchronous algorithms, communication and computation overlap, and over-decomposition [99, 12].

The experiments in this section utilize five systems to run experiments. Xeon Sandy Bridge and Broadwell systems, with Infiniband QDR and OmniPath EDR respectively, were used to evaluate large core performance. A Xeon Nahelem system with QDR Infiniband was used for additional application scaling experiments. A Xeon Phi KNL testbed was used to evaluate the MVAPICH in shared memory. Finally, the Open MPI Xeon Phi experiments used XC40 systems with KNL processors and an Aries network.

Testing for our Phi (KNL) testbed used shared memory communication for micro-benchmark results. This method of testing allows for the greatest pressure on the MPI implementation as the communication mechanism does not become a bottleneck before the MPI matching engine is stressed to its limit. This allows us to fully explore the optimizations, while also testing a key communication channel in both current and future systems, intra-node MPI communications.

Further testing on Phi (KNL), to examine internode communication in a production enviroment, requires a MPI that supports the openfabrics API (OFI). MVAPICH and MPICH do not have working OFI layers for our Aries (uGNI) network. Therefore we have adapted Open MPI, which does have an optimized OFI uGNI transport, to have a MPI queue structure like MPICH/MVAPICH. This is necessary in order to be fair for the comparison to prevent our techniques from appearing to be more effective than they really are in a production environment. Having a simple OFI interface would reduce bandwidth and increase latency of the base case and skew differences between the approaches.

While massive threading through OpenMP and similar approaches are putting pressure on systems to use less than one MPI process per core, MPI-only execution is still an important execution model to study, and these tests seek to better understand how optimizations can enable MPI-only type execution in future systems.

Testing at scale was performed on our traditional Intel Xeon clusters. All of the micro-benchmark results in this section are presented as averages and standard deviations of 10 runs. The application results are an average and standard deviation for three runs.

5.4.2 Optimizations on Xeon Sandy Bridge

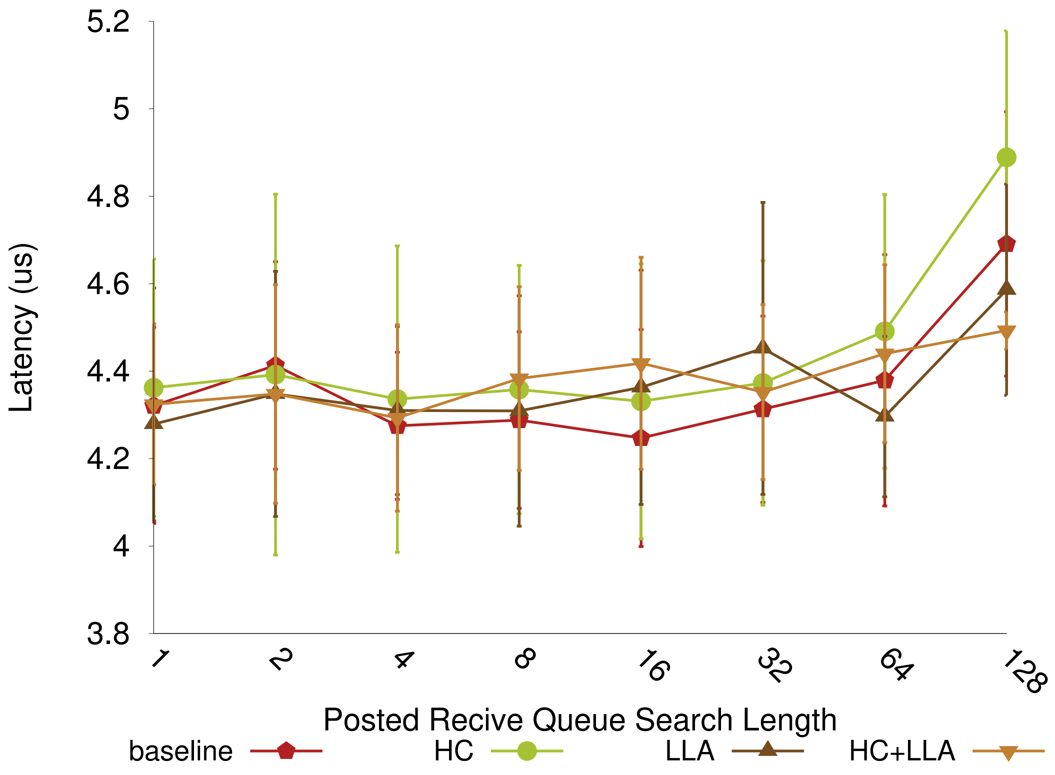

Figure 5.7: Sandy Bridge Latency

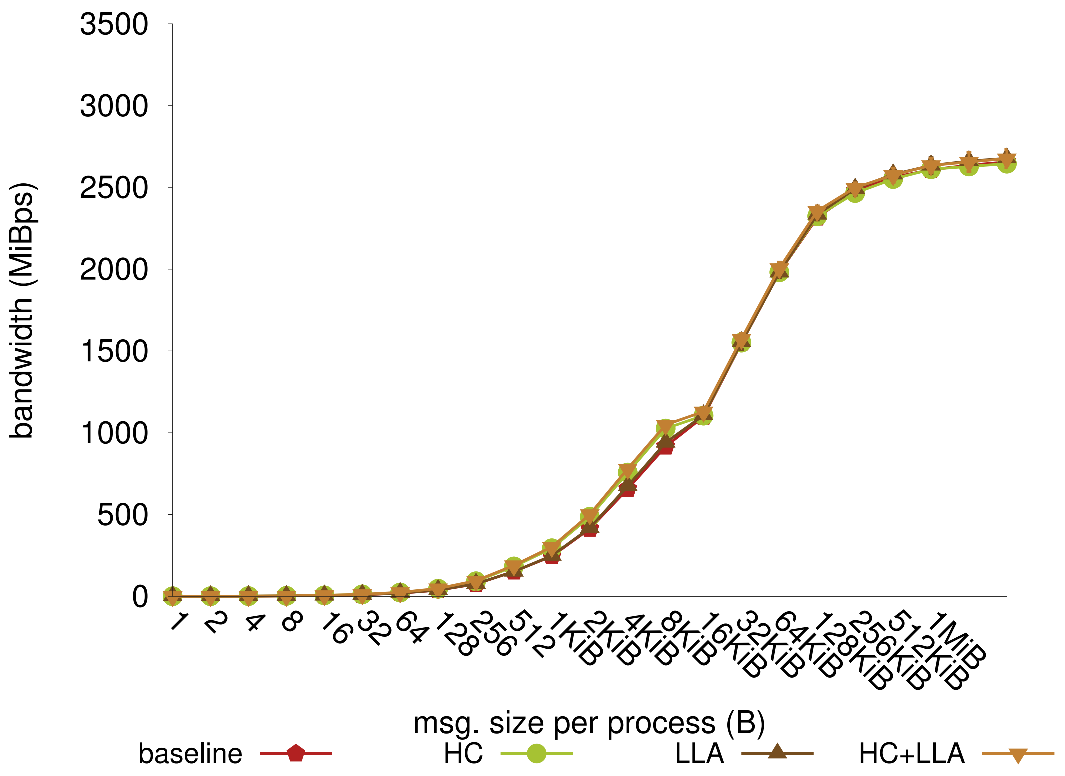

Figure 5.8: Bandwidth Performance With an Empty Matching Engine - Sandy Bridge

Our large scale platform is a dual socket Xeon Sandy Bridge machine. The AVX Vector support does not include AVX2 or AVX512 so these tests exclude that optimization. While the aim of these approaches is to improve message rate, which maps better to the bandwidth test, it's important to evaluate the impact that these approaches may have on latency. Figure 5.7 shows the latency for 1 Byte messages at small queue lengths. It's difficult to identify a pattern, before a 128 queue length, because the impact appears to be minimal and the standard deviation is quite high.

Figures 5.8 and 5.11 show bandwidth results for small and large queue depths, 0 and 1024 respectively. On both graphs, we see a slight variation on small and medium messages and a convergence once the matching engine is no longer a bottleneck at 16 KiB messages. These are further broken out in Figures 5.12 and 5.13 which show how the trends for a small, 1

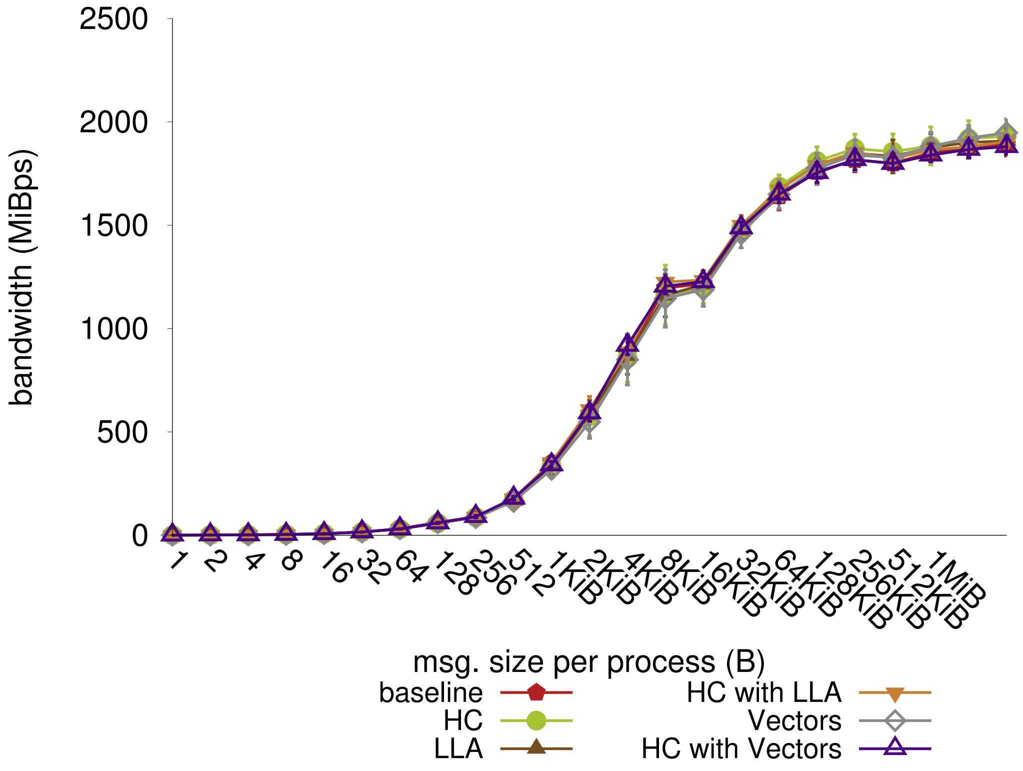

Figure 5.9: Bandwidth Performance With an Empty Matching Engine - Broadwell

Byte, message and a large, 4KiB, message as the queue length changes. We can see in both of these Figures that hot caching has an effect even at small queue lengths, and the link list of arrays increases the scalability of the list.

There are a few interesting trends to notice. First, the matching engine does not become a bottleneck until at least 256 messages are in the queue. However, we still see a marked improvement at small queue lengths when using hot caching due to the latency reductions of having the list in cache. Second, the linked list of arrays performance is often equivalent and sometimes better than the baseline linked lists with twice the elements. This implies that the cache lookup is the determining factor in the performance. As the linked list of arrays approach used here packs two entries into a single cache line sized list element, the worst case for this approach is to have every other element empty. With worst case conditions for

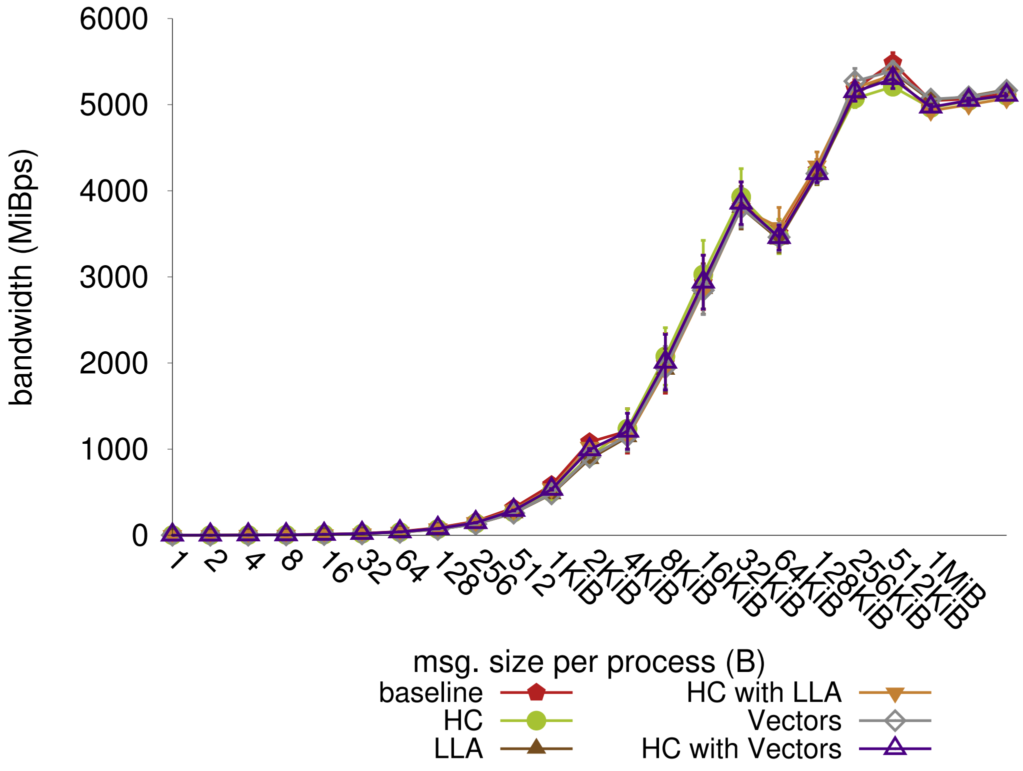

Figure 5.10: Bandwidth Performance With an Empty Matching Engine - Knight's Landing

our approach, it is still equally performant to the original implementation. Finally, while cache heating is more performant for reasonable queue lengths, it becomes less performant at large scales. This is likely due to contention and blocking due to the spinlock and cache limitations. It should be noted that it is easy to define what elements are brought into cache with the current hot caching architecture; therefore, we can establish a tunable threshold for MPI to determine where to stop cache heating.

5.4.3 Optimizations on Xeon Broadwell

The results on a Xeon Broadwell architecture differ from those seen on Sandy Bridge, which is expected as Broadwell is a complete architecture generation ahead of Sandy Bridge. Fig-

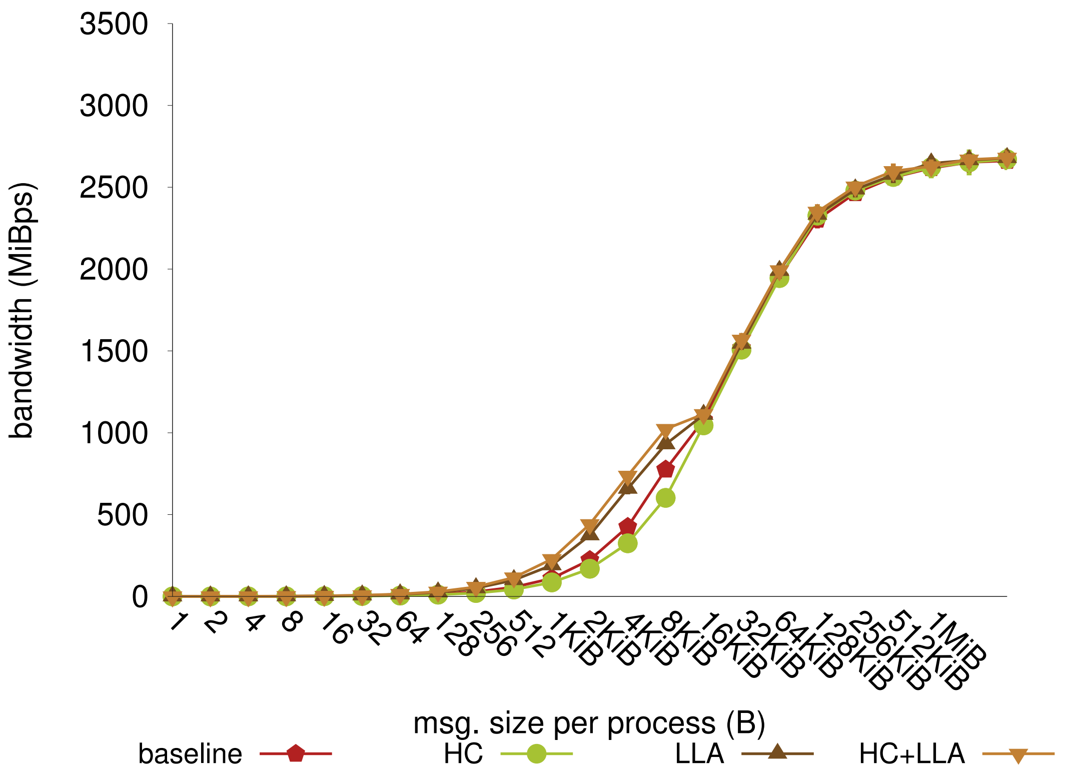

Figure 5.11: Bandwidth Performance on the Sandy Bridge Architecture - Queue Depth 1024

ures 5.9 and 5.14 show bandwidth results for a sweep of message sizes for short and long queue lengths. Figures 5.15 and 5.16 show results for a sweep of queue lengths for 1 Byte and 4 KiB messages.

There are several interesting trends to further explore in this data. First and foremost, Broadwell has support for AVX2 instructions, so our results include a version of the AVX vectors optimization. In the match length figures, the AVX technique improves the match list scaling, having no major drop-off until 4K elements are in the list. This may be primarily due to caching effects of using contiguous memory for 8 entries, but performing multiple comparisons in a single instruction is also contributing to this large performance increase. When compared to Sandy Bridge we observe a similar trend for the linked list of arrays data-structure; packing two elements into a cache line delays the match list bottleneck from

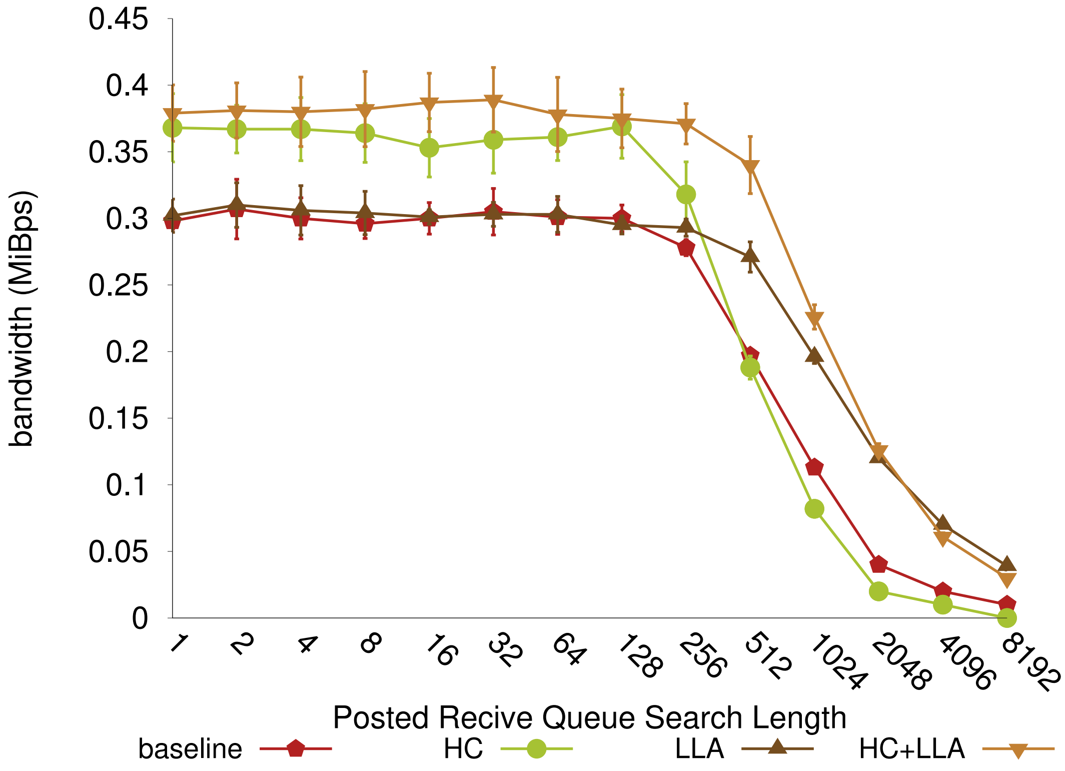

Figure 5.12: Bandwidth Performance on the Sandy Bridge Architecture - 1 Byte Messages

512 to 1024 elements. The worst case analysis (where each linked list element only contains one item) of these two approaches shows that the linked list of arrays approach will still outperform the baseline, however intrinsics will often not. This is likely due to the initial the AVX Vectors implementation requiring more cache lines and cache misses being the dominant performance bottleneck.

