SAND2019-7222B

August 2018

The Upcoming Storm: The Implications of Increasing Core Count on Scalable System Software

Matthew G. F. DOSANJH [a,1], Ryan E. GRANT [a,b], Nathan HJELM [c], Scott LEVY [a] and Whit SCHONBEIN [a,b]

[a] *Sandia National Laboratories*
[b] *University of New Mexico*
[c] *Los Alamos National Laboratories*

Abstract. As clock speeds have stagnated, the number of cores in a node has been drastically increased to improve processor throughput. Most scalable system software was designed and developed for single-threaded environments. Multi-threaded environments become increasingly prominent as application developers optimize their codes to leverage the full performance of the processor; however, these environments are incompatible with a number of assumptions that have driven scalable system software development.

This paper will highlight a case study of this mismatch focusing on MPI message matching. MPI message matching has been designed and optimized for traditional serial execution. The reduced determinism in the order of MPI calls can significantly reduce the performance of MPI message matching, potentially overtaking time-per-iteration targets of many applications. Different proposed techniques attempt to address these issues and enable multithreaded MPI usage. These approaches highlight a number of tradeoffs that make adapting MPI message matching complex. This case study and its proposed solutions highlight a number of general concepts that need to be leveraged in the design of next generation scaleable system software.

Keywords. Message Passing, multithreading, concurrent communications, MPI, communication middleware

1. Introduction

Processor core counts have been increasing for many years. Along with these increasing core counts, use of thread level parallelism is increasing to adapt code for modern architectures. For some architectures, including alternative architectures like many-core accelerators and ARM-based HPC solutions, leveraging multiple threads is key to leveraging the available performance of the processor. Consequently, interest in using multi-threaded communication is also on the rise.

[1]Corresponding Author: Matthew G. F. Dosanjh, Center for Computing Research, Sandia National Laboratories, Albuquerque, New Mexico, USA; E-mail:mdosanj@sandia.gov

Several architectures in use today provide large numbers of hardware threads per node. For example, as of mid-2018, 4 of the 10 top supercomputers in the world provide more than 200 hardware threads per node. 5 of the remaining 10 systems use GPUs, which is also a highly "threaded" environment but these execution environments are treated as computational accelerators and are not conducive to using a many-threaded communication solution.

With high thread-count hardware, leveraging all of the compute power can require the use of threads. As threading models have become increasingly used in applications, thread-safe middleware has become a highly requested feature as it allows application developers to utilize these services without requiring serialization. Providing functional multithreaded access to middleware is not straightforward as it involves the use of common techniques to enforce serialized access to middleware such as locks. Middleware can also be designed to be more concurrency friendly than simple serialization. However, this can prove difficult for communication middleware as network adapters have fundamental limits to their concurrent operations/commands.

Providing functional multithreaded middleware is an engineering exercise, and the mechanics of concurrency are well-studied. However, the impact on application workflows of multithreaded communication are not well understood. The ordering and contention impacts of applications that use a multithreaded model for communication is a new topic that has only recently begun to be explored. This article will focus on the impacts of multithreading on communication middleware for HPC, examining the issues relating to introducing concurrency, as well as the current and upcoming solutions to the problems encountered in emerging research.

Message passing, as represented by the Message Passing Interface (MPI), is the dominant communication model in HPC. Contemporary scientific codes typically adopt a 'hybrid' model with respect to multithreading and communication through MPI: computational work is multithreaded (e.g., through OpenMP), but communication remains single-threaded. However, a recent survey of application developers involved in the US Department of Energy's Exascale Computing project found a strong majority of developers (86%) want to extend multithreading to communication by having multiple threads concurrently engage in message passing [1].

Unfortunately, engaging in multithreaded communication can cause communication to become the dominant bottleneck. For instance, when multithreaded communication is enabled in MiniFE (a mini-application representative of finite-element codes), communication time begins to overwhelm increased solver efficiency (Figure 1). In short, increased core counts bring increased thread counts, and as application developers incorporate a multithreaded communication model leveraging these threads, the result may be an unfortunate degradation in message passing – and hence application – performance.

In this article, we begin by providing more detail regarding the potential performance implications of multithreaded communication under the message passing paradigm (section 2), and then survey some of the steps currently being taken to ensure this potential storm is avoided as we move towards exascale.

