MPI Tag Matching Performance on ConnectX and ARM

W. Pepper Marts*
Center for Computational Research
Sandia National Laboratories
Albuquerque, New Mexico, USA
wmarts@sandia.gov

Matthew G. F. Dosanjh
Center for Computational Research
Sandia National Laboratories
Albuquerque, New Mexico, USA
mdosanj@sandia.gov

Whit Schonbein*
Center for Computational Research
Sandia National Laboratories
Albuquerque, New Mexico, USA
wwschon@sandia.gov

Ryan E. Grant*
Center for Computational Research
Sandia National Laboratories
Albuquerque, New Mexico, USA
regrant@sandia.gov

Patrick G. Bridges
Department of Computer Science
Univeristy of New Mexico
Albuquerque, New Mexico, USA
bridges@cs.unm.edu

ABSTRACT

As we approach Exascale, message matching has increasingly become a significant factor in HPC application performance. To address this, network vendors have placed higher precedence on improving MPI message matching performance. ConnectX-5, Mellanox's new network interface card, has both hardware and software matching layers. The performance characteristics of these layers have yet to be studied under real world circumstances. In this work we offer an initial evaluation of ConnectX-5 message matching performance. To analyze this new hardware we executed a series of micro-benchmarks and applications on Astra, an ARM-based ConnectX-5 HPC system, while varying hardware and software matching parameters. The benchmark results show the ConnectX-5 is sensitive to queue depths, and that hardware message matching increases performance for applications that send messages between 1KiB and 16KiB. Furthermore, the hardware matching system was capable of matching wildcard receives without negatively impacting performance. Finally, for some applications, a significant improvement can be observed when leveraging the ConnectX-5's hardware matching.

CCS CONCEPTS

• **Networks → Network performance analysis**; *Network measurement*; Network adapters;

ACM Reference Format:
W. Pepper Marts, Matthew G. F. Dosanjh, Whit Schonbein, Ryan E. Grant, and Patrick G. Bridges. 2019. MPI Tag Matching Performance on ConnectX and ARM. In *Proceedings of (EuroMPI).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3343211.3343224

*Also with the Department of Computer Science, University of New Mexico.

EuroMPI, 2019

ACM ISBN 978-1-4503-7175-9/19/09…$15.00
https://doi.org/10.1145/3343211.3343224

1 INTRODUCTION

The Message Passing Interface (MPI) is the *de facto* standard for inter-process data exchange in HPC applications. Under MPI, payloads of incoming messages are delivered to receiver-side buffers through a matching semantic: the receiver posts a list of parameters (communicator, source rank, and user-defined tag) with associated buffers, and when those of an incoming message match, data is delivered to the specified location. While well-tuned and synchronized codes can exploit predictability in communication patterns to manage the overhead of having to search for a match [16, 17, 26], less-well-tuned codes, or codes that have the potential to generate large numbers of messages with unpredictable arrival ordering, may experience significant impact to application performance due to message matching overheads [31].

The potential impact of message matching overhead has been explored in several recent works attempting to reduce this overhead, e.g., through more efficient data structures such as hash tables/binning [4, 11, 18, 25], exploitation of time and space locality [13], and enabling parallelism through vector instructions and partial matching [15] and thread-base parallelism [14]. Recently, network vendors have also moved to address matching overheads by offering offloaded, hardware support based in the local HCA [1, 12].

In this paper, we provide an initial evaluation of the network performance on a ConnectX-5 and ARM HPC system. Our experimental platform is the Astra supercomputer, housed at Sandia National Laboratories, which is the world's first ARM-based petascale supercomputer. We quantify the impact of several relevant parameters on network bandwidth, including queue lengths, hash table collisions, message sizes, and hardware matching under various conditions. Furthermore, we consider the scaling performance of two representative scientific applications, LULESH and the Fire Dynamics Simulator (FDS). To the authors' knowledge, this is the first study of the ConnectX-5 network tag matching and the first network study of this scale on an ARM HPC system.

This paper makes the following contributions:

- The first study of the performance of the tag matching capabilities of the ConnectX-5 network;
- A fine grained microbenchmark analysis of the hardware and software matching components of the ConnectX-5;
- A high level study of the application impact of the ConnectX-5's matching implementation.

The rest of the paper is structured as follows. Section 2 presents the background relevant to this work. Section 3 presents the experimental design and setup. Section 4 presents the results of this study. Section 5 presents a discussion of the implications of this study. Section 6 presents the related work for this study. Finally, Section 7 presents the conclusions and future work of this study.

2 BACKGROUND

MPI message matching is a core functionality of MPI which enables point-to-point send-receive communication. For some MPI implementations, send-receive communication is leveraged in some collective communication calls, further increasing the importance of send-receive performance to application codes. The message matching engine is an implementation of the MPI message matching functionality. It is used to match incoming data to a user buffer specified in a `MPI_Recv` function call. This matching is done by comparing three identifiers, the communicator, a tag, and the rank of the sender. It also must both ensure a message ordering from a single process to adhere to the non-overtaking rule in the MPI specification, and it must support wild cards on receive operations for both rank and tag [28].

While message matching engines have not been considered performance critical to MPI in the past, there are strong indications that Exascale applications will require optimized message matching engines. A 2017 survey of Exascale Project application developers showed that a majority of developers expected their projects to leverage multithreaded MPI [5]. Additionally, a study of multithreaded halo exchanges has shown that even simple scalable communication patterns can be problematic for message matching engines when converted to threaded versions [31]. This multithreaded performance has also influenced threading proposals for MPI like finepoints [21, 22] that are currently being refined in an MPI Forum working group.

Traditionally, MPICH and MPICH-derivative MPI implementations have used a pair of linked lists to implement matching: a posted receive queue (PRQ) and an unexpected message queue (UMQ). Recently optimizations have been made in some of the major vendor's drivers to use a hashed-binning solution where the tag is hashed to select one of many linked lists. These include Intel's PSM2 driver and Mellanox's ConnectX-5 driver. The former uses a randomized hash that selects from 64 bins and the latter uses an ordered hash (the modulo of a 64 bit field that includes the MPI message tag) to select from 1021 bins. Additionally, network vendors have also moved to address matching overheads by offering offloaded, hardware support based in the local HCA including Atos [12] and Mellanox [1].

Academic works have proposed experimental matching engine architectures. These include more efficient data structures such as hash tables/binning [4, 11, 18, 25] and 4-D lookup tables [35], exploitation of time and space locality [13], enabling parallelism through vector instructions and partial matching [15] and thread-base parallelism [14], and creating message matching channels [19, 20].