It is also important to note that while cache heating does show improvement in proof of concept tests, the MPI implementation appears to not be able to take advantage of it on this architecture.

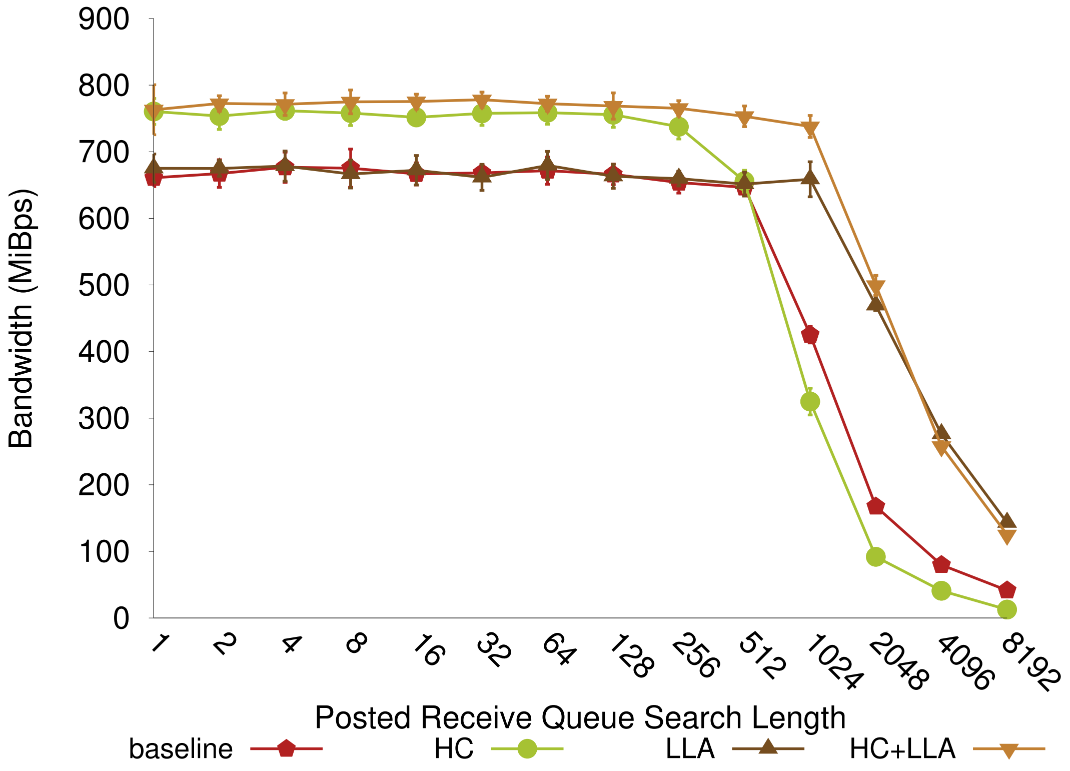

Figure 5.13: Bandwidth Performance on the Sandy Bridge Architecture - 4 KiB Messages

5.4.4 Optimizations on Xeon Phi

The Intel Xeon Phi® architecture as of the latest revision (codename Knight's Landing (KNL)) is a self hosted solution comprising many cores (64-72 depending on the part number). The Phi also provides wide vector operations of 512 bits, called AVX512. Leveraging this capability and making the cache more efficient for message matching results in excellent speedups as evidenced in Figure 5.17. The hot caching and AVX combined technique shows a 2X to 5X improvement of the baseline (5X improvement occurs at 512 byte messages). With list lengths of 4K, the speedup at 512 byte message sizes is approximately 40X over the baseline. In order to fully push the Phi with respect to small message rate, we used a single Phi using shared memory, providing the hardest case for our optimizations to prove themselves versus the baseline (high message rate for small messages at small queue lengths).

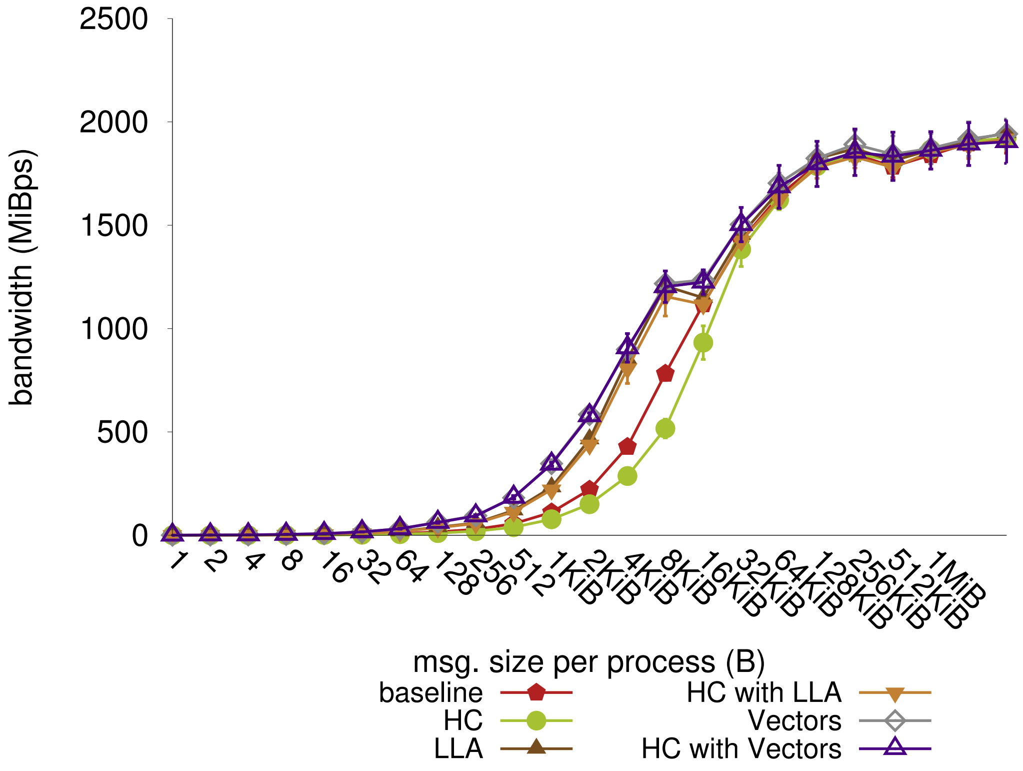

Figure 5.14: Bandwidth Performance on the Broadwell Architecture - Queue Depth 1024

The results for bandwidth with various queue lengths are shown in Figure 5.18 for 1 byte messages and 5.19 for 4KiB messages. Overall we see that for list lengths of several hundred elements that are common for sweep3D and AMR type codes (128 elements to 512 elements) the proposed techniques significantly outperform the baseline. These benefits continue for larger list sizes such as those experienced with more message matching intensive applications. However, for very small messages with short lists, the baseline can outperform our methods. It is worth noting that for halo exchange codes in 3D with 27 elements, the baseline and our hot caching and AVX solution are essentially equivalent.

Figures 5.20-5.22 shows the results of these experiments run with the Open MPI versions of our code. These were run on KNL processors over the Aries network on a Cray XC30. Unlike the experiments in shared memory, these results show the effect of network limits on

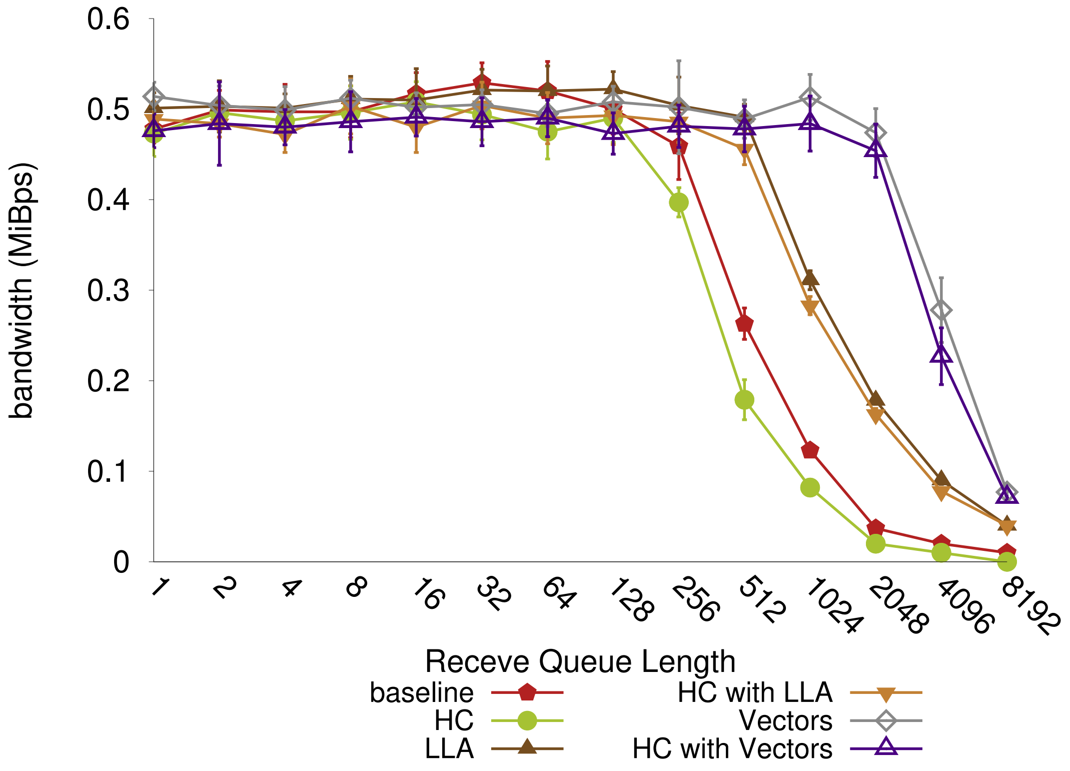

Figure 5.15: Bandwidth Performance on the Broadwell Architecture - 1 Byte Messages

small message sizes, as the bandwidth is clearly limited for short queue lengths.

From these results, we can conclude that these optimizations have great promise with Phi architectures for most codes, providing performance benefits for both intra- and inter- node communication. We expect these optimizations to be useful when large scale KNL systems become commonly available and scientific applications are adapted and optimized for the architecture.

5.4.5 Application Study

Application performance is an important metric for MPI improvements. Application performance improvements can be challenging as weak-scaling applications can have limited

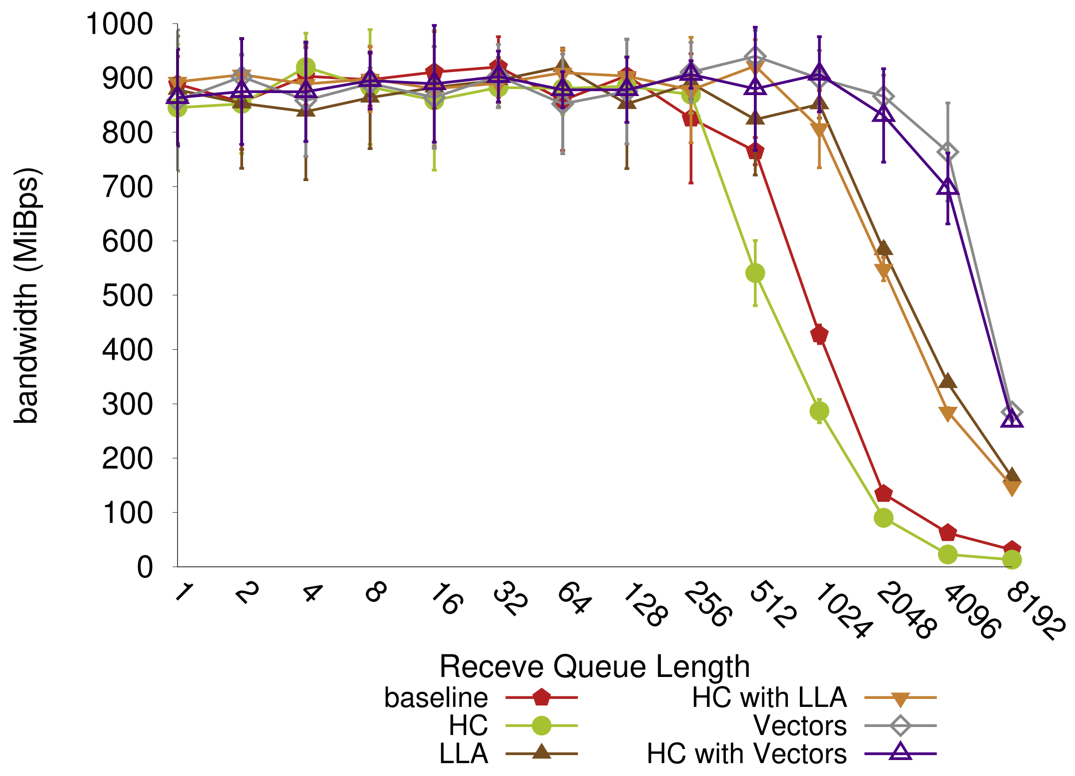

Figure 5.16: Bandwidth Performance on the Broadwell Architecture - 4 KiB Messages

MPI communication times, and strong-scaling applications require sufficient scale before communication becomes the dominant runtime component. This work examines AMG2013, MiniFE, and MiniMD, both common proxy apps for codes of interest at large capability class installations. Prior to examining these results one should note that the capabilities our improvements enable are ones that have not been traditionally supported by MPI implementations and therefore many applications are built to avoid long matching lists, even though this can make code more complicated. We do present one additional application, FDS, that due to its design will be greatly impacted by our proposed changes.

The application study focuses on one platform, our Broadwell system. This system offers some scale along with AVX2 support, which lets us test all of our proposed techniques.

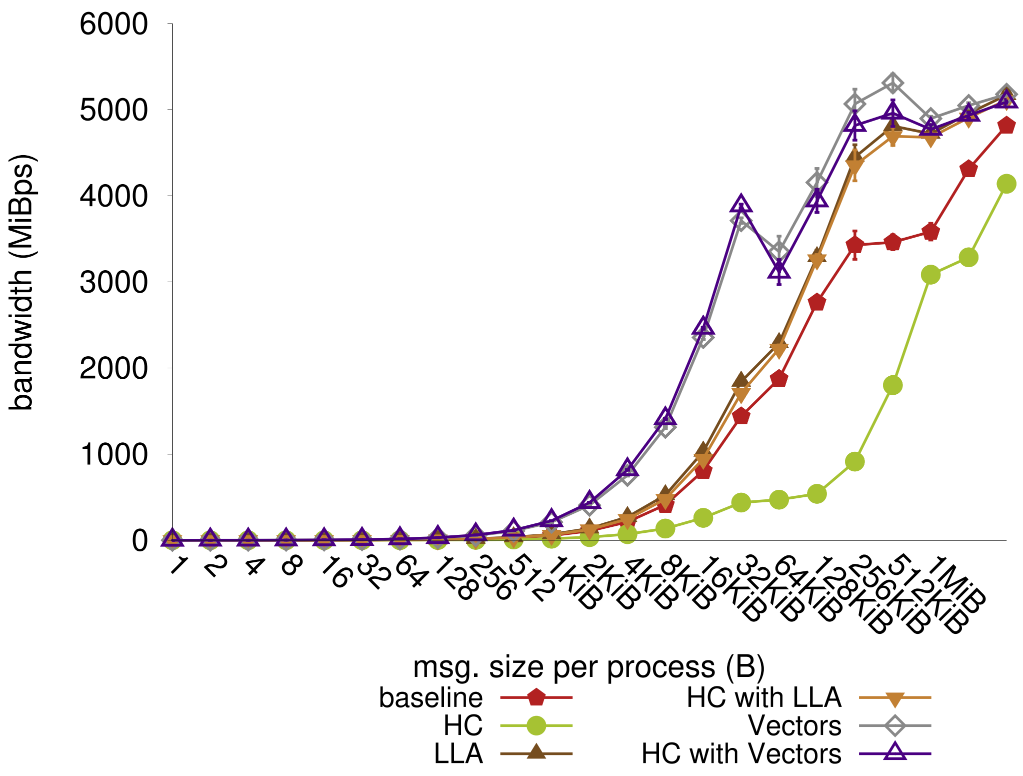

Figure 5.17: MVAPICH Bandwidth Performance on the Xeon Phi Architecture in Shared Memory - Queue Depth 1024

AMG2013

AMG2013 is an adaptive multi-grid solver that is designed for unstructured grids. AMG is a weak-scaling code that has relatively trivial load balancing in that the grid is evenly split between all processes and increasing scale corresponds with proportionally larger prob-

	AMG2013	MiniMD	FDS
Traditional	0.05925 — 2.13692	0.00021 — 0.00020	2.61983 — 724.11930
List of Arrays	0.04117 — 3.14410	0.00027 — 0.00022	2.75083 — 984.89975
Intrinsic Vectors	0.03554 — 2.09832	0.00041 — 0.00027	3.71760 — 363.37943

Table 5.1: Maximum Time, in Seconds, Spent Searching The Matching Queues on Knight's Landing Averaged Across Three Runs (Posted Receives — Unexpected Messages)

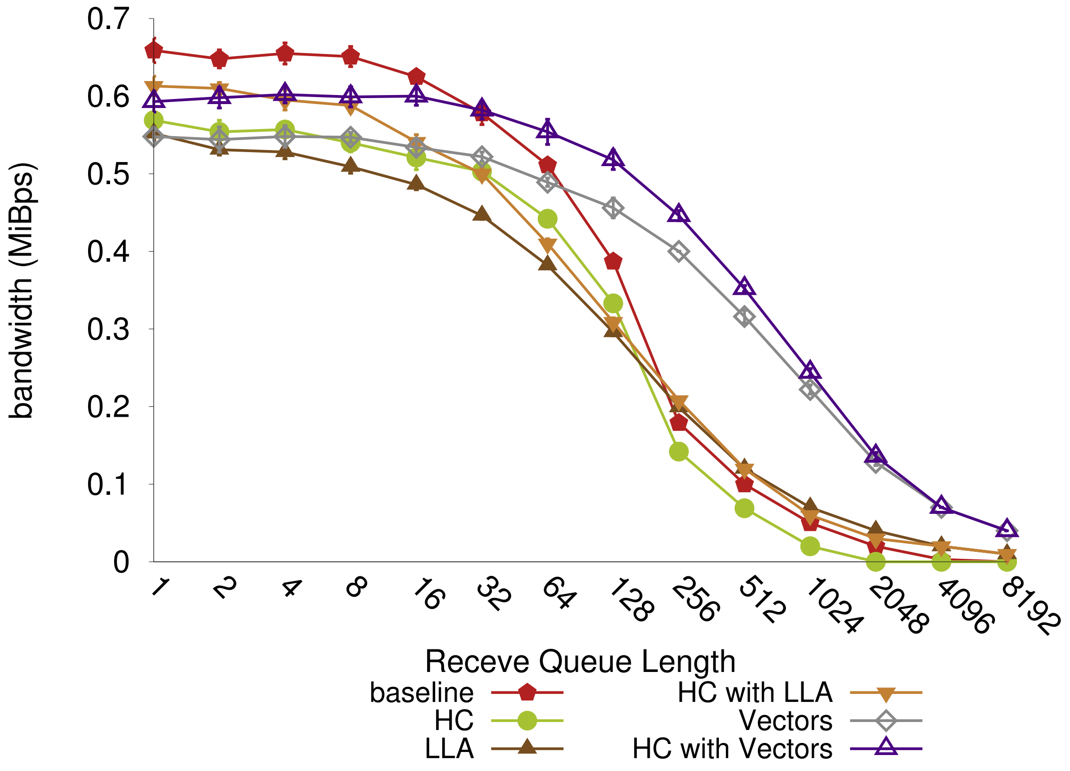

Figure 5.18: MVAPICH Bandwidth Performance on the Xeon Phi Architecture in Shared Memory - 1 Byte Messages

lems/grids. AMG is very memory intensive and requires occasional large message bandwidth. While AMG can be forced to send many small messages if configured in an unrealistic manner, we have used the configuration recommended by the US DOE when AMG has been used for acceptance testing on new systems. Therefore, AMG is more bandwidth sensitive than message rate sensitive.