2. The Coming Storm

In this section we introduce the problem of multithreaded message matching. Section 2.1 provides a background on MPI message matching. Section 2.2 addresses the current state

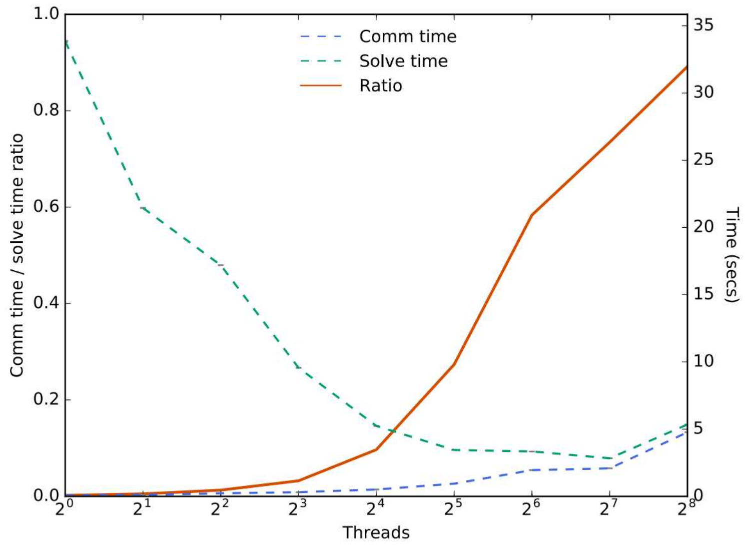


Figure 1. Increasing Communication Requirements for MiniFE with Increasing Thread Count

of MPI message matching for applications that create a serial region to communicate. Finally, Section 2.3 presents the impact multithreaded communication will have on MPI message matching.

2.1. A Brief Overview of MPI Message Matching

MPI groups distinct processing elements of a parallel application into *communicators*, and the processes comprising a communicator are each assigned a unique logical address or *rank*. Furthermore, individual messages can be given user-specified *tags*. This tuple of communicator, rank, and tag enables processes within an application to distinguish between incoming messages so that their payloads can be processed correctly. In other words, to handle incoming data, a process in an application must *match* arriving messages against some record of those it is expecting.

Traditionally, implementations of MPI meet this requirement by creating a pair of linked lists: a posted receive queue (PRQ) to store records of messages the process is expecting, and an unexpected message queue (UMQ) to handle those it is not. The process of matching MPI messages is illustrated in Figure 2. When a message arrives at a process, the MPI matching engine searches the PRQ to determine if a request with the same communicator, rank, and tag has already been posted. If such a request is found, it is removed from the list and the incoming payload is processed. Otherwise, the message is unexpected and is added to the UMQ. Likewise, when the process posts a receive, the UMQ is traversed to determine whether there is a message with the same communicator, rank, and tag that has already arrived but has not yet been matched to a request. If not, the new receive request is added to the tail of the PRQ.

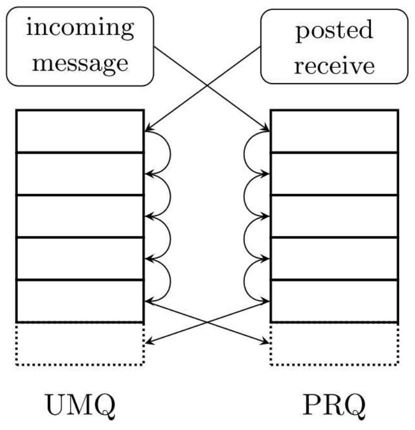


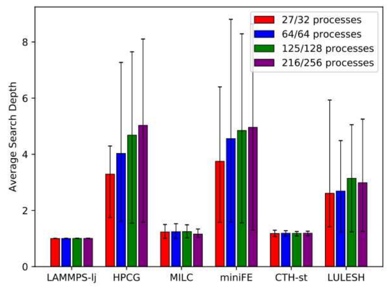
Figure 2. The processes of traditional MPI matching.

2.2. State of the Practice: Single-threaded determinism

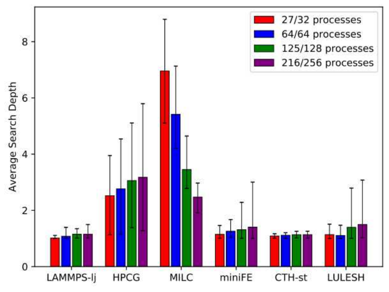
Multithreaded MPI is a rarity in current HPC applications. In cases where computation is threaded (e.g., using OpenMP or qthreads [2]), calls to the MPI library are typically made from a single thread context (e.g., by ensuring that MPI functions are called from a serial region or by 'funneling' requests from multiple worker threads to a single communication thread).