The particular message matching engine that we are examining in this paper is Mellanox's ConnectX-5. ConnectX-5 matches based on a single 64-bit tag (a UCX tag), that MPI configures to contain a full set of matching data including MPI tag, MPI rank, and MPI context ID. ConnectX-5's message matching engine contains two layers, a software layer and a hardware layer. The software layer is open source and can be viewed as part of Open UCX. This layer is a hash binning system[1] with 1021 bins and a hash function that takes the mod 1021 of the upper and lower 32-bits of the UCX tag and combines the result using an bitwise `xor`. The hardware layer is proprietary and the details are not publicly released. Through the environment variables that are available to tune the NIC we're able to see that hardware matching only happens for messages above a certain threshold which defaults to 1KiB. While there may be other scenarios where the message matching engine will default to software message matching the message size threshold is the most clearly defined case. Because these two layers can both be active at the same time, we can infer that there is methodology to keep the matching information consistent between both layers.

3 METHODOLOGY

In this section we will discuss our experimental methodology. In particular we will discuss the hardware evaluated, the micro-benchmarks used to evaluate fine-grained performance, and the proxy-applications and production applications used to evaluate real world impact.

3.1 Hardware

The system we used for these experiments was the Astra supercomputer at Sandia National Laboratories. Astra is the world's first Petascale ARM computing system and comprises 36 compute racks, each containing 18 HPE Apollo 70 chassis. Each chassis holds four dual-socket compute nodes, for a total of 5184 processors. These processors are 28-core Cavium ThunderX2 CN9775 chips operating at 2GHz. The two ThunderX2 processors in each node communicate using a 600Gb/s cache coherent interconnect.

The network is 4x EDR (100Gb/s) Infiniband Mellanox ConnectX-5 hardware. Nodes communicate with the network interface via 8x PCIe 3 (16GT/s). The ConnectX-5 NIC provides hardware assistance for MPI message matching.

Nodes are arranged in a three-level fat tree topology. At the leaves, each rack is divided into three groups of six chassis (24 nodes), and each node in a group is connected to a 36-port level-1 switch. The remaining 12 ports are connected to 30 L2 switches, which connect to 3,540 port L3 switches.

3.2 Benchmarks

To evaluate network performance on the cluster, we collected data using a modified version of the OSU Micro-Benchmarks suite (OSU)[13, 29]. These modifications can be categorized as serving two purposes, (i) testing different list lengths and, (ii) matching the environment of a well written application. To test the network performance at different list lengths, additional unmatched receives are posted in advance of the performance testing. To match the environment of a well written application, all receives are preposted and the cache is cleared before each iteration.

[1] A hash binning data structure is similar to a hash table but with a fixed number of bins that are linked lists of elements

In order to reduce search queues OpenUCX uses a software binning system, where each receive is binned based on its UCX tag mod 1021 (where a UCX tag is a 64 bit field, which OpenMPI populates with the MPI tag, source rank, and context id). Knowing this, we were able to select MPI tags for the unmatched receives that would cause a specific frequency of binning collisions between matched and unmatched receives. Mellanox hardware matching was enabled via the environment variables `UCX_RC_MLX5_TM_ENABLE` and `UCX_DC_MLX5_TM_ENABLE`. The hardware matching message size threshold was set to zero with the environment variable `UCX_TM_THRESH` (default threshold is 1KiB). This allowed us to evaluate hardware matching across a wide variety of message sizes. All benchmarks were run using OpenMPI version 3.1.3 and GCC version 7.2.0 on Sandia's Astra cluster with two 28-core Cavium ThunderX2 processors per node and a Mellanox ConnectX-5 network. Message sizes were varied between $1B$ and $1MiB$ and queue lengths were varied between 1 and $32k$ tags. Each trial was performed with both hardware matching enabled and disabled for 0%, 1%, 10% and 100% collision rate. Each trial was run twenty times, with the mean and standard deviation across the runs presented as each data point.

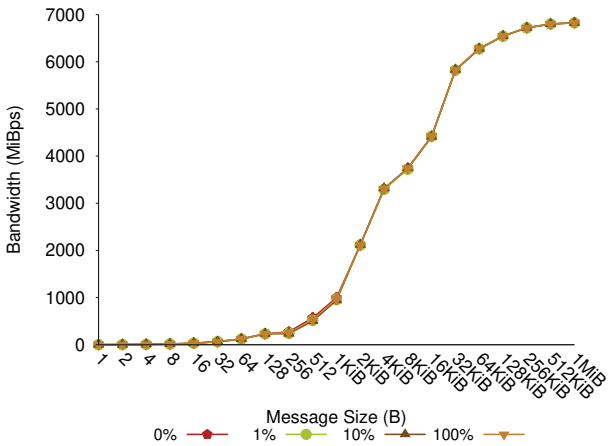
3.3 Applications

To evaluate the application impact we ran two applications/proxies under a variety of matching schemes. The two applications/proxies include Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) and the Fire Dynamics Simulator (FDS). These were run with three different matching techniques; software only, software with hardware, and software with one bucket. These experiments are reported as the average of 5 runs.

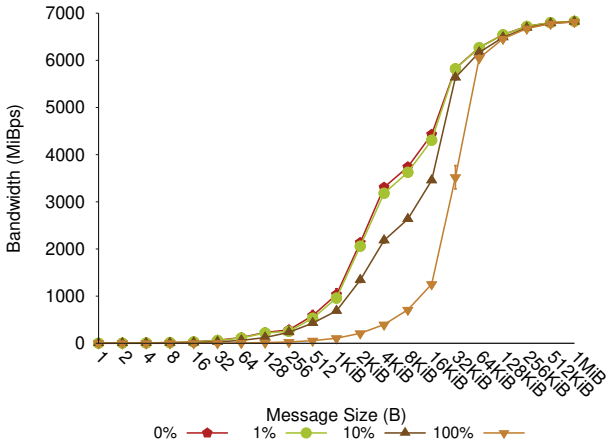
LULESH [23] is a proxy application from Laurence Livermore National Laboratory that represents an important hydrodynamics code. The version used in this paper is 2.0. It has a seven distinct phases with different communication needs, each of which has been highly optimized for scalable performance. For the purposes of this paper we used the default problem size.

FDS [27] is a full application that has been used in many matching studies for its all-to-one communication pattern in its exchange diagnostics phase. The version we use in this paper is 6.7.1 which has optimized this behavior into a single all-to-one rather than the 23 concurrent all-to-ones present in the commonly used 6.5.3 version. Despite this, the all-to-one communication pattern can be problematic from a scalability perspective and is the closest modern application to the expected behavior of multi-threaded application [31]. The input deck used in these tests is a weak scaled version of a circular burner.