The results in Figure 5.23 shows scaling results from using the recommended large problem. These runs are not of large enough scale to show any clear trends for the improvements, but show runtime improvements for LLA at 2.9% and intrinsics of 0.7% at 1024 processes. This is unsurprising as the results in Section 5.4.3 indicate that queue sizes need to be close to 512 to see large benefits. It is still important to note, however, that these approaches don't nega-

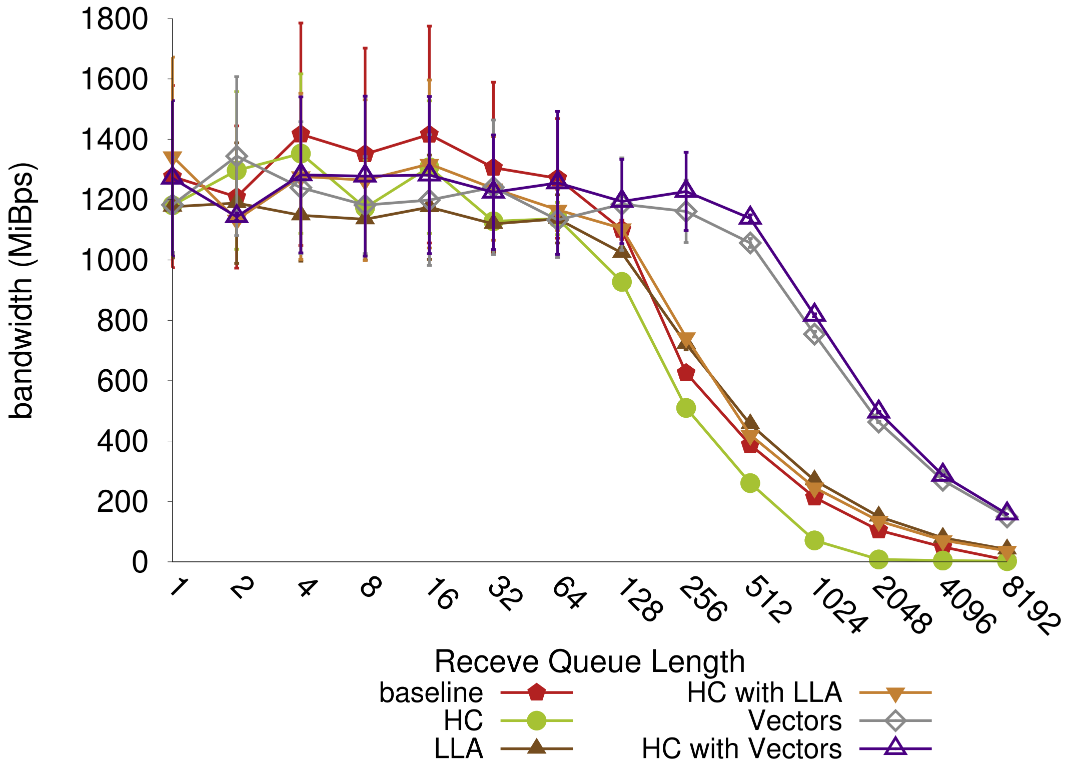

Figure 5.19: MVAPICH Bandwidth Performance on the Xeon Phi Architecture in Shared Memory - 4 KiB Messages

tively impact modern application performance, and improvements of single percentage points in runtimes for applications due to MPI improvements are within the expected performance enhancement range.

On a 64 node KNL system, we observed 2.1% of runtime spent in the matching engine. This test used the same problem as the our experiments on Broadwell, the Open MPI baseline implementation and was run with 64 processes per node for a total of 4196 processes. The total runtime was significantly increased from the Broadwell tests, taking 58.942 seconds. We were able to observe an 8.3% matching improvement for the small recommended problem at these scales on the KNL system. Table 5.1 shows the time spent matching for the various optimizations.

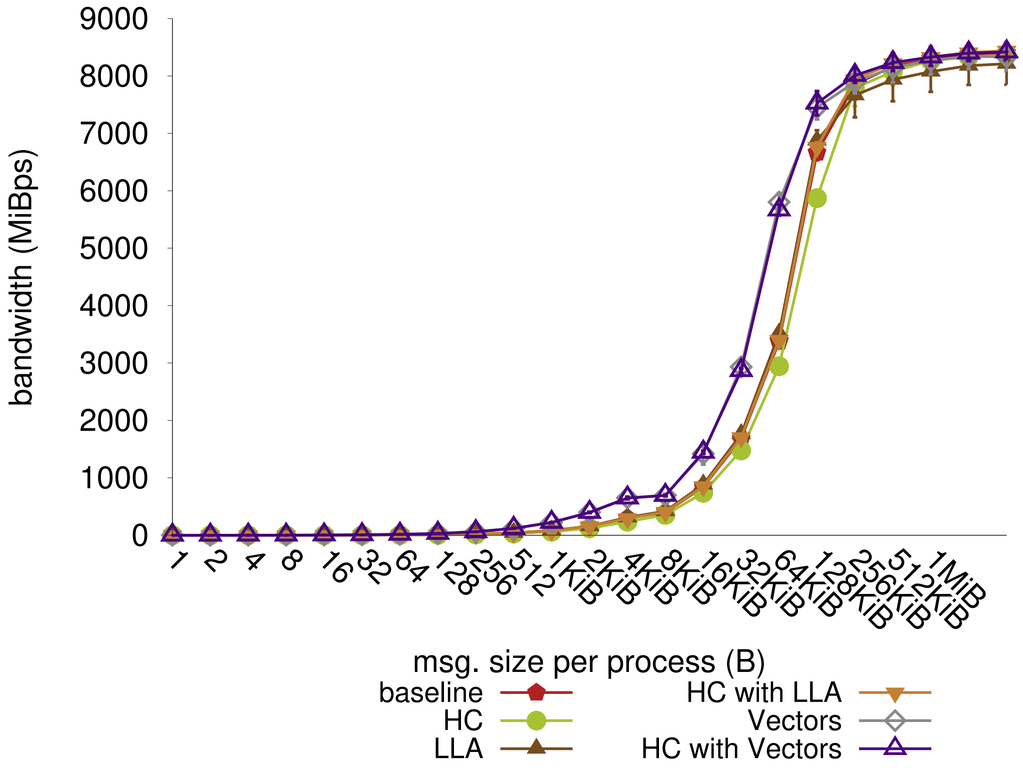

Figure 5.20: Open MPI Bandwidth Performance on the Xeon Phi Architecture With an Aries Network - Queue Depth 1024

MiniFE and MiniMD

To explore any potential effects these optimizations might have on optimized large scale codes, we examined two mini-applications in the Mantevo suite [52]. MiniFE is a unstructured implicit finite elements simulation mini-application that's primary primary computation is a conjugate gradient solver. MiniMD is a molecular dynamics simulation mini-application that is designed as a smaller version of the LAMMPS simulator, concentrating on solving a Lennard Jones liquid simulation. Both mini-applications use the bulk-synchronous halo-exchange communication pattern.

The results in Figure 5.24 show MiniFE on a Broadwell system at a fixed size of 512 processes and a problem size of 1320^3, for various posted receive queue lengths. The results here are

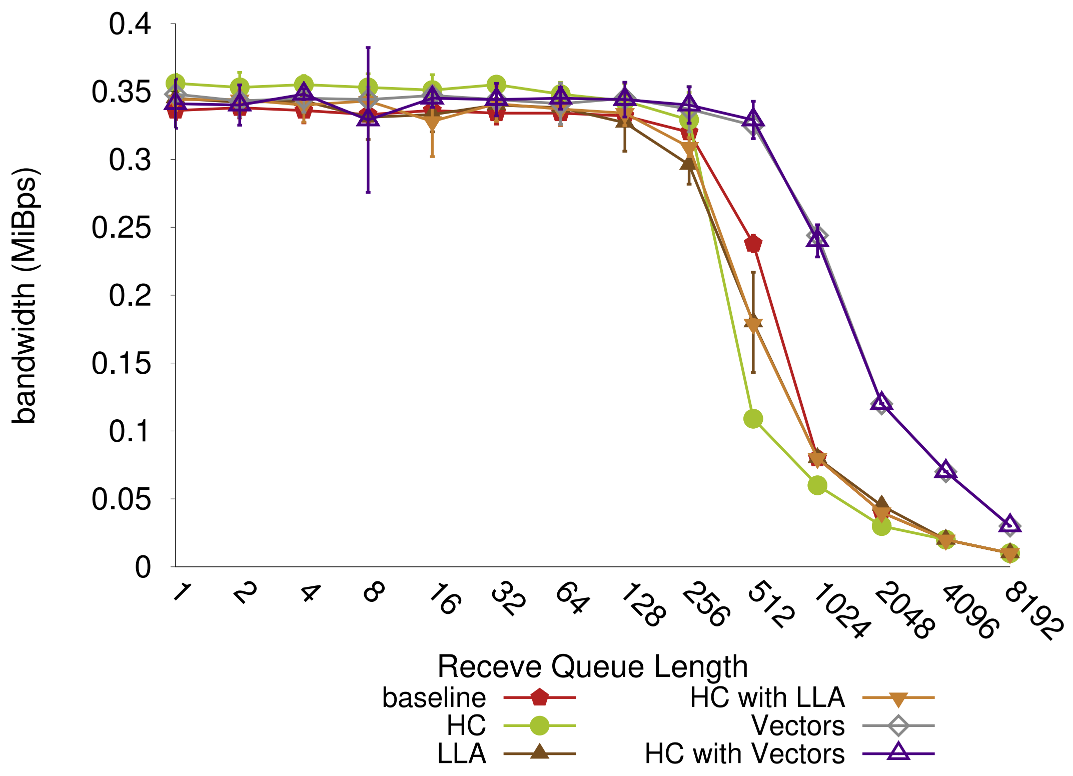

Figure 5.21: Open MPI Bandwidth Performance on the Xeon Phi Architecture With an Aries Network - 1 Byte Messages

similar to the AMG2013 results, in that there is a small but not insignificant improvement from our techniques. Using LLA at 2048 queue sizes results in a 2.3% improvement to runtime while intrinsics show a 1.3% improvement in runtime. Table 5.1 show the MiniMD match list times for the Open MPI implementation on a KNL Cray XC30 system.

Both of these experiments don't show much effect from the improvements. This is due to the communication pattern requiring a limited number and frequency of messages. For example, our MiniMD experiments spent a maximum of 0.3% of their time in the matching engine.

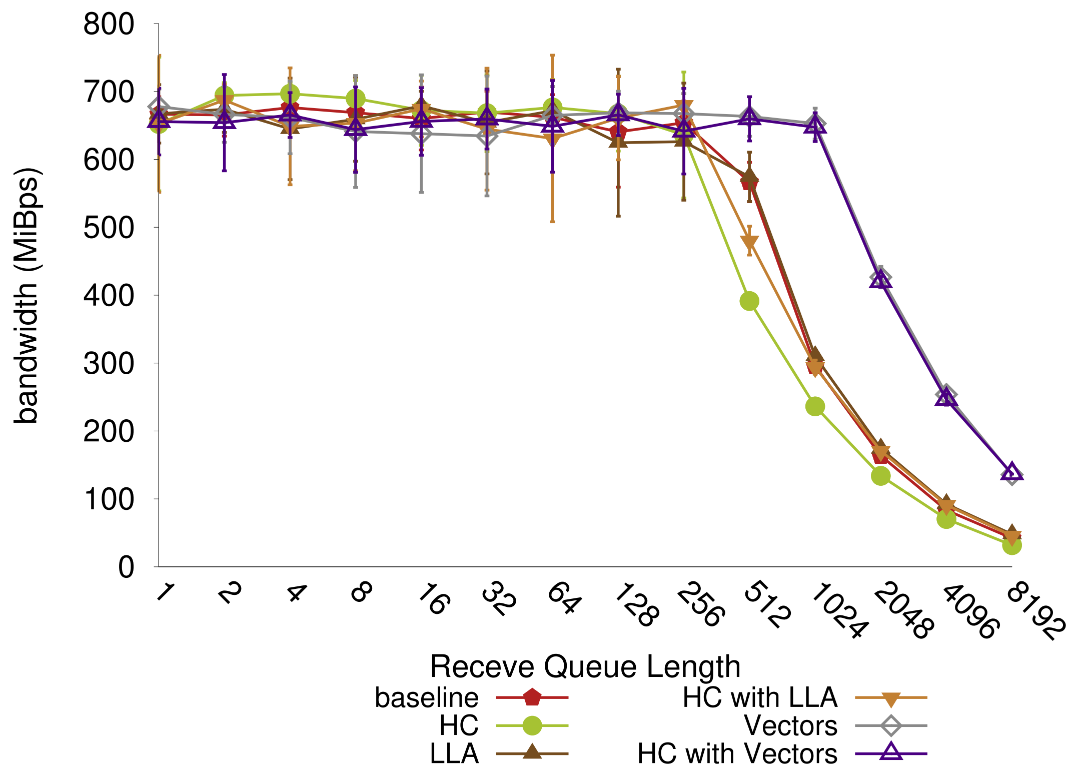

Figure 5.22: Open MPI Bandwidth Performance on the Xeon Phi Architecture With an Aries Network - 4 KiB Messages

5.4.6 Full Application Study

Typical MPI message matching lists have been shown to be in the hundreds of matching list elements for many application communication patterns, however some applications can build list sizes to much larger lengths. In addition, the manner in which these potentially large lists are searched can lead to further performance bottlenecks in MPI. One application that exhibits these behaviors is the Fire Dynamics Simulator (FDS) [79], a benchmark in the SPEC MPI benchmarking suite.

Figure 5.25 shows results from three different tests: linked list of arrays and AVX vectors on Broadwell, hot caching, linked list of arrays, and a combination of the two along with linked lists of large arrays at scale on a Nehalem cluster. Each result is presented as a factor

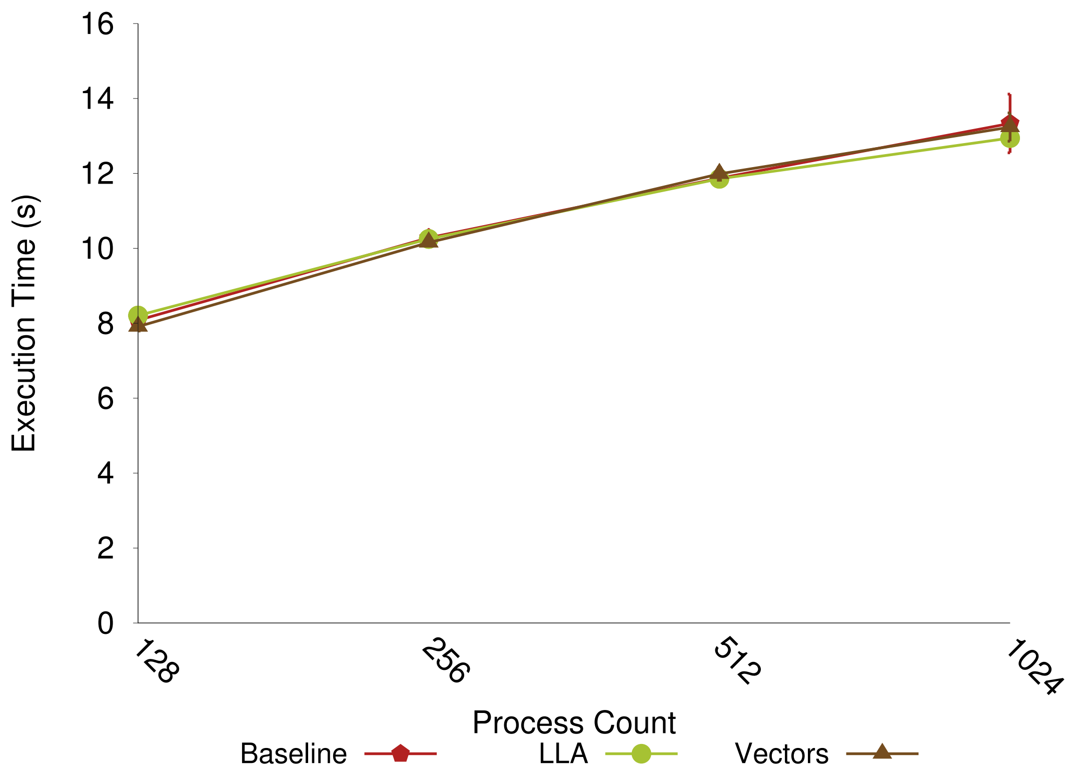

Figure 5.23: AMG2013 Scaling Results for Broadwell

improvement compared to its baseline. For Broadwell, we ran from 128 to 1024 processes, and see a marked performance increase at 1024 with 1.21x for linked list of arrays and 1.16x for vectors. To examine this at larger scale, we ran on a Nehalem cluster. Because we were able to tune hot caching for Nehalem, results for this cluster show hot caching, linked list of arrays, and hot caching with a linked list of arrays. For the two techniques using the linked list of arrays approach we see a further divergence as we scale up resulting in a 2x speed-up at 4k processes. While we see a slow down from the general hot caching approach, hot caching in combination with the linked list of arrays shows a significant speed up at smaller scale. This is due to lock contention as we must remove elements from the hot caching list before MPI can deallocate them. Hot Caching shows a significant improvement at smaller scales. At 1024 processes, linked list of arrays with hot caching performs 14.5% better than the base

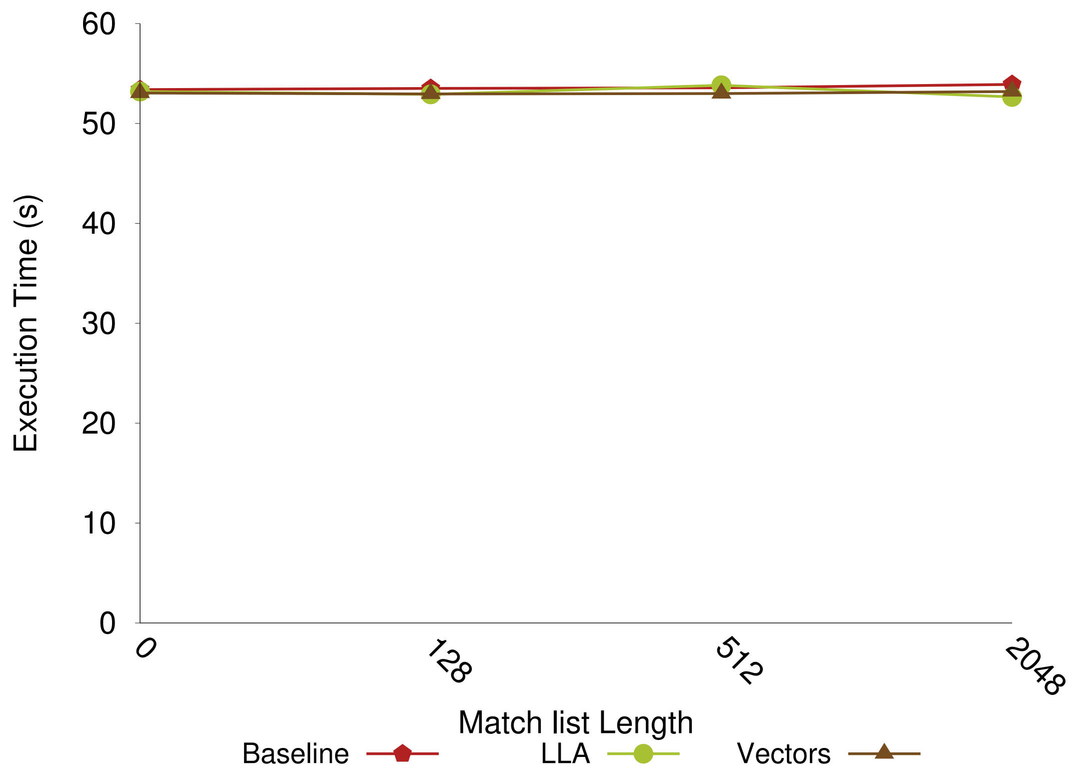

Figure 5.24: MiniFE Results at 512 processes with varying match list lengths for Broadwell

line and has a 10.4% improvement over the linked list of arrays alone. To further optimize at scale, we examined an early linked list of large arrays approach using MVAPICH2 2.0. We can observe that the large arrays results speed up FDS at 8192 MPI processes, where we observe a 3x speed up.