Application	Description
LAMMPS	Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS). A classical molecular dynamics simulator from Sandia National Laboratories [3,4]. The data presented in this paper are from experiments that use the Lennard-Jones (LAMMPS-lj) potential that is included with the LAMMPS distribution.
LULESH	Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH). A proxy application from the Department of Energy Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx). LULESH approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh [5].
HPCG	A benchmark that generates and solves a synthetic 3D sparse linear system using a local symmetric Gauss-Seidel preconditioned conjugate gradient method [6].
CTH	A multi-material, large deformation, strong shock wave, solid mechanics code [7,8] developed at Sandia National Laboratories. The data presented in this paper are from experiments that use an input that describes the simulation of the detonation of a conical explosive charge (CTH-st).
MILC	A large scale numerical simulation to study quantum chromodynamics (QCD), the theory of the strong interactions of subatomic physics [9].
miniFE	A proxy application that captures the key behaviors of unstructured implicit finite element codes [10].

Table 1. Descriptions of the workloads used for evaluating MPI matching performance.



(a) PRQ



(b) UMQ

Figure 3. Average search depth of current HPC workloads. The height of each bar represents the mean search depth over all processes. The error bars extend from the smallest per-process average search depth to the largest per-process average search depth.

When a single thread controls access to the MPI library, programmers can exploit determinism in process communication to ensure that searches of the PRQ and UMQ remain short, even when those queues grow long. As a simple example, if each process first posts receives expecting messages from their neighbors to the north, east, south, and west, then they can issue sends to the south, west, north, and east; in this way, the order of messages arriving corresponds to the order the requests appear in the PRQ, and search depths can be kept small, assuming the application is reasonably synchronized. By adopting strategies such as this, current scientific workloads experience short average search depths and the performance impact of MPI message matching is kept modest [11].

Figure 3 shows average PRQ and UMQ search depths for six workloads that represent important categories of scientific applications. Details about these workloads are provided in Table 1. To generate this figure, we strong-scaled each workload and collected a trace of its MPI operations. With the exception of LULESH, we scaled each workload from 32 to 256 processes. LULESH requires the number of processes to be a perfect cube and so it is scaled from 27 to 216 processes. The data shown in these figures were then collected using a simulator that is based on `LogGOPSim` [12]. `LogGOPSim` is a trace-based, discrete-event simulator for simulating the execution of MPI programs. For the experiments discussed in this section, we configured `LogGOPSim` to use network parameters that correspond to a Cray XC40 [13]. By executing a trace of each of these workloads, the simulator can maintain a detailed record of their MPI matching behavior. Collecting these data using simulation has distinct advantages over using instrumentation provided by existing MPI libraries. MPI libraries are commonly highly-optimized to take advantage of the hardware capabilities of specific machines. These optimizations have the potential to interfere with the accuracy and interpret-ability of MPI message matching statistics [14]. In contrast, the simulator has a simple MPI matching engine that allows us understand how applications use MPI without the confounding influence of hardware optimizations. However, the simulator only allows us to study message matching for single-threaded MPI mode.

As shown in Figure 3, the average PRQ and UMQ search depths are quite short for all six of the workloads. Even the largest per-process average search depths are less than ten. Moreover, the scaling behavior for these applications demonstrate that the average search depth does not grow dramatically with increasing scale. The average PRQ search depth for LAMMPS-lj, MILC, and CTH-st are nearly constant as scale increases. The growth rate of the average PRQ search depth for HPCG and LULESH is a logarithmic function of the number of processes. The average PRQ search depth of miniFE appears to be growing more slowly than the logarithm of the number of processes. With respect to the UMQ, the average search depths for LAMMPS-lj and CTH-st are nearly constant. The average UMQ search depth of MILC *decreases* with scale. HPCG, miniFE, and LULESH appear to exhibit logarithmic growth in their average UMQ search depths.