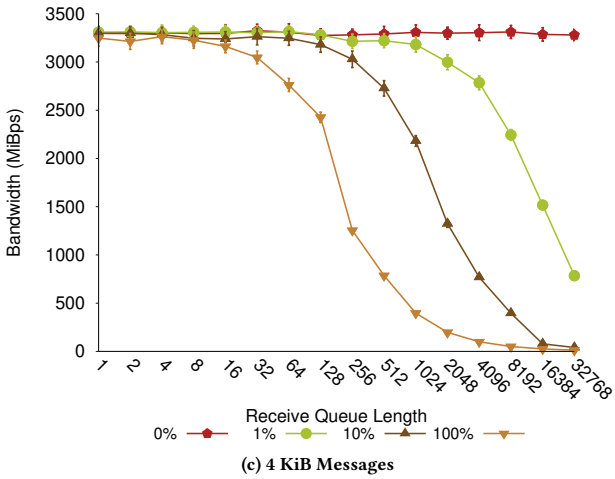
There were three matching environments that we ran the tests for the application codes. Software only, where the only optimization enabled is the UCX software matching layer, is the default on this system. To enable hardware message matching, we set the `UCX_RC_MLX5_TM_ENABLE` and `UCX_DC_MLX5_TM_ENABLE` environment variables, unlike the micro-benchmark tests, we left `UCX_TM_THRESH` at it's tuned default of 1KiB. For the software one-bucket tests, we modified the UCX driver to set `UCP_TAG_MATCH_HASH_SIZE` to 1. This is a fairly reasonable proxy for traditional matching as all of the matching elements are filtered into a single list without significantly changing the underlying software stack.



(a) Queue Depth 0



(b) Queue Depth 1024



(c) 4 KiB Messages

Figure 1: Software-Only Bandwidth

4 EXPERIMENTAL RESULTS

In this section we present our study of message matching on ConnectX-5 and ARM.

4.1 Bandwidth and Message Rate

As discussed in Section 3 the ConnectX-5 network features two matching layers; a software implementation and a hardware acceleration layer. The software layer is publicly available as part of the OpenUCX driver and is a binning system based on the UCX tag mod 1021. The hardware implementation is proprietary. The details of the hardware accelerated matching implementation are not known. However, the operation of the hardware matching is well documented with a number of environment variables provided to influence the behavior of the matching hardware.

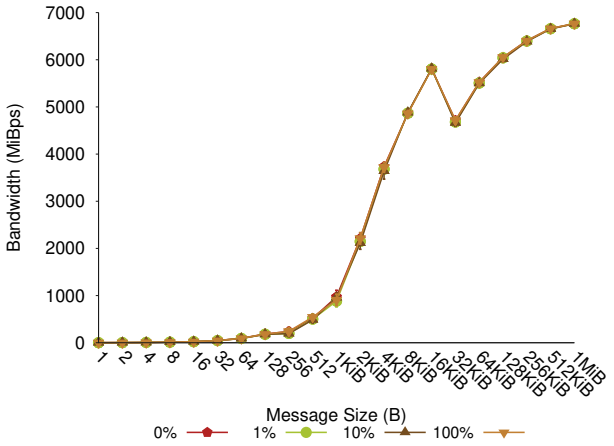
4.1.1 Software Only Matching. To evaluate the hardware accelerated matching implementation, we first need to establish a baseline. In this case our baseline is the software implementation layer with hardware matching disabled. To evaluate performance we ran a series of test using our modified OSU bandwidth benchmark with different numbers of unmatched receives increasing what we call the receive queue depth. We also run using various collision rates, the rate at which the unmatched receives are placed in the same bin as the receives for the actual bandwidth test.

Figure 1a shows results from the modified OSU microbenchmarks bandwidth test with no preposted unmatched receives and no hardware matching. This serves as a baseline to test the software matching implementation. There is a clear transition from eager to rendezvous at 16KiB messages.

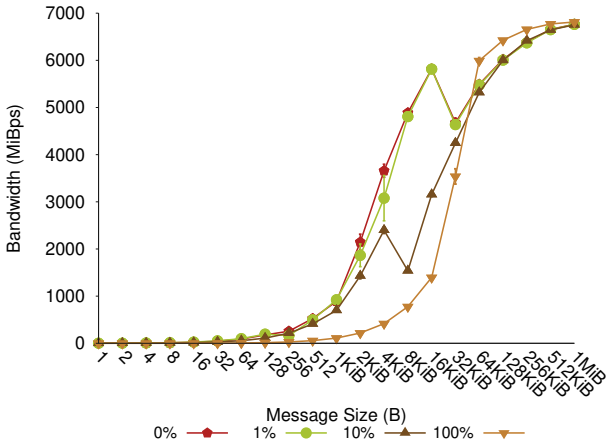
Figure 1b shows the same benchmark after adding 1024 unmatched messages to the queue. For a 0% collision rate the tag binning system works well, with no significant changes in bandwidth. Similarly for a 1% collision rate, differences are minor. At this rate there were an expected 10 messages that collided with the matching receives during each run, and the additional queue search time was insignificant compared to the length of the test. At a 10% collision rate the effects became more severe, reducing bandwidth by more than a third for 4KiB messages with an expected 102 colliding receives. At a 100% collision rate all 1024 unmatched receives collide, and the eager message protocol message rate is dominated by the queue search. Interestingly after the transition to the rendezvous message protocol, bandwidth does not vary significantly with collision rate. The latency inherent in the larger message sizes and the resulting lower message rate can hide the time spent message matching even in more extreme cases of 100% collisions.

Figure 1c shows that the bandwidth of a 4KiB message size across the receive queue depth. This data shows what we expect from binning systems; the performance is directly dependent on the combination of queue depth and collision rate. Because the effective search depth is the number of entries in the selected bin, it can be evaluated at $queue\_depth * collision\_rate$. This is reflected in the data by the decreased performance as collision rates increase for the same queue depth.