Figure 5.26 shows FDS performance using the Open MPI implementation on a Cray XC30 system. In this case, the linked list of arrays approach does worse than the baseline. This is not entirely unexpected, as the microbenchmarks showed a trade off for this approach that slightly reduced short queue length performance when communicating within a node and this application was run with 64 processes per node. The vector approach improves performance over the baseline due to its improved performance on long lists and smaller performance overhead on short lists.

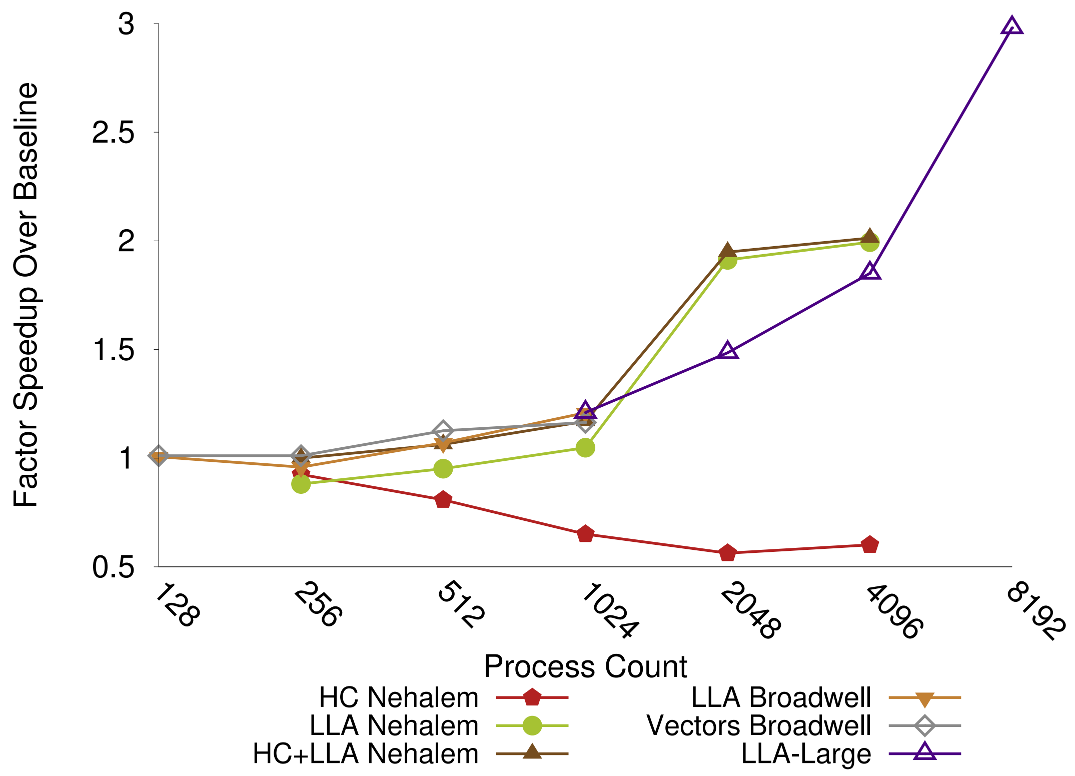

Figure 5.25: Fire Dynamics Simulator Scaling Results For Three Experiments on Xeon

FDS is a widely used code in its subdomain in addition to being part of the SPEC-MPI benchmark suite. It is representative of code behaviors that are expected in future applications. It builds up large match lists and does not typically match the first element in the list. This type of behavior is more representative of what would be expected when using many unsynchronized threads for compute and communication. The lists will grow as thread counts in future systems increase and each thread will interact with the communication subsystem. The lack of synchronization of the threads for communication will also produce more random-like distributions of match entries in the match list, producing behavior more similar to the match list distributions of FDS today. Therefore, we find that these techniques can be very effective for expected future code behaviors, which is encouraging as there is enough time to influence architectures to adopt efficient vector techniques for matching in a

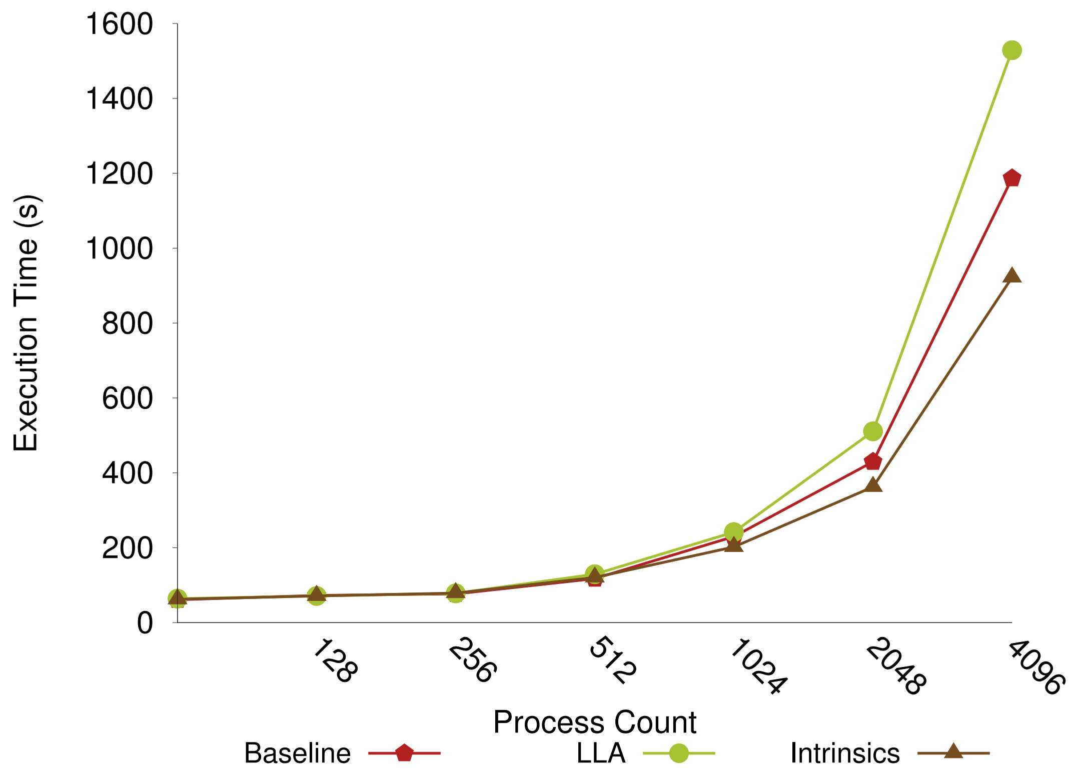

Figure 5.26: Fire Dynamics Simulator Scaling Results For Xeon Phi

time frame that is applicable to this class of codes.

5.4.7 Discussion

This work is targeted at enabling behaviors that are expected to be widely adopted in future applications. The results for applications show that both the LLA and vector approaches are essential options based on the architecture on which code is running. Hot caching shows benefits in combination with other approaches, which is to be expected as it increases efficiency of the approaches by increasing critical data locality.

The results from our methods illustrate why certain methods are effective on different architectures, requiring us to have multiple complimentary methods. Firstly, the LLA method is effective with traditional large cores due to the deep out-of-order instruction piplelining and sophisticated prefetching with large caches. The Phi has significantly shallower out-of-orderness (two instructions) and smaller cache (2MB L2). However, the Phi has a wider vector unit than Broadwell, which allows it to take better advantage of vector comparison operations. This architectural feature comparison is clearly illustrated in the microbenchmark and application results for the different platforms. The LLA method is superior for Broadwell architectures, and with a high frequency (almost double that of Phi for vector operations) vector operations also work well. For the Phi, the vector operation method is the most effective, as the LLA method cannot be fully taken advantage of by the Phi's smaller cache and less sophisticated pre-fetcher.

We have demonstrated that these methods are useful for all major open source MPI implementations, from which all common commercial MPI libraries derive. By applying multiple different approaches to MPI implementations that can be implemented and decided between at run-time, we have demonstrated that we can effectively use current generation architectural features to accelerate MPI communications.

Our work also demonstrates how relatively small architectural improvements could significantly speed up MPI communication. Firstly, we propose adding a very wide vector unit (ARM SVE allows for up to 4kb vectors) that has a very limited set of capabilities, integer compare and bit masking are sufficient to enable communication acceleration. For convenience from a programming standpoint, it is also useful to add a rotate function (on a per register basis). In addition to this, we propose allowing access to caches on deactivated or faulty cores that are on die but not currently usable for yield purposes. These cores can be useful even if they can only perform load and store operations (leaving many functions un-needed, which provides for high yield by allowing for manufacturing defects).

We proposed new methods of exploiting architectural features by designing middleware with conscious effort paid to how to exploit new generations of architectural features. In addition to this, we propose new architectural features that could be developed based on these experimental results. These proposals leave a large exploration space open for future work, in exploring required capabilities further as well as sizing new vector units. We also propose using existing on-die resources that would otherwise be inactive in cases where such capabilities could be used. This creates further opportunities for device manufacturers to provide added value within given hardware binning ranges.

5.5 Conclusions and Future Work

In this chapter we have designed, implemented and evaluated three different methods to improve the performance of MPI message matching. The three methods show that when combined they can provide performance benefits in terms of bandwidth and latency over all list lengths and message sizes. The methods are complimentary and flexible enough to be adapted to several different use cases, while maintaining critical zero-length queue search times.. We have shown the impact of all three techniques on applications at different scales and have explored a full real-world application case in which it is possible to achieve over 3X speedup. We have shown that speedups in matching performance can increase bandwidth for cases of large list lengths by up to 40X for the Phi and can typically produce 2X-5X speedups on other architectures for common message sizes.

Chapter 6

Optimizations For Future Matching Engine Architectures

6.1 Introduction

As described in Chapter 1, new processor hardware features and programing paradigms are emerging in HPC architectures. Chapter 5 presented an architecture that can improve matching for current MPI applications by leveraging cutting-edge processor features. This Chapter presents exploratory work that further improves MPI matching to address the challenges of future multi-threaded applications and leverage upcoming hardware features. This work addresses the research challenge of optimizing the performance of communication libraries for future architectures.

The contributions of this chapter are the following:

- Tail Queues, a novel parallel matching approach;
- Fuzzy Matching, an optimistic matching technique that leverages new SIMD vector operations to increase matching parallelism; and,

UMQ	Unexpected Message Queue
PRQ	Posted Receive Queue
UMQT	Unexpected Message Queue Tail
PRQT	Posted Receive Queue Tail
ML	Matching Lock
PRQL	Posted Receive Queue Lock
UMQL	Unexpected Message Queue Lock
TL	Tail Lock

Table 6.1: Algorithm Terminology

- A performance evaluation of these two approaches.

This chapter is structured as follows: Section 6.2 presents the Tail Queues architecture. Section 6.3 presents the fuzzy matching architecture. Finally, Section 6.4 presents results of proof-of-concept implementations that examine the viability of these techniques.

6.2 Tail Queues

As metioned in Chapter 2, many-core architectures have increased the demand for a performant $MPI\_THREAD\_MULTIPLE$ implementation. Tail Queues is our proposed approach to reduce the critical section in matching, allowing for more performance in hybrid applications that utilize fine grained messaging patterns. It reduces the matching critical section by creating inboxes that isolate the potential race conditions of this data structure. This allows the lists to be parallelized independently. Most current hybrid applications avoid using $MPI\_THREAD\_MULTIPLE$ by limiting multi-threading to computations phases in Bulk Synchronous Parallel (BSP) applications. This is done as $MPI\_THREAD\_MULTIPLE$ has been un-performant when compared to single threaded implementations.

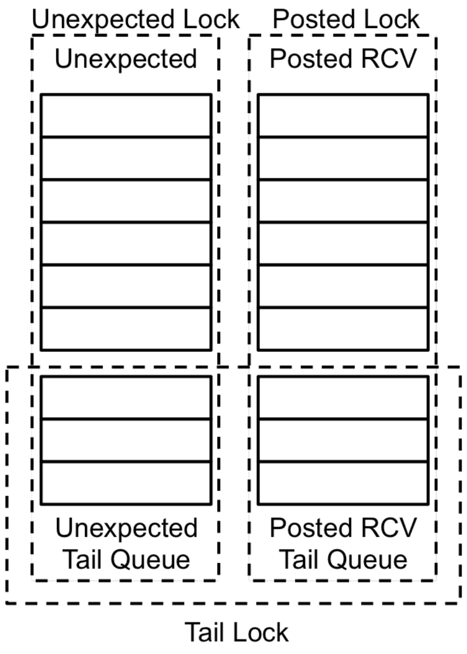

Figure 6.1: Tail Queues Locking

6.2.1 The Tail Queues Architecture

The data structures that the MPI matching engine uses have a number of challenges that make matching difficult to parallelize. The wild card and ordering constraints in addition to a race condition caused by the list dependencies make it difficult to utilize any list parallelism techniques. Tail Queues aims to provide more parallelism by isolating the shared critical section to the smallest portion of the race condition. The lists can then be parallelized in any manner that respects wild cards and ordering. Table 6.1 presents terminology that will be used throughout the rest of this section.

Figure 6.1 is a visual representation the concept behind Tail Queues. This technique is achieved by creating inbox queues that semantically represent the end of each respective list. These inboxes segment the list in a way that allows us to append new entries without locking the other list.

The baseline we are using is MPICH CH3's matching engine, this engine is often found in many derivatives of that codebase, such as MVAPICH, Cray MPI, and Intel MPI. Because it uses one big list, it has three matching parameters: a user given tag, an expected source, and the communicator's context_id. With fine grained locks, the matching lock (ML) is locked upon entry and released on exit. The function then searches the unexpected message queue (UMQ), removes the first match and returns it or if no match is found in the UMQ, it appends a new entry to the posted receive queue (PRQ). For incoming messages, the process is similar, but the queues are switched.

Algorithm 2 shows the same function implemented for Tail Queues. It now acquires the unexpected message queue lock (UMQL) upon entry and unlocks it upon exit. It searches the UMQ as usual, however, if a match is not found, it acquires the tail lock (TL) and continues the search through the unexpected message queue tail (UMQT). It removes all elements in said inbox and appends them to the UMQ itself. If it still has not found a match it then appends a new entry to the posted receive queue tail (PRQT). If the TL is locked, upon exit the processes unlocks it.

This algorithm has a number of advantages. The basic implementation allows two threads to search the PRQ and UMQ simultaneously, and exposes the potential for further parallelization of the PRQ and UMQ. For some of the results in this section, we implemented a simple list parallelization in the form of binning. In this case, our approach creates a bin around the first 50 elements which allows for two threads to be working on the same list simultaneously. It also has the advantage of being compatible with the current state-of-the-art matching implementations. Because the PRQ and UMQ are fundamentally two structures, the technique of isolating the critical section using inboxes can be used with many different implementations including Open MPI's array of linked lists and proposed hashing approaches [69, 40].

6.3 Fuzzy Matching

Fuzzy Matching is an optimistic matching engine approach that leverages small integers to increase the level of SIMD parallelism available to the vector matching technique presented in Chapter 5. Section 6.3.1 presents the details of the Fuzzy Matching approach. Section 6.3.2 describes the new small integer vector instructions that this approach leverages. Section 6.3.3 presents an analysis of applications' use of tag and ranks space to examine whether current applications could benefit from this approach. A more complete performance evaluation is presented in Section 6.4.

6.3.1 Approach

Fuzzy Matching leverages small integer vector instructions to further parallelize the vector matching engine. Small integer vector instructions divide the bits of a vector into more elements that have fewer bits. These instructions are discussed in depth in Section 6.3.2. To utilize these instructions in matching, an implementation must compress tags and ranks for matching to fit in the new integer sizes. This compression could cause a loss of meaningful data. To address this, an initial match has to be verified through a full match process. This optimistic matching technique creates a trade-off between using small integers for a faster initial match and the increased work resulting from false positives in the initial matching process.

Fuzzy Matching creates a new matching identifier for filtering a search. This fast match identifier combines the tag and rank data of a request into a small representation. While there are number of possible compression and combination techniques, our initial implementation uses windows of the lowest-order bits for tag and rank and combines them at an offset into a single integer. The lower order bit-window was chosen as it relates to a high-pass filter, preserving local identification for ranks and will preserve more variation within identifiers decreasing the probability of false positives.

The process of searching in Fuzzy Matching can be broken down into two phases. The first phase utilizes the vector unit to perform a fast parallel filter to identify potential matches based on a subset of the tag bits and rank bits. The second phase evaluates the potential matches individually to check for false positives. If no match is found, the implementation repeats phases on the next set of fast match identifiers.

When this technique is leveraged in environments with few false positives, an implementation can quickly search for matches. The false positive rate for a particular application is determined by its communication pattern and the utilized filter function. While a number of filter functions are possible, our initial implementation uses windows of the lowest-order bits for tag and rank and combines them into a single integer.

6.3.2 New Vector Instructions

Advanced Vector Instructions

Intel's Advanced Vector Instructions (AVX) are an extension of the X86 architecture. These were first supported on the Sandy Bridge architectures. These were developed to support single instruction multiple data (SIMD) parallelism. The Intel Xeon Phi architecture introduced AVX-512, a 512-bit version of AVX instructions. The Knight's Landing models were the first to support this feature, and featured the following modules: AVX-512 Foundation, Conflict Detection, Exponential Reciprocal, and Pre Fetch. This initial version had support for 32 bit integers with compare, bit-mask, and load instructions which allowed us to build a vector matching framework.

The AVX-512 included with processors such as Sky Lake include enhanced integer support in the Bytes Words module. This module supports smaller integer sizes, allowing for more entries. This allows for increased levels of parallelism in vector operations allowing for a vector operation to operate on 16 32-bit integers, 32 16-bit integers, and 64 8-bit integers.

Scaleable Vector Extensions

ARM's Scaleable Vector Extensions is a vector processing extension to the AArch64 architecture. While this architecture is early in its lifecycle, it proposes the ability to incorporate up to 2048-bit vectors. This would allow for 64 32-bit integers to be simultaneously evaluated. As with the AVX-512 Bytes Words module, SVE allows for different subdivisions of the vectors from 8-bit to 128-bit.