Overall, this figure demonstrates that the average search depths of current workloads are short and are not growing rapidly as the number of processes increases.[2] As a result, the performance impact of MPI matching when operating in single threaded mode is expected to remain modest unless or until application behavior changes or the per-element cost of match queue searches increases significantly, *cf.* [11]. However, recent surveys indicate that the number of applications that will use MPI multithreaded mode in the future is going to rise [1].

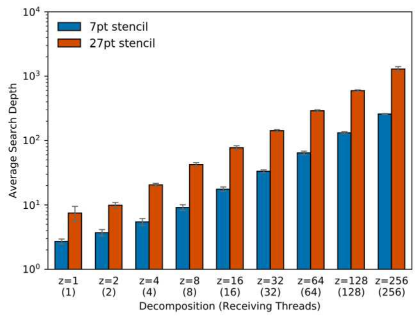
2.3. The Impact of Multithreaded Non-determinism

While search depths in current applications remain low, the MPI standard allows for *multithreaded* access to the communication library (through `MPI_THREAD_MULTIPLE`). This has the potential to disrupt determinism in message ordering: contention between threads for control of message matching data structures can result in a pseudo-random insertion of requests in the MPI library [15]. While some higher-level order may be introduced by applications themselves (e.g., have all threads post receives for the north before those for the east), the ordering of individual requests is not something the developer can control.

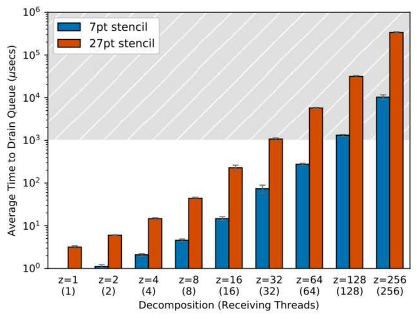
The non-determinism problem is aggravated by having a larger volume of messages with multithreaded code. While some message aggregation is possible, a logical solution for multithreaded communication code is to send at least one message per thread per parallel communication region. This means the *volume* of messages per process expands as a multiple of the number of threads. The combination of pseudo-random insertion order and increase in message volume thus has the potential to create a ‘message matching storm’, stressing the MPI message matching engine and expanding matching overheads.

To explore the impact of non-determinism on message matching in detail, we developed a benchmark [15] that emulates a multithreaded version of a common communication pattern, the halo exchange. In this benchmark, one multithreaded MPI process is designated as the receiver, and a second emulates the group of surrounding sending MPI processes participating in a halo exchange. Following the standard bulk synchronous processing model, the threads of the receiving process concurrently post their receive requests, after which the threads of the sending process compete to issue their sends. This benchmark aims to create a realistic thread level decomposition of a halo exchange, where threads compute on a subdomain of the process’s problem space and only communicates along the intra-process edges the subdomain is in contact with. A version of MPI was instrumented to record the average search depths required to match each incoming message, and the total time elapsed during processing of the queue. The benchmark was

[2]Additional details on the match queue performance of current scientific workloads are available from Ferreira et al. [13].



(a) Average search depth



(b) Average queue drain time

Figure 4. Impact of multithreading ($1\times1\times z$ decomposition).

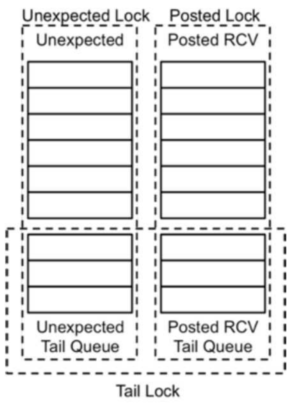
then executed using different decompositions (2- and 3-dimensional) and stencils (5, 7, 9, and 27 points).

Figures 4(a) and 4(b) show results for $1 \times 1 \times z$ decompositions for increasing values of z and two 3-dimensional stencils, executed on an Intel Xeon Phi Knight's Landing system. Even with a modest number of threads participating in inter-process communication, average search depths can inflate by multiple orders of magnitude in comparison to the single-threaded cases described above. Perhaps more striking is the time to process the queue. For instance, molecular dynamics (MD) codes use halo exchanges, and are run at some of the largest scales of any scientific applications. MD codes simulate individual timesteps, typically in the femtosecond range per step. To be productive, an MD code should be able to calculate many microseconds of simulated time per real-world day of execution [16,17]. These requirements enforce a real-world deadline on completing each simulated time step, including both computations and communication (e.g., 1.728μs per step to reach 50μs of simulated time per day). The grey region in figure 4(b) shows the point at which queue processing times require a microsecond or more. The issue, then, is under some decompositions and stencils, communication time may consume the *entire* MD timestep budget currently allocated for both computation and communication. This possibility is troublesome.