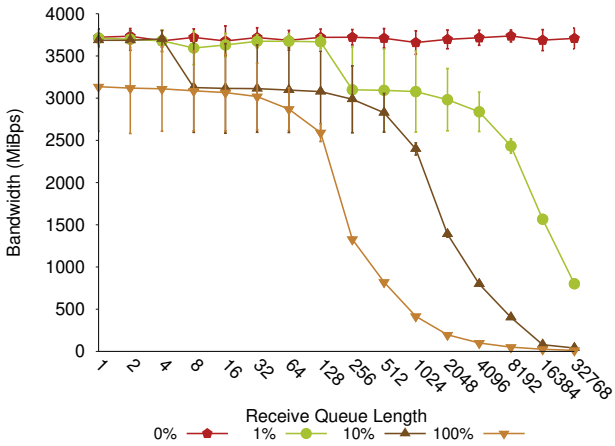
4.1.2 Software + Hardware Matching. Figure 2 shows benchmark results after enabling hardware tag matching for benchmark



(a) Queue Depth 0



(b) Queue Depth 1024



(c) 4 KiB Messages

Figure 2: Software + Hardware Matching Bandwidth

configurations identical to those of the software-only experiments. We see that the performance gains of the new hardware are highly

dependent on message size, with messages of approximately 1KiB to 16 KiB seeing improved performance from offloading tag matching to the NIC. For example, messages of 8KiB enjoy 4864.3 MiBps in the zero-queue-depth hardware matching case (Figure 2a) in comparison to the 3744.9 MiBps observed in the software-only case (Figure 1a). Messages of size 16KiB had an increase of 1119.4 MiBps, a 31.4% increase in bandwidth when enabling hardware. Outside of this range however, there is actually a reduction in bandwidth. 32 bit messages are 27.1% slower with hardware enabled. For smaller messages, this reduction is likely due to overheads of hardware matching not being masked by overlap; the additional latency from hardware message matching becomes the performance bottleneck. The performance dip that occurs between 16KiB and 256KiB is unexpected and warrants further exploration that we leave to future work.

Figure 2b shows the results of hardware matching with 1024 elements in the queue. In this figure we begin to see the differences between collision rates. This is generally expected, as an increased collision rate leads to a higher effective search depth. Comparing the this to the software only bandwidth results, 2KiB to 16KiB message sizes show a performance increase ranging from 0.48% to 31.4%. As the collision rate increases, the performance in this range trends downward, with 16KiB messages at 100% collision rate seeing a 11.3% improvement. Outside of this range, the performance becomes worse with hardware matching on, which again is an unexpected result for larger messages which we will explore further in future work.

Figure 2c shows how hardware matching handles 4KiB message with an increasing number of matching entries. This presentation shows us where hardware message matching becomes ineffective with respect to collision rate. As we can see from the the data, there is significant performance drop when the expected value for collisions becomes 1 (when queue depth $*$ collision rate = 1). This seems to indicate that the benefit of hardware matching is limited to entries that are first in their bucket.

To further evaluate the messaging scenarios that might benefit hardware tag matching we plotted runs with hardware tag matching on and off side-by-side. Figures 3 through 6 demonstrate the effect of tag collision rates on hardware tag matching performance. Although it is a benefit to use hardware tag matching for configurations with low collision rates and messages between 1KiB and 16KiB, it can actually be an impediment for configurations with higher tag collision rates. Once the queue depth becomes large enough that even a single collision is expected, hardware matching is quickly surpassed by software matching performance and the variance between runs increases dramatically. This puts further impetus on the programmer to correctly manage tags. Not only is communications performance at stake, but the hardware offloading present on these next generation systems depends on it.

4.1.3 Wild card Support. Wild cards present an interesting challenge to the matching engine. Any incoming message could potentially be a match for a wild card receive regardless of hash value. In order to test the ConnectX-5's support of wild cards we designed two tests. First, we tested how the hardware matching's performance changes with the existence of wild cards. To achieve this we post a `MPI_ANY_TAG` receive and send a message that matched it before the iteration begins. This tests if the existence of a wildcard prior to a given communication impacts performance. This would occur if hardware changed to an alternative method upon observing a wildcard and did not revert back to its previous method when there are no more wildcards. The second test explores how hardware matching is affected by having wild cards enqueued. To achieve this we post a `MPI_ANY_TAG` receive before the iteration and send a message that matched it after the iteration ends. For both of these tests, the wild card receive is added after the receives are posted for the iteration. This was done to make the two tests consistent and allow for a wild card receive to remain enqueued during the test. Using this test we can compare against hardware and software message matching to determine when hardware matching remains on or is negatively impacted by wild cards.

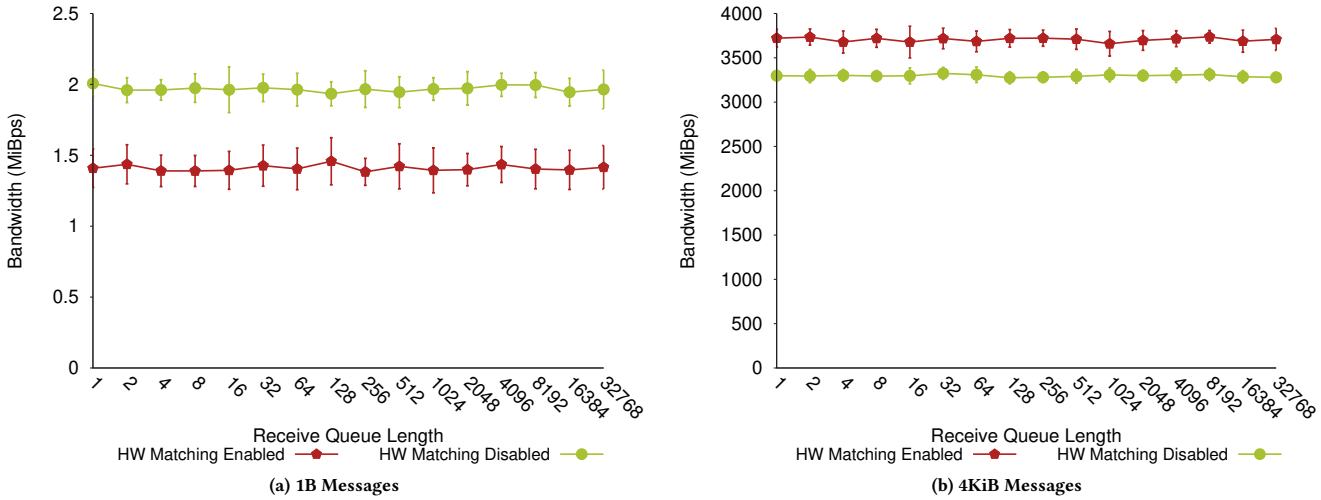
Figure 7 shows a comparison of three wild card scenarios; the two tests described above and a baseline that doesn't use wild cards. The results in this figure show roughly consistent performance through out the three tests. This indicates that the hardware offload does not appear to be impacted by wild cards. The exception being for wild cards being used before iteration for message sizes below 2KiB. This is likely an artifact of the test; because the send designed to match the wild card is sent before the iteration begins, any initial cost of moving the matching information to the NIC happens outside of the timed region.