6.3.3 Application Tag and Rank Characteristics

One of the challenges of this approach is to explore the trade-off between the improved fast matching performance and the cost of verification and false positive fast match results. To gain a better understanding of this trade off space, Whit Schonbein, collected tag and rank data from a representative sample of proxy applications and full applications [94]. This data-set helps identify the how the application behavior scales, using three scales for each application, 256, 512, and 1024 MPI processes. This data set contains information on on the bits amount of bits used by each processes in an application. Three values are particularly important to our evaluation:

- Maximum theoretical minimum (MTM) is the smallest window of bits, starting at the least significant bit, needed to ensure an exact match;
- An overlap rate for each combination of application and bit-window size, without remapping.

This data is available for both tag and rank for each application profiled. MTM represents the bits required for a naive fuzzy matching implementation with no false positives. MPDW represents the theoretical potential for fuzzy matching given an ideal re-mapping strategy. The overlap rate takes into account the tags and ranks used for communication on a single process over the entire run. This is an approximation of a potential false positive rate as it ignores any temporal correlation of messages.

To analyze this data, I examined the data set with the features mentioned above for patterns in the use of tags and ranks. Tags have more potential for limiting the number of bits as the number of ranks will scale with the size of a job. I looked for a couple different patterns in tag usage. For example, scaleable tag usage, scaling tag usage, and sparse tag usage are patterns that impact the viability of fuzzy matching for different applications. Rank usage was similarly examined; however, ranks were mostly interesting when looking at non-exact matching as the bit over lap in ranks used in common communication patterns often scale with the job size.

Tag and Rank usage Both tag usage and rank usage for these apps have a few key take-aways. In general, many of these codes can benefit from fuzzy matching while still maintaining a perfect match. This is especially true for tag-space. For example, highly tuned codes, such as HPCG and MiniMD both use only a single tag. Most codes use a reasonable number of bits for their tags, but can be fully determined by a smaller number of bits. For example, AMG 2013 uses 11 bit tags, but has only requires a window of 5 bits for perfect matching. Finally, there were a couple of codes, kripke, neckbone, and FDS, that used a larger number of tag-bits, particularly in a way that grew with the number of processes. This last case, unlike the former two may not be ideal for fuzzy matching a the trade-offs and tuning will change at different scales.

Maximum theoretical minimum MTM is the number of bits needed for a perfect match given an ideal fast match representation. For tag space, this shows the potential to re-map some of the less scaleable behaviors, such as kripke and neckbone, into a smaller and more efficient space that can better leverage fuzzy matching. For rank-space this shows the potential to significantly reduce the number of bits needed to perfectly match a rank. For example, hpcg, kripke, minighost, minimd, and nekbone see a significant reduction in number of bits needed to ensure a perfect match in rank space. This is due to their sparse communication patterns; a 3 dimensional halo exchange like MiniMD communicates in six points which given an ideal fast mach representation would only require 3 bits, but because of the structure of the problem decomposition, does not map as well into a least significant bit window. The challenge this presents is to find better fast match representation.

False positive trade-offs As with most optimistic approaches, the goal of fuzzy matching is not to guarantee perfect matching, but to provide a fast approximation that can be performant with an acceptable number of false positives. Given this, it is interesting to note that the false positive rate for certain applications declines rapidly as we increase the size of the matching window. For example, if we limit the rank-space to 5% false positive rate, all four cases of AMG2013 can be represented by 5 bits. This is highly application dependent as multi-dimensional halo exchanges, such as MiniMD are characterized by periodic large drops in false positive rate as the window size increases, reducing the benefit of tolerating false positives.

6.4 Experimental Results

In this section, I present the preliminary results for the two approaches described in this chapter. Both approaches are designed for future environments and the results in presented in this chapter are limited proof-of-concept designs tested in today's HPC environment. Section 6.4.1 presents the results of the initial Tail Queues evaluation. Section 6.4.2 shows the results of a KNL-based fuzzy matching implementation.

6.4.1 Tail Queues Results

This section presents an experimental study of two implementations of Tail Queues. This section will present the details of the experimental setup, the results of the experiments, and the implications of the results.

Experimental Methodology

To measure the potential impact of the Tail Queues algorithm, we created two implementations of the algorithm. First, we built a simple prototype of the algorithm external to MPI.

Prototype To examine the performance characteristics in detail, we built a self contained prototype implementation. This prototype implements Tail Queues and baseline of a traditional matching engine protected by a single lock. To ensure that the prototype would fully exercise the data structures, we used MCS scalable locks [80] to help avoid things that could falsely impact the results, like thread starvation. These two implementations use the same linked list and lock implementations to make a fair comparison. This prototype allows us to study how the match list can perform with varying amounts of parallelism without the need to manage coarse grained locks in sections of existing MPI libraries. In addition, this provides a high search rate that pushes the locking mechanisms with high intensity, potentially highlighting any concurrency issues that could arise from the Tail Queues method.

We built a benchmark to compare the two implementations. The benchmark starts by posting a given number of posted receive entries and unexpected message entries, to evaluate the algorithms at different queue lengths. The driver then starts a given number of threads. Each thread alternates simulating posting of a receive and the arrival of corresponding incoming message. This experimental setup allowed us to explore the performance characteristics of each implementation by isolating the matching rate.

The platform for these tests was a small scale testbed cluster at Sandia National Laboratories. It is a Dual Socket Intel Xeon Sandy Bridge, 2.6 GHz 8-core cluster with two Xeon Phi Knight's Corner accelerator cards in each node. The Xeon tests were run on the node and the Xeon Phi tests were run in a shell on the accelerator. The data associated with these test are the mean and standard deviation of 10 runs. Unless stated otherwise the Queue Length independent variable represents the size of both queues.

MPICH To further evaluate this technique we implemented Tail Queues in MPICH 3.1.4. This was done using the internal linked list data structures and locking mechanism to be consistent with the MPICH baseline. Additionally, these tests show the results of independent linked list parallelization in the form of binning. To test this we modified the Sandia Microbenchmarks (SMBs) [32] to use multithreading and have different queue lengths. The SMBs aim to capture real world message rate. They accomplish this in a number of ways including making parallel transfers, clearing cache between iterations, and testing different communication patterns. These tests represent the message rate of the single direction test for 8 processes. The SMBs' single direction is similar to a multi-bandwidth test, where half of the processes are designated as senders and send multiple successive messages to an assigned receiving peer.

The tests shown in this section were run using the shared memory netmod. This allowed us to ensure that we captured the performance characteristics of each approach by removing the older InfiniBand network as a bottleneck. Additionally, the shared memory netmod currently supports fine grained locks. This provides the hardest experimental conditions for the Tail Queues locks, as they are exercised to the highest possible level of intensity on our test system. The MPICH tests were run a single node with an I7 quad core processor. This limits the amount of parallelism available below the number of threads required by the SMBs (8 processors with 2 threads each) and thus these results should be a worst case for these techniques. These results are presented mainly as proof that the techniques are feasible and reasonable to implement in a MPI implementation that supports fine grained locking. The results presented represent the mean of 10 runs.

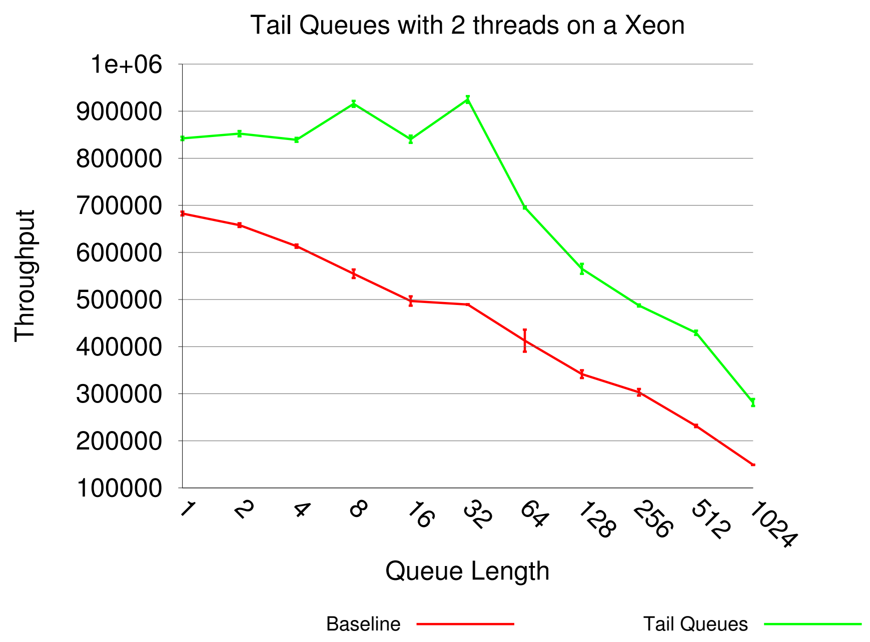

Figure 6.2: Tail Queues Prototype Using 2 Threads on a Xeon Sandy Bridge

Results

Figure 6.2 presents the results of running the shared memory test on a Sandy Bridge processor. The throughput shows a significant difference even at a small queue length. At a queue length of size 1, we see a 20% improvement. This is unexpected as the algorithm adds additional instructions to the code path and cannot do much work in parallel. However, Tail Queues results in reduced lock contention, as the contention has been spread out over many locks,which is a factor in improving performance. The difference peaks at 32 entries, where it gets close to a 95% improvement. This is where the threads are spending the majority of the benchmark working in parallel. Because both queues have 32 elements in them at this point, both sides of the data structure present an equal amount of work, allowing for near maximum parallel work.

Figures 6.3, 6.4, and 6.5 show the results of Tail Queues with different thread levels on the Xeon Phi Knight's Corner architecture. There are a several interesting trends here. First,

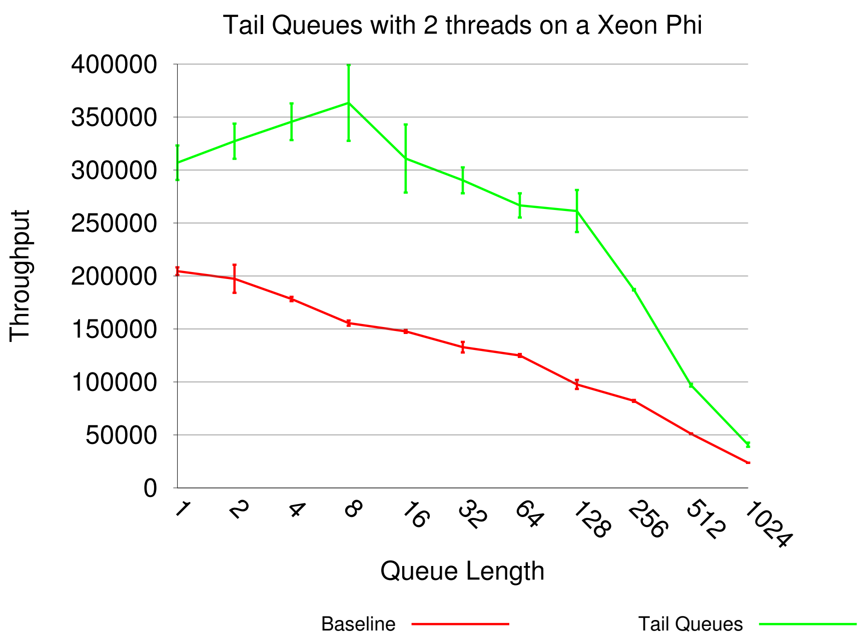

Figure 6.3: Tail Queues Prototype Using 2 Threads on a KNC

while the general trend lines are similar to the Xeon numbers, there are a couple of major differences: the maximum throughput is significantly lower, which is unsurprising as the cores are slower and less complex, and the maximum difference point has moved to be closer to 128 entries, where it appears to exceed 100% improvement, in some instances reaching close to 150% improvement. This further indicates that, in addition to increasing parallelism, we see an effect from reducing lock contention, like we observed for the Xeon processor results. Second is the general trend as thread counts increase. As this implementation only allows 2 threads to operate simultaneously, the lack of performance improvement as the number of threads increase is unsurprising. However, the decrease in throughput from 2 to 4 threads is surprising. This may be due to limited cache sharing on the Phis increasing the cost of obtaining the lock. As we increase from 4 to 8 threads, we observe the throughput growing to roughly the equivalent throughput of 2 threads.

Figure 6.6 shows the results of an experiment where the queue lengths only increase in

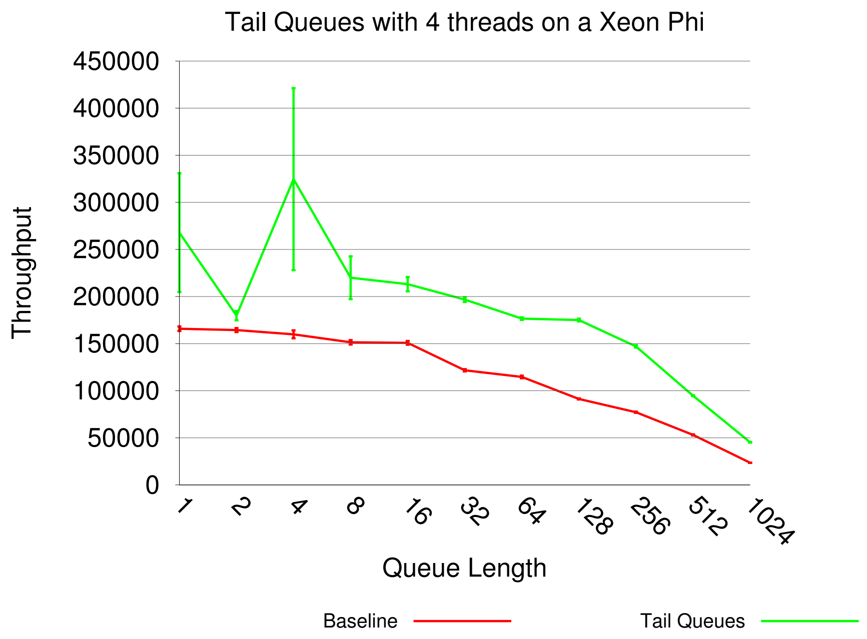

Figure 6.4: Tail Queues Prototype Using 4 Threads on a KNC

the PRQ. This is done to show that even with an imbalanced workload, Tail Queues can significantly increase performance. While the improvement is smaller than the other Xeon Phi experiments, the improvement is significant. While there is less work that can be done in parallel with two threads in this case, there is still an impact of the extra parallelism and reduced lock contention.

Figure 6.7 shows the preliminary results in an MPI implementation. These show roughly a 7.3% improvement to MPI message rate for Tail Queues plus binning over the baseline. At these queue lengths, MPI matching is significant enough of a bottle neck that we can see improvement on oversubscribed hardware. These techniques would likely show improvement with more threads as this experiment only uses 2 threads when the binning implementation supports 4 way thread parallelism.

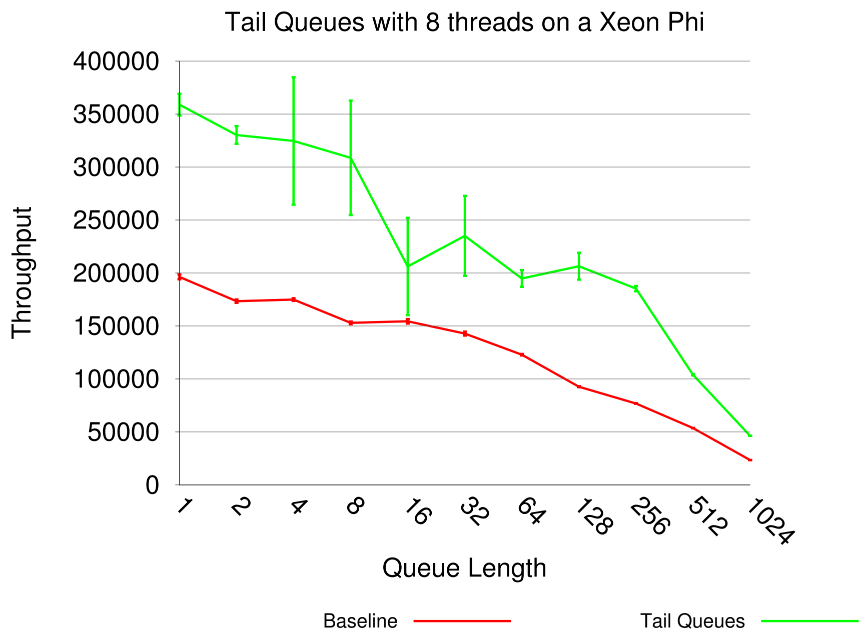

Figure 6.5: Tail Queues Prototype Using 8 Threads on a KNC

Discussion

These results show that Tail Queues can improve *MPI_THREAD_MULTIPLE* performance by reducing the matching critical section and allowing for independent parallelization of the PRQ and UMQ. While matching is currently a small part of applications, as fine grained messaging and multi-threading lead to a large increase in requirement for message rate per MPI process, Tail Queues can help alleviate performance problems associated with the MPI Matching critical section.

There are also a number of challenges for these techniques to be effective. First, this technique requires a fine-grained locking MPI implementation is required to be effective. Secondly, an implementation's performance depends on an efficient locking mechanism like MCS locks. Finally, a tuned linked list parallelization structure will allow for more performance as lists get long, which is non-trivial as the performance may depend on application behaviors such as average and distribution of matching search depth.

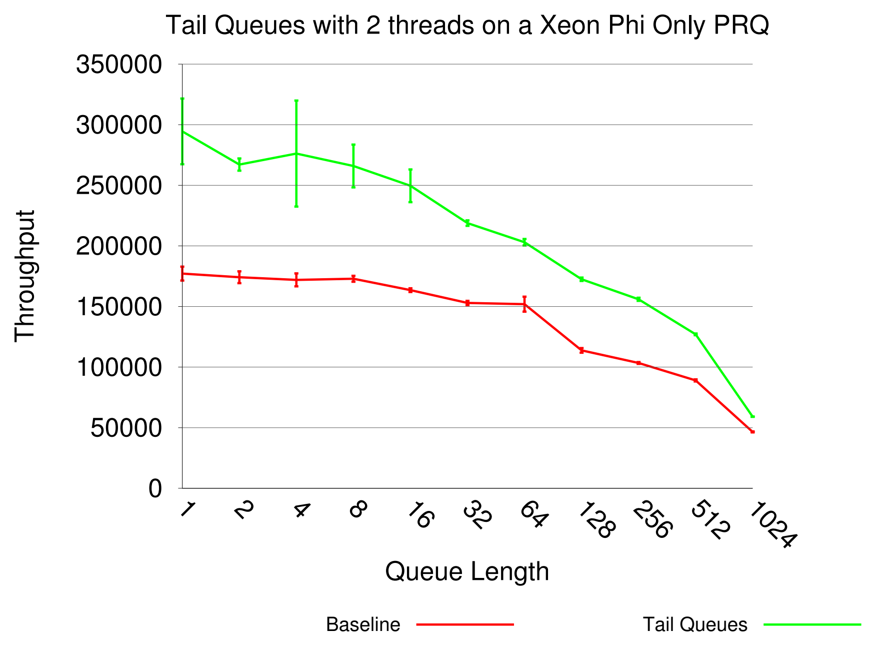

Figure 6.6: Tail Queues Prototype Using 2 Threads on a KNC - No UMQ Entries

Implications for Applications

The goal of Tail Queues is to enable applications to leverage $MPI\_THREAD\_MULTIPLE$. It is difficult to evaluate impact on applications due to the lack of applications that utilize

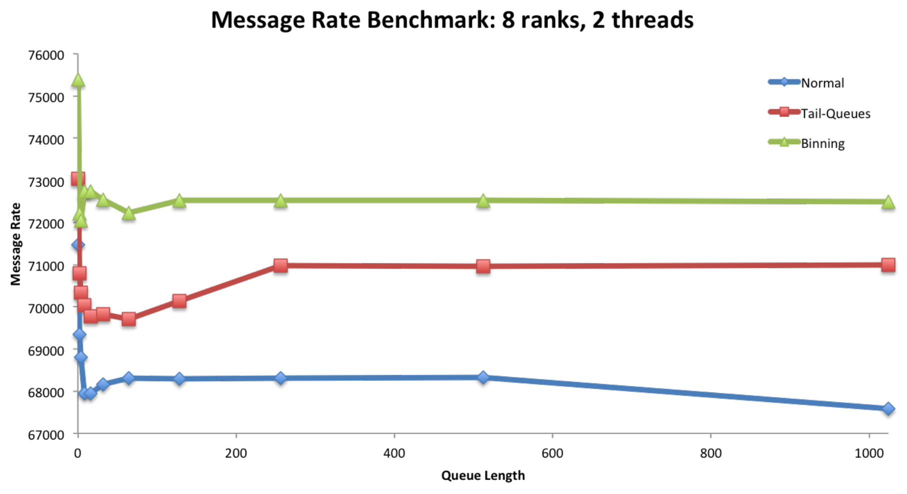

Figure 6.7: Tail Queues MPICH Implementation

$MPI\_THREAD\_MULTIPLE$. While $MPI\_THREAD\_MULTIPLE$ is desirable to support fine grained and parallel communication models, the lack of a performant implementation has prevented application developers from utilizing these code paths.