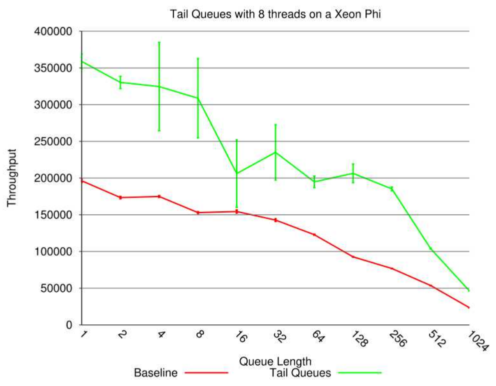
To summarize, current scientific applications keep search depths shallow at least in part by exploiting determinism in message ordering. However, while a semblance of this ordering can be retained under multithreaded communication, scheduling and lock contention introduce noise, resulting in increased average search depths. This situation is exacerbated by a growth in message volume, and can ultimately lead to unacceptable amounts of time spent in communication. Since, as previously noted, developers have expressed the intention to adopt multithreaded communication, steps should be taken to avoid this potential 'storm'. In the following sections, we survey some of the strategies we've explored for addressing the issue.

3. Improving Parallelism

One strategy for addressing the potential effects of matching on performance is to improve parallelism in the matching engine. In this section, we describe some of the strategies we've investigated for doing so.



(a) Tail locks



(b) Tail locks results

Figure 5. The Design and performance of Tail Queues.

3.1. Simultaneous Progress

A straightforward way to make the traditional matching approach (figure 2) thread-safe is to protect it with a lock. However, doing so effectively serializes matching by blocking threads from simultaneously making progress in their respective searches. Providing separate locks for the PRQ and UMQ has the potential to increase multithreaded matching performance by permitting searches to progress on each queue independently of the other, but there remains the problem of coordinating failed searches. As noted in section 2, if no match is found in one queue, the other must be modified to include the unmatched request, creating a race condition.

To address this issue, we designed a new approach that we call Tail Queues. It supplements the unexpected and receive queues with an 'inbox' mediating access to the tails of each (figure 5(a)) [18]. This inbox contains two supplementary data structures storing requests to be appended to the UMQ and PRQ, respectively. These data structures allow for 'lazy appends', that create a separate critical section encapsulating the interactions with the inbox queues. For example, consider a thread that will access the inboxes after searching the PRQ. When it acquires access to the inbox, it inspects the inbox items to be appended to the PRQ and, if none match, adds its request to those to be appended to the UMQ. If an item matches, it is returned and removed from the inbox. In either case, the thread appends the contents of the PRQ portion of the inbox to the actual PRQ, making those items available to future parallel searches.

Figure 5(b) shows results for eight threads accessing a simple Tail Queues data structure. The experiments were run on an Intel Knights Corner Xeon Phi. This implementation focuses on a basic implementation of Tail Queues. While Tail Queues can enable larger degrees of parallelism, the basic implementation allows for two threads to access the matching engine; allowing for one thread to access PRQ and one thread to access the UMQ. In this case the use of Tail Queues increases throughput by roughly a factor of two, which is the maximum we can expect given only two threads can access the data structure at a time. Additionally, this improvement is seen even at smaller list sizes, which makes the technique viable for both modern and future applications.