4.2 Applications

The applications we studied were selected to represent two application classes. LULESH was selected as representative of highly optimized HPC applications. This class of applications have spent a significant amount of time optimizing the application's MPI interactions to reduce communication and message processing overheads. FDS represents typical HPC applications, ones that use MPI correctly but may have complex communication requirements, suboptimal implementations, and/or scale sensitive communication patterns. Additionally, we expect future highly optimized multithreaded applications may experience similar challenges with MPI interactions [31] as codes like FDS do today.

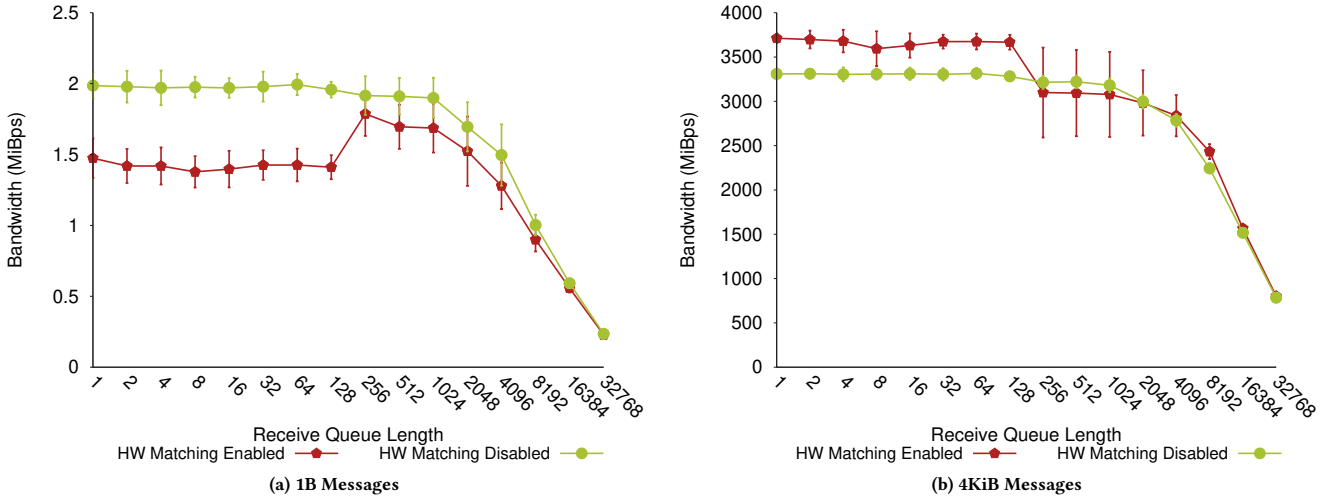
Figure 8 shows the results for LULESH under three different matching environments. These tests used up to 128 nodes and were run for each process count available (LULESH requires a cubic number of processes). The three matching environments perform within error for these experiments. This is in line with what studies have shown of highly optimized HPC applications, they do not spend a significant amount of time in the matching engine and thus do not see a runtime impact from improved message matching [26].

Figure 9 shows the results for FDS under the different matching environments. The data shows that hardware matching greatly improves the application runtime over UCX's software implementation. As these runs use the tuned threshold for hardware matching, this reflects the data shown above. The unexpected result out of these experiments was the improved performance of the one bucket software implementation over the default software implementation. This is likely due to the hashing overhead outweighing the cost of a linear search. Memory prefetching is easier to optimize for linked



(a) 1B Messages (b) 4KiB Messages

Figure 3: Hardware vs Software with no collisions



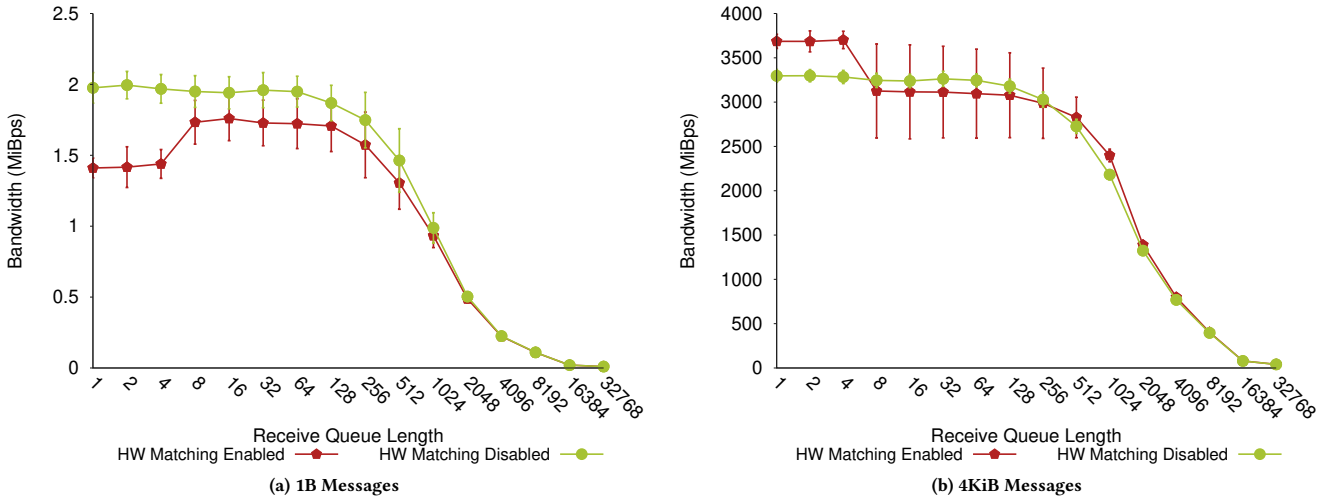
(a) 1B Messages (b) 4KiB Messages

Figure 4: Hardware vs Software with 1% collisions

list iteration than hash table look-up. Additionally hash tables often use additional cache resources that can increase overhead to look-ups.