Current codes and show a wide range of queue lengths and search depths. The evaluation of Tail Queues is limited to list lengths of 1024 elements or less to represent the behavior of today's applications. However, future multithreaded codes that use MPI parallel access through endpoints [98] or $MPI\_THREAD\_MULTIPLE$, queue lengths and search depths are expected to increase in relation to the number of threads. To utilize all of the hardware threads on a Knight's Landing processor, one must utilize over 256 threads. Even if these threads are isolated into a processes per quadrant this could increase list lengths and search depths by 64x. This will make the matching engine even more of a bottleneck and will require leveraging the parallelism enabled by Tail Queues to remain performant.

6.4.2 Fuzzy Matching Results

Memory Savings in Open MPI

Open MPI's matching engine creates a 'proc' object for every other processor in the communicator. This object requires roughly 244 bytes. The per-peer matching structure contained within consists of two opal list objects, an posted receive queue and an unexpected message queue. An empty opal list structure requires 64 bytes of memory, 128 for both.

This becomes problematic with many-core systems, where we have many weaker cores. In many cases we have more processes and more processes per node. This requires more MPI contexts per node, and for each MPI context to maintain state on more peers. This can cause a large increase in the memory requirements per node to maintain MPI state. Some hybrid applications can minimize this, however many applications, including legacy applications, perform best in an MPI everywhere model. We also are observing tighter memory constraints as application developers are trying to fit their application entirely in high bandwidth memory to maximize performance.

The memory overhead for Open MPI is $128 * number\_of\_processes * processes\_per\_node$ bytes per full communicator per node. Given this we can calculate the memory overhead for different scenarios for just having the default communicator. Table 6.2 shows the memory overhead for Open MPI given four different scenarios; half scale and full scale MPI everywhere applications as they would appear on Trinity, and two forward looking scenarios with a doubling or quadrupling of the number of cores per node. Those last scenario could also be mapped to a scenario with mpi-endpoints, where each endpoint has its own matching engine and the application is running with an endpoint per hyper thread. As we can see in the table, the memory requirements of an MPI everywhere application grow as $O(processes\_per\_node^2)$ if the number of nodes is constant.

Knight's Landing - Small Scale Performance

To evaluate Fuzzy Matching at small scale, we used the same bandwidth microbenchmark as used in Chapter 5. As mentioned there, this bandwidth microbenchmark is based on the OSU microbenchmarks, with modifications to ensure preposting, clear the cache between iterations, and test different queue lengths. It is important to note that this is the worst case for the baseline, Open MPI, as all of the matching elements come from a single peer. This results in Open MPI's array of linked lists approach being the equivalent of a single linked list.

All of the micro-benchmark graphs show the average of 10 runs with error bars for standard deviation. These tests were run on a Cray XC 40 using KNL processors using a modified Open MPI. Each contain four different Open MPI optimization levels:

- Baseline, the default Open MPI implementation;
- Tradiitional, an MPICH style list implemented in Open MPI;

- Vectors, a vector matching algorithm as presented in Chapter 5 ; and,
- Fuzzy, a fuzzy matching implementation for the KNLs that uses 8 bits of tags and 24 bits of rank.

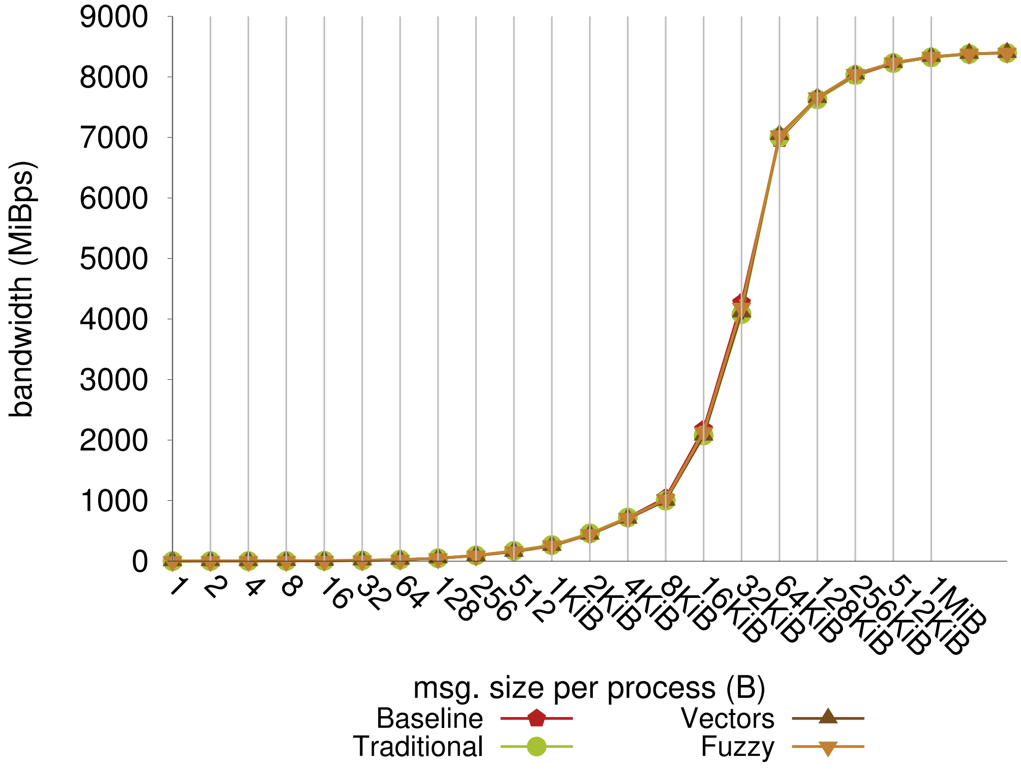

Figure 6.8: Bandwidth Performance With an Empty Matching Engine

The data collected in these tests are presented in four graphs. Figure 6.8 shows the bandwidth of the different approaches with an empty matching engine. Figure 6.9 shows the bandwidth of the different approaches when dealing with 1024 queue search depths. Figure 6.10 shows the bandwidth of small, 1 byte, messages for varying queue lengths. Figure 6.11 shows the bandwidth of medium, 4 kibibyte, messages for varying queue lengths. As expected, when the list is empty these approaches show no difference operating on an empty list, as the network is still a significant bottleneck. As the search depth grows, we observe a larger difference between the implementations. The Open MPI baseline is the worst due

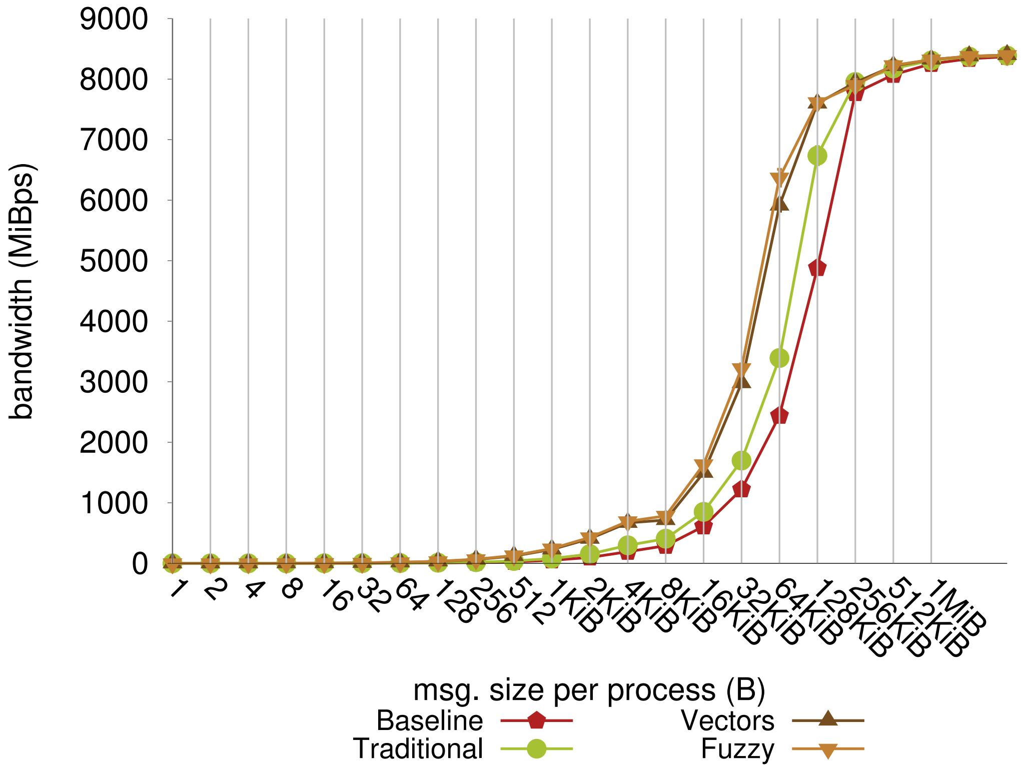

Figure 6.9: Bandwidth Performance at Queue Depth 1024

to overheads in dealing with the array of linked lists data structure and limited benefit from this structure under these conditions. Traditional performs a bit better by avoiding that overhead. Vectors performs significantly better which replicates results in Chapter 5. Finally, fuzzy matching consistently performs slightly better than vectors. This is due to the reduced number of AVX instructions by combining both the Tag mask and comparison and Rank mask and comparison while maintaining the same 16-element SIMD parallelism as the vectors approach.

Knight's Landing - Scaling Performance

To test our approach at scale, we leveraged the KNL partition of Trinity, a top 10 supercomputer. This allowed us to test up to 139,264 cores for four different applications. To

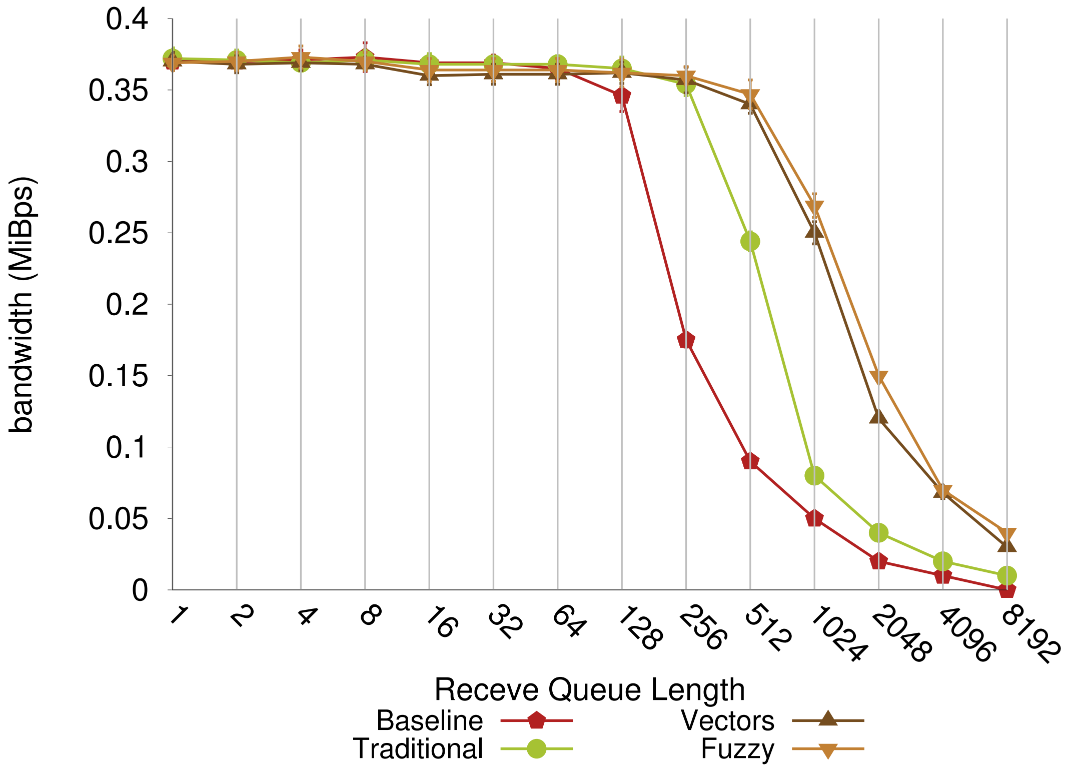

Figure 6.10: Bandwidth Performance at 1 byte Messages

test our performance impact at scale, we used four highly scalable codes from the exascale project's designated Mini Applications [1]. When available, we used the the best measured threading levels for each application [88]. The exception to this is Kripke; because this data wasn't available at the time we defaulted to a general default of 64 processes per node.

Each graph in this section presents a comparison between fuzzy matching and Open MPI's array of linked lists style matching engine and calculated memory savings. We ran each application 3 times per scale, displaying mean and standard deviation for all of the data points. All of the memory savings data is calculated based on Open MPI overhead generated in the same way as Section 6.4.2 and a Fuzzy matching overhead that is based on an assumption that maximum of concurrent match list entries for these applications is 8196 entries.

Figure 6.12 shows the results for MiniMD. MiniMD is a proxy application for the Lennard-

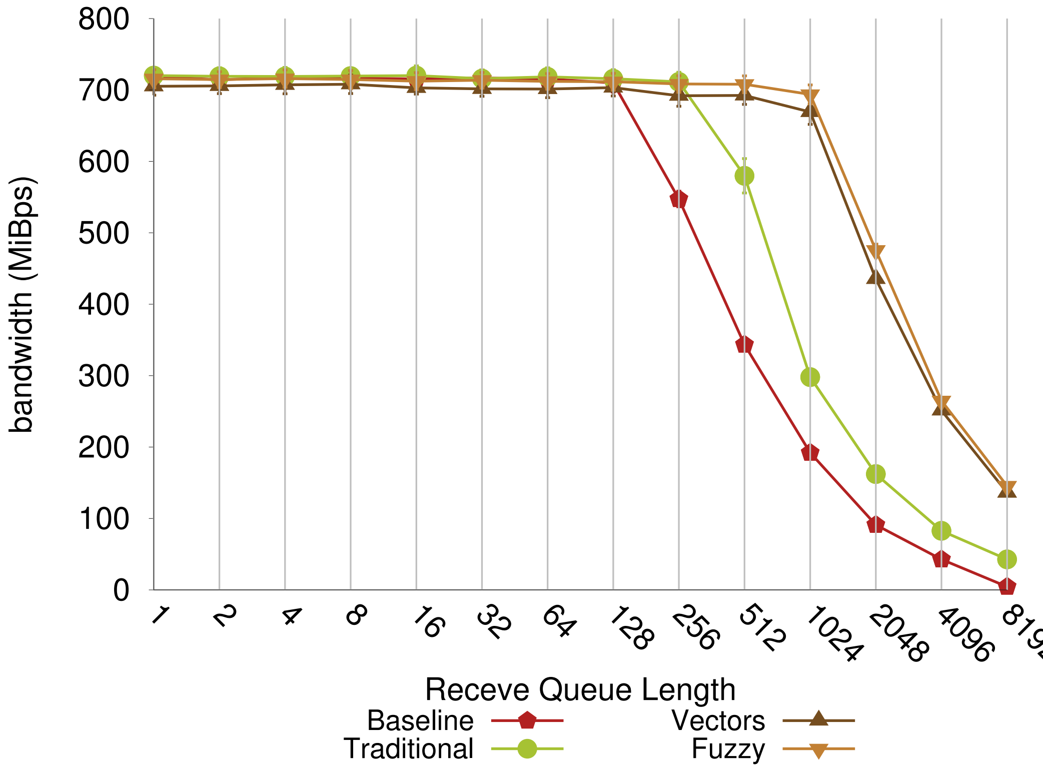

Figure 6.11: Bandwidth Performance at 4 KiB Messages

Jones molecular dynamics solver. This was run as an MPI everywhere application with a process per core (68 processes per node). Given the threading level it is unsurprising that the memory savings is up to 1.1 GiB at 2048 nodes. The performance does not show a significant difference, though standard deviation is high.

Figure 6.13 show the results for MiniFE. MiniFE is a proxy application for an unstructured finite element solver. The threading level for these runs was a hybrid MPI+OpenMP, with a single MPI process per node using a thread per hardware thread context (272 threads per node). Given the threading level for MiniFE it is unsurprising that the memory savings are minimal as there are only 4096 MPI ranks at the largest scale. Like MiniMD, the performance difference in negligible.

Figure 6.14 shows the results for LULESH. LULESH is a proxy application for Lagrangian

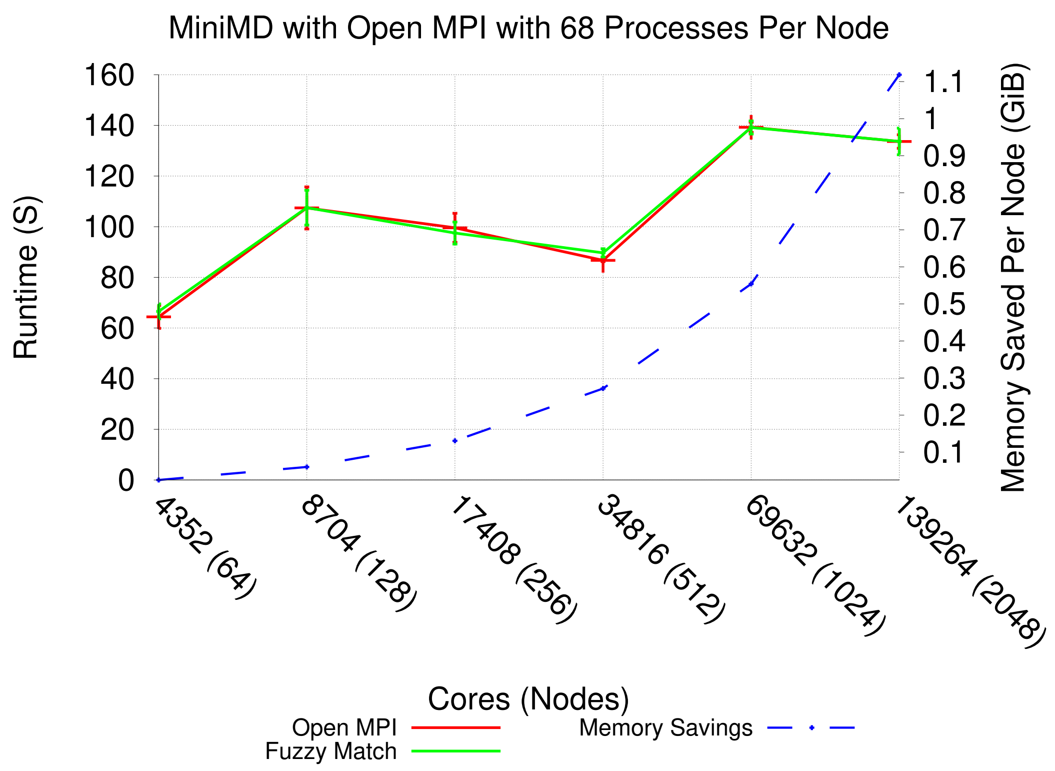

Figure 6.12: MiniMD at Scale

explicit shock hydrodynamics codes. The threading level for these runs was a hybrid MPI+OpenMP, with 8 MPI processes per node using 8 threads per process. As only a limited number of MPI Processes share each node, the threading level for LULESH limits the amount of memory saving to 0.01 GiB. Like MiniMD and MiniFE, the performance difference in negligible with one anomaly at 1024 nodes. This anomaly is likely due to the unstable nature of Trinity during the open science period in which these tests were run.