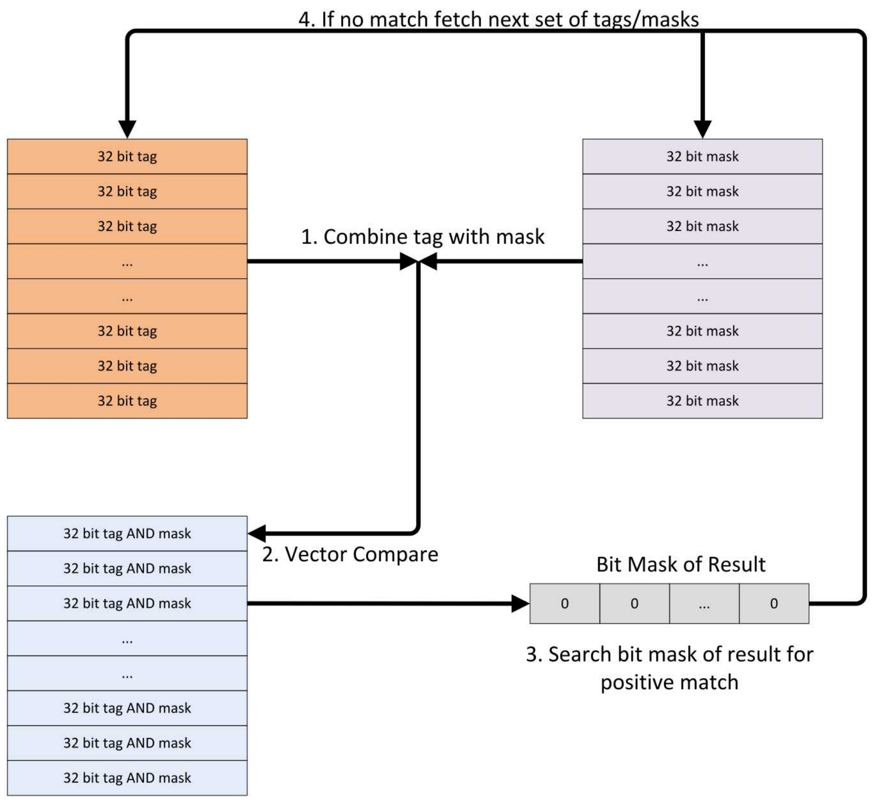


Figure 6. Vector matching.

3.2. *Vector and Fuzzy Matching*

Most if not all contemporary processors include SIMD vector instructions, and these instructions typically contain comparison operations. Consequently, vector instructions provide another opportunity for improving parallelism in message matching.

To investigate the potential of this opportunity, we designed and implemented a modified matching engine utilizing the AVX-512 SIMD instructions offered by Intel's Xeon Phi. Figure 6 illustrates how this works for matching message tags (the process is similar for matching ranks). In this approach, vector matching requires two vector operations each for matching rank and tag (applying the mask and performing the comparison), as well as some followup work to evaluate the resulting bit masks. Each step simultaneously evaluates 16 matching queue entries [19].

The amount of parallelism could be increased by leveraging vector instructions with smaller sizes, below the 8 bytes typically allocated for rank and tag by standard MPI implementations. Fuzzy Matching leverages small bit with vector by *truncating* bits from the tag and/or rank representation, matching only on the least 'significant' bits [19]. The

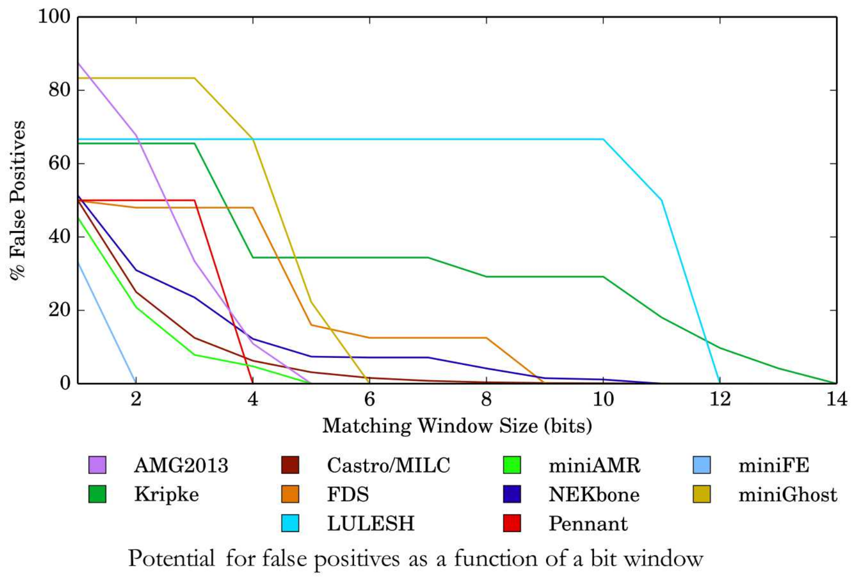
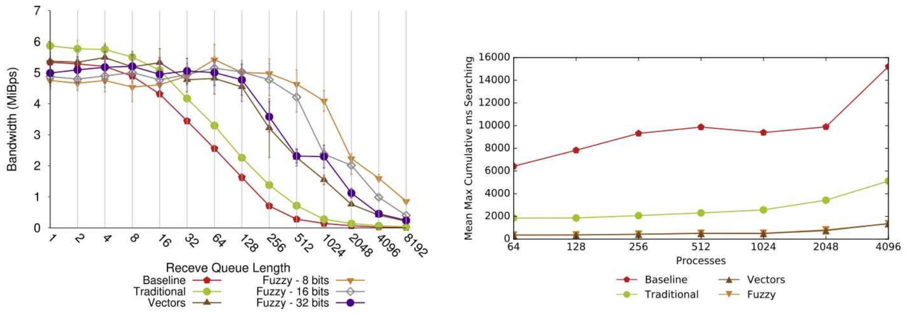