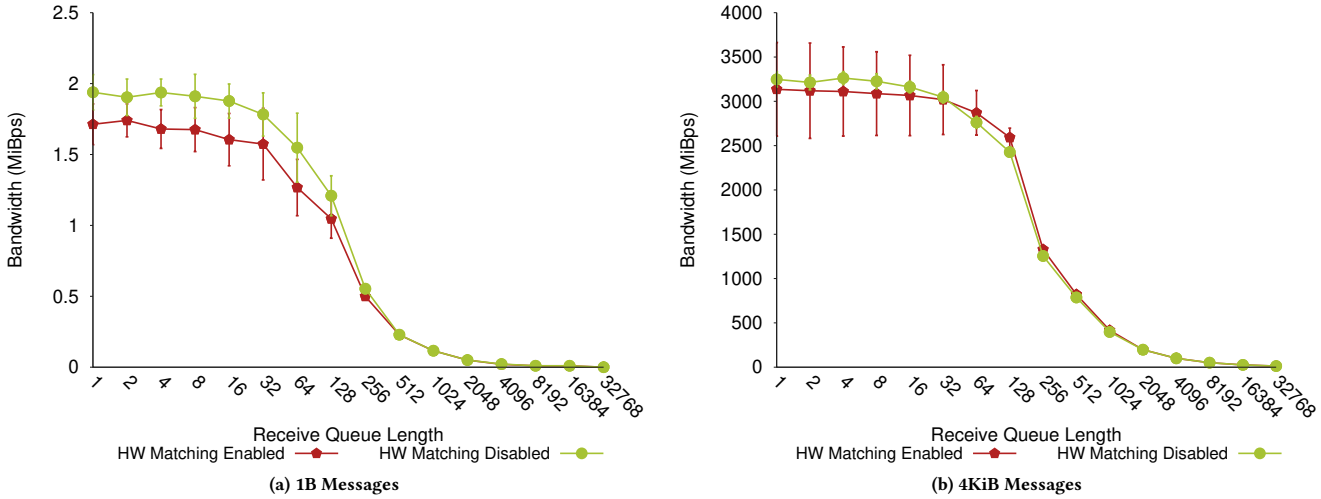
5 DISCUSSION

What does this study show about the ConnectX-5 matching scheme? This study has explored the interactions between the two layers of ConnectX-5's message matching engine. The software layer is available through the Open UCX repository and utilizes a hash binning data structure containing 1021 bins. The hash function takes the mod 1021 of the upper and lower 32 bits of the UCX tag and combines them using bitwise `xor`. The hardware layer resides on the NIC, and performs UCX tag matching if a message is larger than a configurable threshold. We found through our experiments that small messages actually incur a penalty from hardware matching, while larger messages (those over 1KiB) see a benefit. This indicates that the on-NIC matching takes more time than matching on CPU, but allows larger message sizes to see improved performance from overlapping matching with another task. Additionally, we discovered that the hardware matching layer is highly dependent on the hash binning data structure, and will only accelerate requests that are the first element in a bin. Our wild card experiments show that wild cards do not negatively impact hardware matching, and that there is an initial cost for the first message that accesses the matching engine. Finally, our application experiments show that while hardware matching is beneficial to applications, the software layer hash-binning system has trade-offs



(a) 1B Messages (b) 4KiB Messages

Figure 5: Hardware vs Software with 10% collisions



(a) 1B Messages (b) 4KiB Messages

Figure 6: Hardware vs Software with 100% collisions

between the caching behavior and the message matching speed-up that warrant further study.

1021 is a large number of bins, why worry about collision rate? Bin collision rate appears to affect the performance of ConnectX-5's software and hardware matching layers. This means that it's likely that the hardware implementation is dependent on the binning structure. While there are a large number of bins there are a number of ways where the collision rate could be significantly higher that one might expect. If we assume that tags are picked on a random basis, collisions can be more common than expected as, given these assumptions, this becomes a version of the birthday problem. Even though the individual collision rate is less than 0.1%, adding 100 random tags in the queue would result in a expected value of 9.4 collisions. Finally, as we expect multithreaded MPI use to cause the number of distinct messages to increase and the reliable ordering to decrease [31], the probability of conflicts will greatly increase.

6 RELATED WORK

MPI message matching previous work can be broken into two main categories: matching list performance characterization and alternative message matching approaches. In this section we will review work done in both of these areas and how it relates to more recent research that we have covered in this paper.

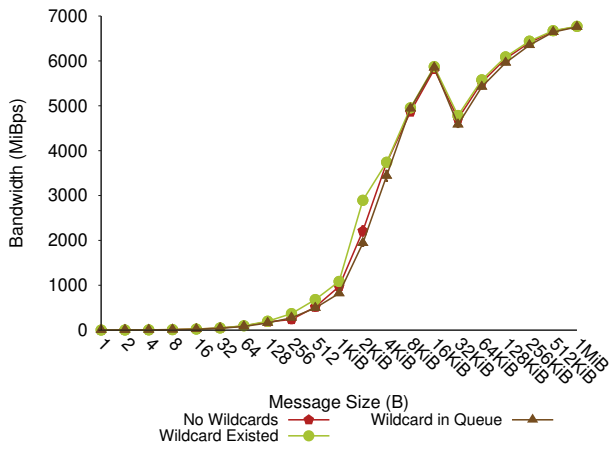


Figure 7: Bandwidth vs. message size on Astra with no posted unreceived messages, added `MPI_ANY_TAG` messages, and hardware tag matching enabled.

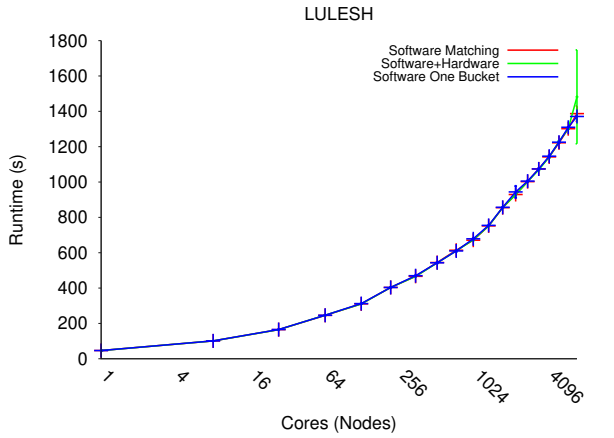


Figure 8: LULESH

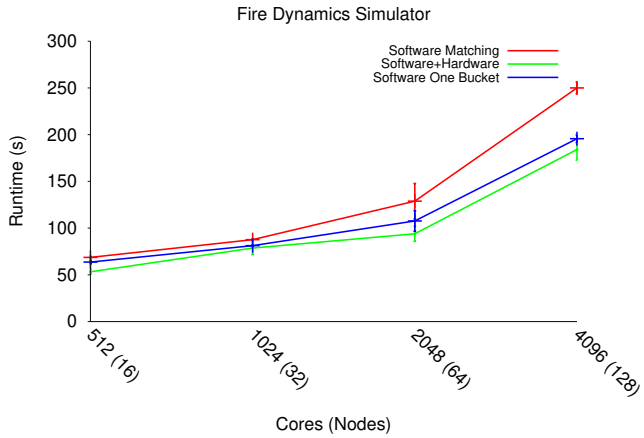


Figure 9: The Fire Dynamics Simulator application using 32 processes per node

6.1 Efficient MPI Message Matching

MPI message matching has been explored in the context of evaluating its performance [34] and investigating the impact of match list length [32] and performance on different types of CPU architecture [2]. Vetter et al. conducted an early study on communication overhead with several scientific applications were they found that the issue of communication overhead was worthy of further investigation [34]. Underwood and Brightwell were the first to build MPI message matching microbenchmarks to study the impact of long list length and unexpected messages [32]. This was followed by work attempting to overcome such overheads by using processing-in-memory operations on a very-wide ALU in order to process multiple matches at once in memory. With the introduction of manycore architectures, Barrett et al. assessed the match performance of MPI with many different CPU architectures [2]. They found that manycore architectures can have an order of magnitude worse performance than a tradition big-core out of order processing core.