Figure 6.15 shows the results for Kripke. Kripke is a proxy application for 3D Sn deterministic particle transport, specifically the ARDRA application. The threading level for these runs was MPI everywhere with highest power of 2 MPI processes per node available (64 processes per node). Since the threading level is similar to MiniMD, it is unsurprising that the memory savings approaches 1GiB on 2048 nodes. Unlike the other miniapps, the per-

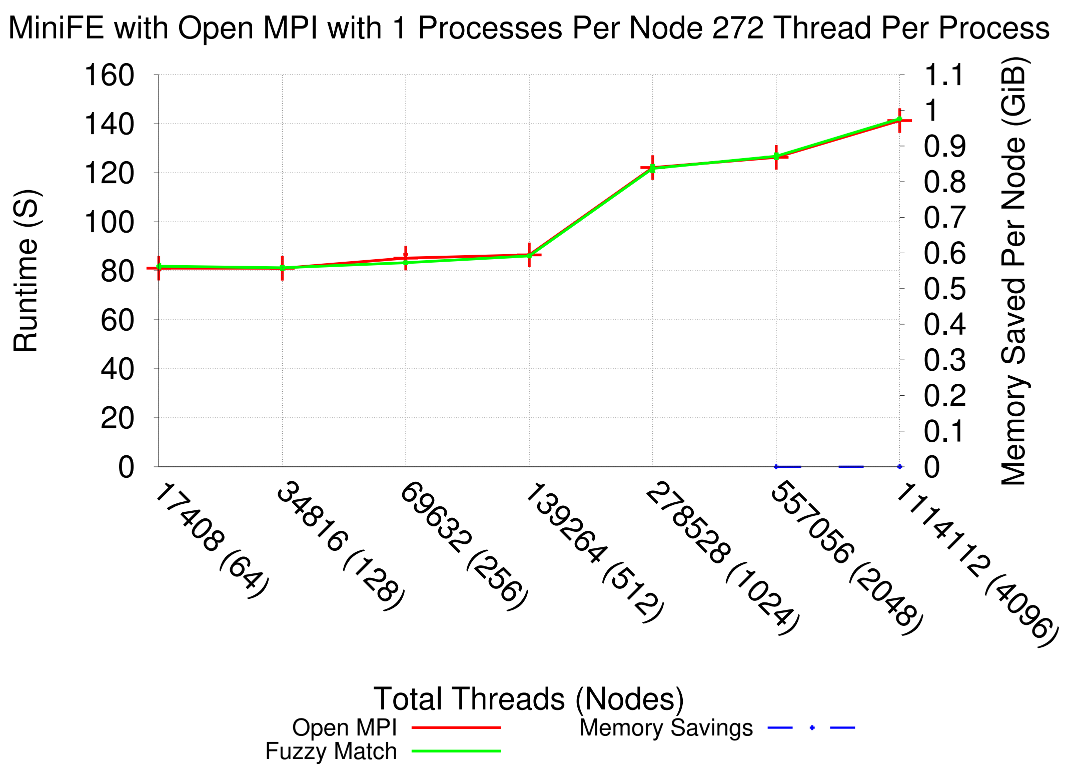

Figure 6.13: MiniFE at Scale

formance difference was larger, with Fuzzy Matching reducing the runtime by up to 6.12% for some scales. While 5 out of the 6 scales tested showed speedup with Fuzzy Matching the high standard deviation makes it difficult to drawn any significant conclusions.

Comparing the memory requirements of the Open MPI's matching engine and the fuzzy matching approach, it becomes clear that there is significant benefit to using fuzzy matching for MPI Everywhere applications for modern extreme scale systems. Given the applications results in this section, it is clear that the approach does not negatively impact highly scalable codes, even when their run a hybrid MPI+X mode.

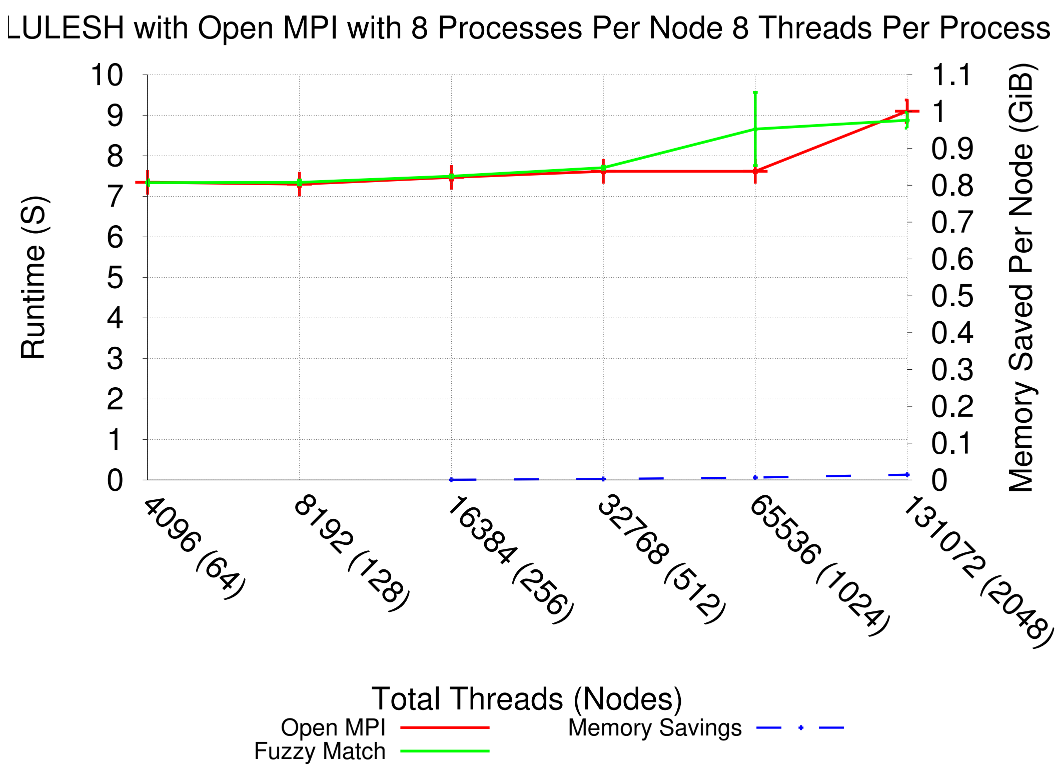

Figure 6.14: LULESH at Scale

Discussion

As shown in both the memory analysis and the large scale application evaluation, fuzzy matching can improve the memory use of the Open MPI matching engine on the KNL architecture. This memory savings can be critical for architectures like the KNL where application developers often try to limit the applications memory usage to the small performant MCDRAM. Looking at the microbenchmarks, fuzzy matching performance over the Cray Aries network does not is not degraded by matching until the search operations reach a depth 1024 for small messages and 2048 for medium size messages. This is sufficient to accommodate most of the scaleable communication patterns exascale applications are expected to use. This is illustrated by proxy applications we tested at scale which did not experience any degradation. The application tag and rank usage analysis showed potential for fuzzy

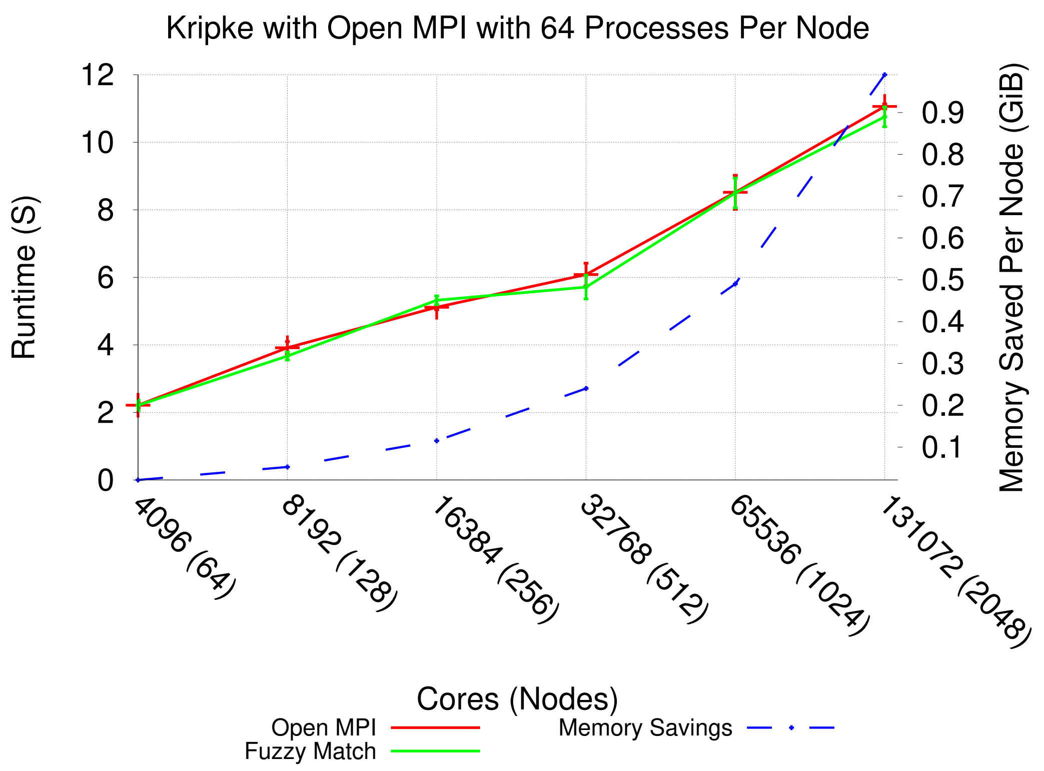

Figure 6.15: Kripke at Scale

matching to be performant on a number of applications.

The missing piece from evaluating fuzzy matching as a technique is the lack of tests on an architecture that supports different vector subdivisions allowing for more elements at the cost of smaller element sizes. This is needed to explore the performance difference between a standard vector matching technique and the fuzzy matching technique. On the KNL, these approaches are limited to the same level of SIMD parallelism and thus it's unsurprising that these results show them as similar. At the time of this dissertation, we are in the process of getting access to a system that supports other SIMD parallelism granularities, which will allow us to explore trade-offs between different fuzzy matching bit windows and false positive rates.

```
Data: Request with Tag, Source, and Context ID
Result: Matched Element (null if not found)
lock UMQL;
foreach element in UMQ do
    if element matches Tag, Source, and Context ID then
        remove element from UMQ;
        unlock UMQL;
        return element ;
    end
end
lock TL;
foreach element in UMQI do
    if element matches Tag, Source, and Context ID then
        remove element from UMQI;
        append UMQI to UMQ;
        UMQI = empty list;
        unlock TL;
        unlock UMQL;
        return element ;
    end
    remove element from UMQI;
    append element to UMQ;
end
append new element to PRQI;
unlock TL;
unlock UMQ;
return null ;
```

Algorithm 2: Tail Queues - Post Receive

Half Scale	4096 Nodes, 68 Ranks Per	2.26 GiB
Full Scale	8192 Nodes, 68 Ranks Per	4.52 GiB
2x Cores	8192 Nodes, 136 Ranks Per	18.06 GiB
4x Cores	8192 Nodes, 272 Ranks Per	72.25 GiB

Table 6.2: Open MPI Memory Overhead for different Run Scenarios for Trinity's KNL Partition

Chapter 7

Conclusions and Future Work

The goal of this dissertation was to quantify the impact of many-core systems on HPC communication libraries and explore methods of mitigating these issues for both legacy and future applications. In this chapter, I present a summary of my contributions and explore future work.

7.1 Summary

In this dissertation, I examined the performance impact of many-core processors on HPC communication library performance and strategies to mitigate that impact. I demonstrated this impact through empirical studies on a range of current generation hardware set-ups. To mitigate this effect, I explored new parallel programing models for future applications and a number of HPC communication library optimizations targeted at supporting legacy applications. Below I discus the contributions of this work.

An evaluation of the impact of many-core processors on modern communication systems including onload and offload network architectures In Chapter 3, I explored the impact of many-core processor on the power and performance of HPC communication libraries on the two major interconnect hardware approaches. Using DVFS, I was able to explore the effects of the CPU speed on HPC communication library processing, with speed ranging from many-core equivalents (1.4 GHz) to the fast large core processors (3.8 GHz). By leveraging power insight measurement devices, I was able to examine the trade off spaces with hardware optimizations for network processing and the effect on high scale parallel applications.

A suite of benchmarks, RMA-MT, and studies using them to explore one-sided programing models In Chapter 4, I explored one-sided multi-threaded programing models through two suites of benchmarks, RMA-MT and SHMEM-MT. One sided communication, like MPI's RMA or OpenSHMEM, avoid the majority of the computational steps required by shifting memory management from the receiver to the sender. For this reason, this programing model has been proposed as way to achieve HPC communication library performance at Exascale. The RMA-MT benchmarks exposed both performance and functionality issues with current support for these models. The SHMEM-MT benchmarks showed the performance benefit of one-sided communication by examining a simple one-sided focused HPC communication library.

A modern matching architecture that leverages vector operations and caching behavior In Chapter 5, I presented three approaches for improving two-sided MPI performance on KNL processors. The three optimizations were: packing multiple elements into a single cache line, vector matching, and explicit cache management through cache heating. I then presented a study which throughly examined performance of these three approaches, separately and in combination, on a variety of architectures. The combination of these approaches improved performance on all of the systems tested including KNL performance that saw up to a 40x increases to MPI bandwidth.

Matching approaches that improve matching performance for future programing models and hardware In Chapter 6, I explored two new matching approaches that should improve performance in future systems. Tail Queues is an approach that seeks to parallelize the matching engine, by isolating the critical section at the end of the lists. This allows each list to be parallelized independently, which improves thread multiple performance. This is targeted at future programing models, where MPI+X and fine grained messaging require *MPI_THREAD_MULTIPLE*. The other approach is fuzzy matching. This is an extension of vector matching utilizing new vector instructions found in the AVX-512 Bytes and Words module. This module provides instructions for increasing the number of integers processes in a single vector operation by reducing the size of each. By optimistically matching against a bit window, MPI can utilize these vector instructions to increase the amount of parallelism. This speeds up matching when applied to a traditional matching engine and can significantly reduce the memory overhead of Open MPI style approaches by allowing the implementation to use one list without a significant performance penalty for most applications.

7.2 Future Work

There are a number of different future directions that directly extend from the work presented in this dissertation. These include characterizing power and performance trade-offs for different offload hardware, developing new benchmarks that explore the performance of mutil-threaded one-sided communication using algorithms that leverage fine grained messaging designs, and leveraging new hardware to increase the efficiency of vector based matching. There are also several longer term directions this research can lead. These include an evaluation an exploration of hot caching for both MPI communication libraries and applications, and designing new hardware that leverages vector matching.

With the growth in the number of different offload network architectures available, the trade-off space for these is becoming more complex. Characterizing power and performance trade-offs for different offload hardware would increase our understanding of the power and performance trade-offs of simplistic offload architectures and those that are more complex. This presents a number of challenges to make ensure fair comparisons across vendors, as isolating the impact of the network card becomes more difficult when the other components are varied.

While RMA-MT and SHMEM-MT are good first steps at exploring one-sided multi-threaded performance, they focus on pushing the implementation rather than characterizing the programing models these are designed for. Developing new application-level benchmarks that explore the performance of mutil-threaded one-sided communication when using algorithms that leverage fine grained messaging designs provides a better understanding these communication patters. This can be in the form of increasing performance; one-sided communication should consolidate the amount of synchronization needed to support fine-grained communication algorithms. However, it is also valuable to explore the challenges and restrictions of writing and adapting codes to the programming models. Developing this with a modular communication architecture to additionally support two sided communication could also help us explore the application impact of novel advances in $MPI\_THREAD\_MULTIPLE$ codepaths, such as Tail Queues.

There are a number of new iteration of vector units in the various CPU architecture that are interesting to leverage for matching. Intel's Sky Lake expands support for AVX 512, including different sizes of integers. As mentioned in Chapter 6 this would allow us to further explore vector matching architectures like fuzzy matching. There is also ARM's SVE which supports a modular vector unit which can support up to 2KiB vectors. This could allow for the exploration of the effects of vector size, which would require a characterization of how applications order their matching elements to determine how many holes each long vector will have and what size of vector is most performant for each application.

The exploration of Hot Caching is very preliminary at this point. I have demonstrated the technique and its potential use in MPI, however there is still missing information needed to fully explore this concept. First, we need to explore how viable it is to have a Hot Cache that can be toggled. While the positive impact of Hot Caching is verified, it is useful to have a Hot Cache that can accelerate serial sections of code and without causing interference or requiring a dedicated core during parallel sections of code. There is also potential to allow for user specified regions, which would allow for an exploration of the trade-off of hot caching performance critical data structures in applications against the resources used and resource contention caused by the technique.

Finally, there is potential to leverage the vector matching technique in offload cards. By utilizing an FPGA for matching, one could theoretically design many new features in a wide vector unit. This would allow for utility functions like delete and rotate, where an element is deleted and the following elements are shifted back one place. This would reduce the number of holes in the data structure and could be used to ensure the vectors are packed properly reducing the needed number of vector compares.

References

[1] Exascale project - proxy applications. *URL: https://exascaleproject.github.io/proxy-apps/all-apps/*, 2017.

[2] Robert Alverson, Duncan Roweth, and Larry Kaplan. The Gemini system interconnect. In *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, pages 83–87, Aug 2010.

[3] Abdelhalim Amer, Huiwei Lu, Yanjie Wei, Pavan Balaji, and Satoshi Matsuoka. MPI+ threads: Runtime contention and remedies. *ACM SIGPLAN Notices*, 50(8):239–248, 2015.

[4] Katie Antypas. Nersc-6 workload analysis and benchmark selection process. *Lawrence Berkeley National Laboratory*, 2008.

[5] Baba Arimilli, Ravi Arimilli, Vicente Chung, Scott Clark, Wolfgang Denzel, Ben Drerup, Torsten Hoefler, Jody Joyner, Jerry Lewis, Jian Li, Nan Ni, and Ram Rajamony. The PERCS high-performance interconnect. In *IEEE Symposium on High-Performance Interconnects*, August 2010.

[6] Rob T. Aulwes, David J. Daniel, Nehal N. Desai, Richard L. Graham, L. Dean Risinger, Mark A. Taylor, Timothy S. Woodall, and Mitchel W. Sukalski. Architecture of LA-MPI, a network-fault-tolerant MPI. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 15. IEEE, 2004.

[7] Allison H. Baker, Robert D. Falgout, Tzanio V. Kolev, and Ulrike Meier Yang. Multigrid smoothers for ultraparallel computing. *SIAM Journal on Scientific Computing*, 33(5):2864–2887, 2011.