The length of an MPI match list can be an important factor in overall performance. Unexpected messages can lead to performance issues when they are sufficiently large in number at any given time. Keller and Graham looked at MPI applications and the impact of unexpected messages on performance. They found that for a subset of scientific applications, processing unexpected messages can be a significant bottleneck to performance [24]. This finding is corroborated by earlier work that focused directly on MPI unexpected list performance using microbenchmarks. Brightwell et al. have investigated this impact with several microbenchmarks and the impact of unexpected message queue length on overall observed communication latency [7, 8, 10]. Recently, studies by Ferreira et al. and Levy et al. have developed and use a simulation based framework to explore the matching characteristics of modern applications [16, 17, 26]. To explore the potential implications of multi-threaded MPI access on message matching Schonbein et al. created a multithreaded halo exchange benchmark [31].

6.2 Matching Techniques

Several matching techniques have been proposed to avoid performance degradation. Dang, Snir and Gropp proposed [11] a multi-threaded hash table approach to matching for MPI. Unlike other techniques in matching, this technique relies on a concurrent hash table design that is highly scalable. The concurrent nature of the hash table requires that no wildcards can be used in MPI messages at all. This constraining of the MPI model allows for more concurrency that allows multiple threads to interact with MPI efficiently. In contrast, Flajslik et al. demonstrated a hash-map keyed to use the entire set of matching criteria [18]. This allowed them to include wildcards in a hashing-based matching scheme. Their design requires setting a user configurable number of bins to which message match requests are posted. They use 256 bins in the default configuration, allowing for list lengths to ideally be divided in length by that amount (in practice the division will not always be equal). This approach seems to have solved major long list match performance issues, but the approach has a small overhead in the hash mapping for lists of any size, and therefore has higher overhead than a traditional list when the match would be near the front of a

traditional linked list. Due to the fact that many applications have tuned their match list performance over time to have the vast majority of matches occur near the beginning of the list, this approach may not be ideal for some applications. Indeed, there has been work in allowing an MPI implementation to dynamically swap between a hash-table and a traditional list [4].

Some solutions to MPI matching have been aimed at specific compute architectures, namely GPUs. Klenk et al. proposed a solution for matching on GPUs that used two phases, a scan phase and a reduce phase, that could take advantage of the large amount of concurrency available [25]. Unfortunately, there are no MPI implementations that run on GPUs today, so the technique is not immediately applicable to the MPI state of the art. It is also unknown what overheads exists for short lists or the case of the first match element being the correct match as the work only used medium and large size lists for performance evaluation.

Alternative match list structures based on new data structure layouts have also been explored. Zounmevo and Afsahi showed that a 4-dimensional list structure could be used to accelerate matching. It works by decomposing the list into a 4D lookup that allows skipping portions of the list where the match cannot occur [35]. Other approaches have sought to create new queues dynamically to reduce the lengths of matching lists by separating out traffic from specific source nodes as the traffic is observed arriving at the destination [19, 20]. Dosanjh et al. have explored the effect of data locality on matching [13], increase parallelism through fine grained matching engines [14], and using vector units and reduced fidelity filters to accelerate message matching [15].

MPI message matching has also been addressed by creating hardware designed to offload the matching processing itself. Examples of such efforts are the Portals communication API [3] that defines an interface for MPI message matching on hardware that is also descriptive of general hardware design. Approaches have been done in FPGAs and with TCAMs [33]. The Seastar interconnect [9] is an early example of a Portals MPI matching offloading NIC, but it did so with a general CPU, much like early designs that could be adapted to perform message matching like the Quadrics network QSNet II [30] and the Myrinet network [6]. More recent examples of message matching NICs include the Bull-Atos BXI NIC [12] that implements the Portals interface and ConnectX-5 NICs from Mellanox that also perform message matching [1]

7 CONCLUSIONS

In this work, we've provided an initial evaluation of MPI performance on a Mellanox ConnectX-5 and ARM HPC system. The ConnectX-5 network combines both hardware and software optimizations to reduce message matching overhead. To explore the impact of these optimizations on network bandwidth, we utilized a modified OSU benchmark to vary queue lengths, hash table collisions, message sizes, and hardware matching parameters. Furthermore, we assessed their impacts for two representative applications/proxies, LULESH and FDS.

The OSU Benchmark results show that the ConnectX-5 is sensitive to both message queue depth and tag collision rate. We demonstrated that hardware message matching increases performance for applications that send messages between 1KiB and 16KiB, validating the default choice of the threshold parameter `UCX_TM_THRESH`. This result is sensitive to high rates of tag collision, but is promising for optimized applications. The hardware tag matching system was capable of matching `MPI_ANY_TAG` wildcard receives without offloading them to software or negatively impacting performance.

The performance impact of these matching layers on applications is tied to the level of optimizations made for traditional matching engines. Because the hardware and software layers for the Connect X5 matching scheme are network specific, their performance impacts are not portable across systems. For this reason, highly optimized HPC applications opt to do the matching optimization within the application. Therefore, matching is not a significant problem for that class of applications. For regular applications, a significant improvement can be observed when leveraging software and hardware matching.

This initial study suggests multiple paths for future work. For example, ConnectX-5 HCAs include bounce buffer optimizations, controllable through an additional environmental parameter (`UCX_TM_MAX_BCOPY`); the current analysis may be extended to include interactions with this parameter. It may also be of use to evaluate the performance of ConnectX-5 under conditions that may be expected in future multithreaded MPI scenarios. A current area of work is the generation of long-term performance data on how the performance of Astra, the first AArch64 cluster of its scale, changes as its software stack matures. Additional evaluation of the current software stack is warranted as well, for example, our current study only explores wild cards added as the last element in the message matching engine. In the future, we are looking into expanding this to see how wildcard placement affects performance.

ACKNOWLEDGMENTS

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

REFERENCES

[1] [n. d.]. Understanding MPI Tag Matching and Rendezvous Offloads (ConnectX-5). https://community.mellanox.com/docs/DOC-2583. ([n. d.]). Accessed: 2018-07-25.

[2] Brian W Barrett, Ron Brightwell, Ryan Grant, Simon D Hammond, and K Scott Hemmert. 2014. An evaluation of MPI message rate on hybrid-core processors. *The International Journal of High Performance Computing Applications* 28, 4 (2014), 415–424. https://doi.org/10.1177/1094342014552085 arXiv:http://dx.doi.org/10.1177/109434201455208

[3] Brian W. Barrett, Ron Brightwell, Ryan E. Grant, Scott Hemmert, Kevin Pedretti, Kyle Wheeler, Keith Underwood, Rolf Riesen, Arthur B. Maccabe, and Trammell Hudson. 2017. The Portals 4.1 Networking Programming Interface. (2017).

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

[5] David E. Bernholdt, Swen Boehm, George Bosilca, Manjunath Venkata, Ryan E. Grant, Thomas Naughton, Howard Pritchard, and Geoffroy Vallee. [n. d.]. A survey of MPI usage in the U. S. Exascale Computing Project. *Concurrency and Computation: Practice and Experience* ([n. d.]). in press.

[6] Nanette J Boden, Danny Cohen, Robert E Felderman, Alan E. Kulawik, Charles L Seitz, Jakov N Seizovic, and Wen-King Su. 1995. Myrinet: A gigabit-per-second local area network. *IEEE micro* 15, 1 (1995), 29–36.

[7] R. Brightwell, S. Goudy, and K. Underwood. 2005. A Preliminary Analysis of the MPI Queue Characteristics of Several Applications. (2005).

[8] Ron Brightwell, Kevin Pedretti, and Kurt Ferreira. 2008. Instrumentation and Analysis of MPI Queue Times on the seaStar High-performance Network. *Proceedings of the International Conference on Computer Communications and Networks (ICCCN)* (2008), 590–596. http://ieeexplore.ieee.org/document/4674276/

[9] Ron Brightwell, Kevin T Pedretti, Keith D Underwood, and Trammell Hudson. 2006. SeaStar interconnect: Balanced bandwidth for scalable performance. *IEEE Micro* 26, 3 (2006), 41–57.

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

[11] Hoang-Vu Dang, Marc Snir, and William Gropp. 2016. Towards millions of communicating threads. In *Proceedings of the 23rd European MPI Users' Group Meeting*. ACM, 1–14.

[12] Said Derradji, Thibaut Palfer-Sollier, Jean-Pierre Panziera, Axel Poudes, and Francois Atos Wellenreiter. 2015. The BXI interconnect architecture. *In Proceedings of the 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects, HOTI* (2015), 18–25. http://dl.acm.org/citation.cfm?id=2861514

[13] Matthew GF Dosanjh, S Mahdieh Ghazimirsaeed, Ryan E Grant, Whit Schonbein, Michael J Levenhagen, Patrick G Bridges, and Ahmad Afsahi. 2018. The Case for Semi-Permanent Cache Occupancy: Understanding the Impact of Data Locality on Network Processing. In *Proceedings of the 47th International Conference on Parallel Processing*. ACM, 73.

[14] Matthew G. F. Dosanjh, Ryan E. Grant, Whit Schonbein, and Patrick G. Bridges. [n. d.]. Tail Queues: A Multi-threaded Matching Architecture. *Concurrency and Computation: Practice and Experience* ([n. d.]). in press.

[15] Matthew G F Dosanjh, Whit Schonbein, Ryan E Grant, Patrick G Bridges, S. Mahdieh Ghazimirsaeed, and Ahmad Afsahi. 2019. Fuzzy Matching: Hardware Accelerated MPI Communication Middleware. *19th Annual IEEE/ACM International Symposium in Cluster, Cloud, and Grid Computing (CCGrid 2019)* (2019).

[16] Kurt Ferreira, Ryan E Grant, Michael J Levenhagen, Scott Levy, and Taylor Groves. 2019. Hardware MPI message matching: Insights into MPI matching behavior to inform design. *Concurrency and Computation: Practice and Experience* (2019), e5150.

[17] Kurt B Ferreira, Scott Levy, Kevin Pedretti, and Ryan E Grant. 2017. Characterizing MPI matching via trace-based simulation. In *Proceedings of the 24th European MPI Users' Group Meeting*. ACM, 8.

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

[19] S Mahdieh Ghazimirsaeed, Ryan E Grant, and Ahmad Afsahi. 2018. A Dedicated Message Matching Mechanism for Collective Communications. In *Proceedings of the 47th International Conference on Parallel Processing Companion*. ACM, 26.

[20] S Mahdieh Ghazimirsaeed, Seyed H Mirsadeghi, and Ahmad Afsahi. [n. d.]. Communication-aware message matching in MPI. *Concurrency and Computation: Practice and Experience* ([n. d.]), e4862.

[21] Ryan Grant, Anthony Skjellum, and Purushotham V Bangalore. 2015. *Lightweight threading with MPI using Persistent Communications Semantics.* Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

[22] Ryan E Grant, Matthew G F Dosanjh, Michael Levenhagen, Ron Brigthwell, and Anthony Skjellum. 2019. Finepoints: Partitioned Multithreaded MPI Communication. *ISC High Performance Conference (ISC 2019)* (2019).

[23] Ian Karlin, Jeff Keasler, and Rob Neely. 2013. *LULESH 2.0 Updates and Changes.* Technical Report LLNL-TR-641973. 1–9 pages.

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

[25] Benjamin Klenk, Holger Froning, Hans Eberle, and Larry Dennison. 2017. Relaxations for High-Performance Message Passing on Massively Parallel SIMT Processors. In *31st International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE.

[26] Scott Levy, Kurt B Ferreira, Whit Schonbein, Ryan E Grant, and Matthew GF Dosanjh. 2019. Using Simulation to Examine the Effect of MPI Message Matching Costs on ApplicationPerformance. *Parallel Comput.* (2019).

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

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

[29] DK Panda et al. [n. d.]. OSU Microbenchmarks v5. 1. *URL http://mvapich. cse. ohio-state. edu/benchmarks* ([n. d.]).

[30] Fabrizio Petrini, Wu-chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. 2002. The Quadrics network: High-performance clustering technology. *Ieee Micro* 22, 1 (2002), 46–57.

[31] Whit Schonbein, Matthew GF Dosanjh, Ryan E Grant, and Patrick G Bridges. 2018. Measuring Multithreaded Message Matching Misery. In *European Conference on Parallel Processing*. Springer, 480–491.

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

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

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

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