[8] Pavan Balaji, Darius Buntinas, David Goodell, William D. Gropp, and Rajeev Thakur. Fine-grained multithreading support for hybrid threaded MPI programming. *International Journal of High Performance Computing Applications*, 24(1):49–57, 2010.

[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. *The International Journal of High Performance Computing Applications*, 28(4):415–424, 2014.

[10] Brian W. Barrett, Ron Brightwell, Ryan E. Grant, K. Scott Scott Hemmert, Kevin T. Pedretti, Kyle Wheeler, Keith D. Underwood, Rolf Riesen, Arthur B. Maccabe, and Trammell Hudson. The Portals 4.0.2 networking programming interface, 2014.

[11] 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*, EuroMPI '13, pages 67–71, New York, NY, USA, 2013. ACM.

[12] Richard F. Barrett, Dylan T. Stark, Courtenay T. Vaughan, Ryan E. Grant, Stephen L. Olivier, and Kevin T. Pedretti. Toward an evolutionary task parallel integrated MPI+X programming model. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 30–39. ACM, 2015.

[13] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*, page 66. IEEE Computer Society Press, 2012.

[14] Mohammadreza Bayatpour, Hari Subramoni, Sourav Chakraborty, and Dhabaleswar K. Panda. Adaptive and dynamic design for MPI tag matching. In *Cluster Computing (CLUSTER), 2016 IEEE International Conference on*, pages 1–10. IEEE, 2016.

[15] Mark S. Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovettt, Todd Rimmer, Keith D. Underwood, and Robert C. Zak. Intel Omni-Path architecture enabling scalable, high performance fabrics. *Annual Symposium on High-Performance Interconnects (HOTI)*, pages 1–9, 2015.

[16] Ron Brightwell, Sue P. Goudy, Arun F. Rodrigues, and Keith D. Underwood. Implications of application usage characteristics for collective communication offload. *International Journal of High Performance Computing and Networking*, 4(3):104–116, 2006.

[17] Ron Brightwell, Sue P. Goudy, and Keith D. Underwood. A preliminary analysis of the MPI queue characteristics of several applications. 2005.

[18] Ron Brightwell, Kevin T. Pedretti, and Kurt B. Ferreira. Instrumentation and analysis of MPI queue times on the SeaStar high-performance network. *Proceedings of the International Conference on Computer Communications and Networks (ICCCN)*, pages 590–596, 2008.

[19] Ron Brightwell and Keith D. Underwood. An analysis of NIC resource usage for offloading MPI. In *18th International Parallel and Distributed Processing Symposium (IPDPS)*, page 183. IEEE, 2004.

[20] Ron Brightwell, Keith D. Underwood, and Rolf Riesen. An initial analysis of the impact of overlap and independent progress for MPI. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 370–377. Springer, 2004.

[21] John Byrne, Jichuan Chang, Kevin T. Lim, Laura Ramirez, and Parthasarathy Ranganathan. Power-efficient networking for balanced system designs: Early experiences with PCIe. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, HotPower '11, pages 3:1–3:5, New York, NY, USA, 2011. ACM.

[22] Barbara M. Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeffery Kuehn, Chuck Koelbel, and Lauren Smith. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, page 2. ACM, 2010.

[23] Fabio Checconi and Fabrizio Petrini. Massive data analytics: the graph 500 on IBM Blue Gene/Q. *IBM Journal of Research and Development*, 57(1/2):10–1, 2013.

[24] Dong Chen, Noel A. Eisley, Philip Heidelberger, Robert M. Senger, Yutaka Sugawara, Sameer Kumar, Valentina Salapura, David L. Satterfield, Burkhard Steinmacher-Burow, and Jeffrey J. Parker. The IBM Blue Gene/Q interconnection network and message unit. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 26:1–26:10, New York, NY, USA, 2011. ACM.

[25] Aswini Chowdappa and Anthony Skjellum. Thread-safe message passing with p4 and MPI. Technical report.

[26] MILC Collaboration et al. Mimd lattice computation (MILC) collaboration home page. *Information available at http://physics.indiana.edu/sg/milc.html.*

[27] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[28] Said Derradji, Thibaut Palfer-Sollier, Jean-Pierre Panziera, Axel Poudes, and Francois Atos Wellenreiter. The BXI interconnect architecture. *In Proceedings of the 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects, HOTI*, pages 18–25, 2015.

[29] Rossen Dimitrov. ChaMPIon/Pro-the complete MPI-2 for massively parallel linux. In *Linux clusters: the HPC revolution*, 2004.

[30] Rossen Dimitrov and Anthony Skjellum. Software architecture and performance comparison of MPI/Pro and MPICH. In *International Conference on Computational Science*, pages 307–315. Springer, 2003.

[31] 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.

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

[33] 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 *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 342–350. IEEE, 2015.

[34] Matthew G. F. Dosanjh, Taylor Groves, Ryan E. Grant, Patrick G. 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.

[35] Thomas H. Dunigan, Jeffrey S. Vetter, and Patrick H. Worley. Performance evaluation of the SGI Altix 3700. In *2005 International Conference on Parallel Processing (ICPP'05)*, pages 231–240. IEEE, 2005.

[36] Greg Faanes, Abdulla Bataineh, Duncan Roweth, Edwin Froese, Bob Alverson, Tim Johnson, Joe Kopnick, Mike Higgins, James Reinhard, et al. Cray Cascade: A scalable HPC system based on a Dragonfly network. In *Proceedings of the ACM/IEEE International Conference on High-Performance Computing, Networking, Storage, and Analysis (SC'12)*, November 2012.

[37] Graham Fagg and Jack Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. *Recent advances in parallel virtual machine and message passing interface*, pages 346–353, 2000.

[38] Karl Feind. Shared memory access (SHMEM) routines. In *Proceedings of the Cray User Group Conference*, pages 203–208, 1995.

[39] Kurt B. Ferreira, Scott Levy, Kevin T. Pedretti, and Ryan E. Grant. Characterizing MPI matching via trace-based simulation. In *Proceedings of the 24th European MPI Users' Group Meeting*, page 8. ACM, 2017.

[40] Mario Flajslik, James Dinan, and Keith D. Underwood. Mitigating MPI message matching misery. In *International Conference on High Performance Computing*, pages 281–299. Springer, 2016.

[41] Edgar Gabriel, Michael Resch, Thomas Beisel, and Rainer Keller. Distributed computing in a heterogeneous computing environment. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 180–187, 1998.

[42] Richard L. Graham, Steve Poole, Pavel Shamis, Gil Bloch, Noam Bloch, Hillel Chapman, Michael Kagan, Ariel Shahar, Ishai Rabinovitz, and Gilad Shainer. ConnectX-2 InfiniBand management queues: First investigation of the new support for network offloaded collective operations. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 53–62, May 2010.

[43] Ryan E. Grant, James H. Laros III, Michael Levenhagen, Stephen L. Olivier, Kevin T. Pedretti, Lee Ward, and Andrew J. Younge. Evaluating energy and power profiling techniques for HPC workloads.

[44] Ryan E. Grant, Michael Levenhagen, Stephen L. Olivier, David DeBonis, Kevin T. Pedretti, and James H. Laros III. Standardizing power monitoring and control at exascale. *Computer*, 49(10):38–46, 2016.

[45] Ryan E. Grant, Mohammad J. Rashti, Ahmad Afsahi, and Pavan Balaji. RDMA capable iWARP over datagrams. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 628–639. IEEE, 2011.

[46] Ryan E. Grant, Anthony Skjellum, and Purushotham V. Bangalore. Lightweight threading with MPI using persistent communications semantics. Technical report, Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), 2015.

[47] William D. Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[48] William D. Gropp and Rajeev Thakur. Thread-safety in an MPI implementation: Requirements and analysis. *Parallel Computing*, 33(9):595–604, 2007.

[49] Taylor Groves, Ryan E. Grant, and Dorian C. Arnold. NiMC: Characterizing and eliminating network-induced memory contention. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 253–262. IEEE, 2016.

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

[52] Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thorn-

quist, and Robert W. Numrich. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep*, 2009.

[53] Nathan T. Hjelm, Samuel K. Gutierrez, and Manjunath Gorentla Venkata. On the current state of Open MPI on Cray systems, 2014.

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

[55] Torsten Hoefler, Greg Bronevetsky, Brian W. Barrett, Bronis R. De Supinski, and Andrew Lumsdaine. Efficient MPI support for advanced hybrid programming models. In *Recent Advances in the Message Passing Interface*, pages 50–61. Springer, 2010.

[56] Wei Huang, Gopalakrishnan Santhanaraman, H-W Jin, Qi Gao, and Dhabaleswar K. Panda. Design of high performance MVAPICH2: MPI2 over InfiniBand. In *Cluster Computing and the Grid, 2006 (CCGRID 06). Sixth IEEE International Symposium on*, volume 1, pages 43–48. IEEE, 2006.

[57] 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.

[58] Grigori Inozemtsev and Ahmad Afsahi. Designing an offloaded nonblocking MPI_Allgather collective using CORE-direct. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 477–485. IEEE, 2012.

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

[60] Jithin Jose, Sreeram Potluri, Karen Tomko, and Dhabaleswar K. Panda. Designing scalable graph500 benchmark with hybrid MPI+OpenSHMEM programming models. In *Supercomputing*, pages 109–124. Springer, 2013.

[61] Jithin Jose, Jie Zhang, Akshay Venkatesh, Sreeram Potluri, and Dhabaleswar K. Panda. A comprehensive performance evaluation of OpenSHMEM libraries on InfiniBand clusters. In *Openshmem and Related Technologies. Experiences, Implementations, and Tools*, pages 14–28. Springer, 2014.

[62] Laxmikant V. Kale and Sanjeev Krishnan. CHARM++: a portable concurrent object oriented system based on C++. In *ACM Sigplan Notices*, volume 28, pages 91–108. ACM, 1993.

[63] Humaira Kamal and Alan Wagner. FG-MPI: Fine-grain MPI for multicore and clusters. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.

[64] Arkady Kanevsky, Anthony Skjellum, and Anna Rounbehler. MPI/RT - an emerging standard for high-performance real-time systems. In *System Sciences, 1998., Proceedings of the Thirty-First Hawaii International Conference on*, volume 3, pages 157–166. IEEE, 1998.

[65] Ian Karlin, Abhinav Bhatele, Bradford L. Chamberlain, Jonathab Cohen, Zachary Devito, Maya Gokhale, Riyaz Haque, Rich Hornung, Jeff Keasler, Dan Laney, et al. LULESH programming model and performance ports overview. *Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-608824*, 2012.

[66] Ian Karlin, Jeff Keasler, and Rob Neely. LULESH 2.0 updates and changes. Technical Report LLNL-TR-641973, August 2013.

[67] Rainer Keller and Richard L. Graham. Characteristics of the unexpected message queue of MPI applications. In *European MPI Users' Group Meeting*, pages 179–188. Springer, 2010.

[68] Benjamin Klenk and Holger Fröning. An overview of MPI characteristics of exascale proxy applications. In *International Supercomputing Conference*, pages 217–236. Springer, 2017.

[69] Benjamin Klenk, Holger Fröning, Hans Eberle, and Larry Dennison. Relaxations for high-performance message passing on massively parallel SIMT processors. In *31st International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017.

[70] James H. Laros III, Ryan E. Grant, Michael Levenhagen, Stephen L. Olivier, Kevin T. Pedretti, Lee Ward, and Andrew J. Younge. High performance computing-power application programming interface specification version 2.0.

[71] James H. Laros III, Phil Pokorny, and David DeBonis. PowerInsight - A commodity power measurement capability. In *The Third International Workshop on Power Measurement and Profiling in conjunction with IEEE IGCC 2013*, Arlington Va, 2013.

[72] Edgar A. León, Ian Karlin, and Ryan E. Grant. Optimizing explicit hydrodynamics for power, energy, and performance. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 11–21. IEEE, 2015.

[73] Edgar A. León, Ian Karlin, Ryan E. Grant, and Matthew G. F. Dosanjh. Program optimizations: The interplay between power, performance, and energy. *Parallel Computing*, 58:56–75, 2016.

[74] Mingzhe Li, Jian Lin, Xiaoyi Lu, Khaled Hamidouche, Karen Tomko, and Dhabaleswar K. Panda. Scalable MiniMD design with hybrid MPI and OpenSHMEM. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, page 24. ACM, 2014.

[75] Guangdeng Liao, Xia Zhu, Steen Larsen, Laxmi Bhuyan, and Ram Huggahalli. Understanding power efficiency of TCP/IP packet processing over 10GbE. In *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, pages 32–39, Aug 2010.

[76] Glenn R. Luecke, Bruno Raffin, and James J. Coyle. Comparing the communication performance and scalability of a SGI Origin 2000, a cluster of Origin 2000's and a Cray T3E-1200 using SHMEM and MPI routines. In *THE JOURNAL OF PEMCS, IA*. Citeseer, 1999.

[77] Glenn R. Luecke, Silvia Spanoyannis, and Marina Kraeva. The performance and scalability of SHMEM and MPI-2 one-sided routines on a SGI Origin 2000 and a Cray T3E-600. *Concurrency and Computation: Practice and Experience*, 16(10):1037–1060, 2004.

[78] Chris Maynard. Comparing one-sided communication with MPI, UPC and SHMEM. *Proceedings of the Cray User Group (CUG)*, 2012, 2012.

[79] Kevin McGrattan, Simo Hostikka, Randall McDermott, Jason Floyd, Craig Weinschenk, and Kristopher Overholt. Fire dynamics simulator, user's guide. *NIST special publication*, 1019:6th Edition, 2013.

[80] John M. Mellor-Crummey and Michael L. Scott. Synchronization without contention. *ACM SIGPLAN Notices*, 26(4):269–278, 1991.

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

[82] Bradford Nichols, Dick Buttlar, and Jacqueline Farrell. *Pthreads programming: A POSIX standard for better multiprocessing.* " O'Reilly Media, Inc.", 1996.

[83] Jarek Nieplocha, Vinod Tipparaju, Manojkumar Krishnan, and Dhabaleswar K. Panda. High performance remote memory access communication: The ARMCI approach. *International Journal of High Performance Computing Applications*, 20:233–253, 2006.

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

[85] Swaroop Pophale, Ramachandra Nanjegowda, Tony Curtis, Barbara Chapman, Haoqiang Jin, Stephen Poole, and Jeffery Kuehn. OpenSHMEM performance and potential: A NPB experimental study. In *The 6th Conference on Partitioned Global Address Space Programming Models, PGAS*. Citeseer, 2012.

[86] Portals Development Team. Portals 4.0 reference implementation. available at: http://code.google.com/p/portals4/ (Mar. 2013).

[87] Howard Pritchard, Duncan Roweth, David Henseler, and Paul Cassella. Leveraging the Cray Linux Environment Core Specialization feature to realize MPI asynchronous progress on Cray XE systems. In *Proceedings of the Cray User Group Conference*, May 2012.

[88] Mahesh Rajan, Douglas W. Doerfler, and Simon D. Hammond. Trinity benchmarks on Intel Xeon Phi. Technical report, Sandia National Laboratories, 2015.

[89] Mohammad J. Rashti, Ryan E. Grant, Ahmad Afsahi, and Pavan Balaji. iWARP redefined: Scalable connectionless communication over high-speed ethernet. In *High Performance Computing (HiPC), 2010 International Conference on*, pages 1–10. IEEE, 2010.

[90] Arun F. Rodrigues, K. Scott Hemmert, Brian W. Barrett, Chad D. Kersey, Ron Oldfield, Marlo Weston, Rolf Riesen, Jeanine Cook, Paul Rosenfeld, Elliot Cooper-Balis, et al. The structural simulation toolkit. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):37–42, 2011.

[91] Arun F. Rodrigues, Richard Murphy, Ron Brightwell, and Keith D. Underwood. Enhancing NIC performance for MPI using processing-in-memory. In *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 8–pp. IEEE, 2005.

[92] Jose Carlos Sancho, Darren J. Kerbyson, and Kevin J. Barker. Efficient offloading of collective communications in large-scale systems. In *Cluster Computing, 2007 IEEE International Conference on*, pages 169–178. IEEE, 2007.

[93] Timo Schneider, Torsten Hoefler, Ryan E. Grant, Brian W. Barrett, and Ron Brightwell. Protocols for fully offloaded collective operations on accelerated network adapters. In *42nd International Conference on Parallel Processing (ICPP'13)*, Lyon, France, October 2013.

[94] Whit Schonbein. Personal communications.

[95] Leah Shalev, Julian Satran, Eran Borovik, and Muli Ben-Yehuda. IsoStack: Highly efficient network processing on dedicated cores. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.

[96] Quinn O. Snell, Armin R. Mikler, and John L. Gustafson. NETPIPE: A network protocol independent performance evaluator. 6, 1996.

[97] Avinash Sodani. Knights landing (knl): 2nd generation Intel® Xeon Phi processor. In *Hot Chips 27 Symposium (HCS), 2015 IEEE*, pages 1–24. IEEE, 2015.

[98] 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.

[99] Dylan T. Stark, Richard F. Barrett, Ryan E. Grant, Stephen L. Olivier, Kevin T. Pedretti, and Courtenay T. Vaughan. Early experiences co-scheduling work and communication tasks for hybrid MPI+ X applications. In *Proceedings of the 2014 Workshop on Exascale MPI*, pages 9–19. IEEE Press, 2014.

[100] Nikolaos Tampouratzis, Pavlos M. Mattheakis, and Ioannis Papaefstathiou. Accelerating intercommunication in highly parallel systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(4):40, 2016.

[101] MPICH Development Team. CH4, 2016. Last accessed: 1/4/2017.

[102] MPICH Development Team. MPICH, 2017. Last accessed: 30/3/2017.

[103] MVAPICH Team. MVAPICH2 2.0 user guide. 2001.

[104] OPENMPI Development Team. Open MPI, 2017. Last accessed: 28/3/2017.

[105] 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.

[106] 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.

[107] Ehsan Totoni, Nikhil Jain, and Laxmikant V. Kale. Toward runtime power management of exascale networks by on/off control of links. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 915–922, May 2013.

[108] Keith D. Underwood and Ron Brightwell. The impact of MPI queue usage on message latency. In *Parallel Processing, 2004. ICPP 2004. International Conference on*, pages 152–160. IEEE, 2004.

[109] Keith D. Underwood, Jerrie Coffman, Roy Larsen, K. Scott Hemmert, Brian W. Barrett, Ron Brightwell, and Michael Levenhagen. Enabling flexible collective communi-

cation offload with triggered operations. In *IEEE 19th Annual Symposium on High Performance Interconnects (HOTI)*, pages 35–42. IEEE, 2011.

[110] Keith D. Underwood, K. Scott Hemmert, Arun F. Rodrigues, Richard Murphy, and Ron Brightwell. A hardware acceleration unit for MPI queue processing. In *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 10–pp. IEEE, 2005.

[111] Karthikeyan Vaidyanathan, Dhiraj D. Kalamkar, Kiran Pamnany, Jeff R. Hammond, Pavan Balaji, Dipankar Das, Jongsoo Park, and Bálint Joó. Improving concurrency and asynchrony in multithreaded MPI applications using software offloading. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 30. ACM, 2015.

[112] Jeffrey Vetter and Chris Chambreau. mpiP: Lightweight, scalable MPI profiling. *URL: http://www.llnl.gov/CASC/mpiP*, 2005.

[113] Jeffrey S. Vetter and Andy Yoo. An empirical performance evaluation of scalable scientific applications. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 16–16. IEEE, 2002.

[114] Hans Weeks, Matthew G. F. Dosanjh, Patrick G. Bridges, and Ryan E. Grant. SHMEM-MT: A benchmark suite for assessing multi-threaded SHMEM performance. In *Workshop on OpenSHMEM and Related Technologies*, pages 227–231. Springer, 2016.

[115] Judicael A. Zounmevo and Ahmad Afsahi. A fast and resource-conscious MPI message queue mechanism for large-scale jobs. *Future Generation Computer Systems*, 30:265–290, 2014.