Figure 7. False positive tag matches as a function of matching window size for applications run at 1024 procs.

cost of doing so is the possibility of false positives, tags or ranks that are incorrectly identified as matches. Consequently, this approach requires a follow-up step to confirm a match identified by the vector operations is correct. The feasibility and performance of an implementation of this approach is dependent on the number of false positives.

To assess false positives, we considered a variety of applications and mini-applications at different scales. We recorded the ranks of sender processes issuing point-to-point sends, the rank of the destination process, and the tags used in those sends. We found most of the applications surveyed had tag spaces that were either zero (i.e., none of the inter-process communication actually required a tag) or scale-invariant (i.e., the number of distinct tags did not grow as scale increased). Figure 7 shows false positives for the worst-performing rank some of these applications run at 1024 procs. Eight bits is sufficient for half to have perfect tag discrimination, and eight still remain below 5% possible false positives.

Figure 8 shows the performance impacts of Vector and Fuzzy Matching. Figure 8(a) shows the impact of these approaches on bandwidth on a Intel Sky Lake processor. This figure shows a significant performance improvement as we increase the level of vector parallelism, where using an 8-bit element vector allows us to do 64 matches in parallel. Figure 8(b) shows the time spent matching in AMG2013. This metric is gathered by averaging the result for the bottle-neck process across the run. This allows us to evaluate application impact, as laggard processes are often determinant of application performance. Both of these compare against the Open MPI default matching engine and a lightweight single linked list.



(a) Bandwidth for small (1B) messages.

(b) Mean maximum cumulative time spent searching, AMG2013.

Figure 8. Performance Analysis of Vector and Fuzzy Matching

4. Managing Memory

Another strategy for increasing the efficiency of message matching is to utilize memory in more intelligent ways, e.g. minimizing cache misses by encouraging list items to be available in the cache hierarchy [20]. To explore this possibility we examined two sorts of locality, spatial and temporal. Linked lists often span multiple non-contiguous cache lines, and are can be difficult for the CPU's pre-fetcher. With spatial locality, multiple matching elements are placed into contiguous memory. This is so that the act of fetching one matching element into cache also brings in a number that follow. In contrast, with temporal locality, the goal is to access the list periodically to prevent list elements from being evicted from cache. There are a number of ways this can be accomplished, including a dedicated network cache, pinning memory to cache, and our new technique, software hot caching.

Figure 9 demonstrates the impact that spatial locality of MPI matching data has on achievable bandwidth. The baseline is the default layout for a linked list approach to message queues. LLA-n indicates that the linked list has been aggregated together to be a contiguous memory regions that can hold *n* matching elements per list element (e.g. LLA-32 has linked list elements that hold an array of 32 match elements). It can be seen that spatial locality matters most when list lengths exceed 512 elements, with the largest amounts element counts being the most effective.

Figure 10 examines the impact of data placement in the cache hierarchy by using hot caching (HC). Hot caching is a technique that uses compute cores that share a cache level with the core performing MPI processing. The hot caching core keeps the relevant MPI match list data in cache by accessing it periodically, ensuring that it does not get evicted from cache. We can observe that the hot caching technique is complimentary to the spatial locality techniques that we previously discussed. Hot caching is preferred for short list searches, but degrades at a similar point to the baseline MPI implementation. Combining the effect with increased spacial locality, the utility of hot caching is extended to longer list searches, eventually converging with the spatial locality technique in performance.

Figure 11 illustrates the improvements that spacial and temporal aware MPI message matching can have on Fire Dynamics Simulator (FDS). The benefits can be up to 2X the baseline when the job core count exceeds 2048. This shows the application im-

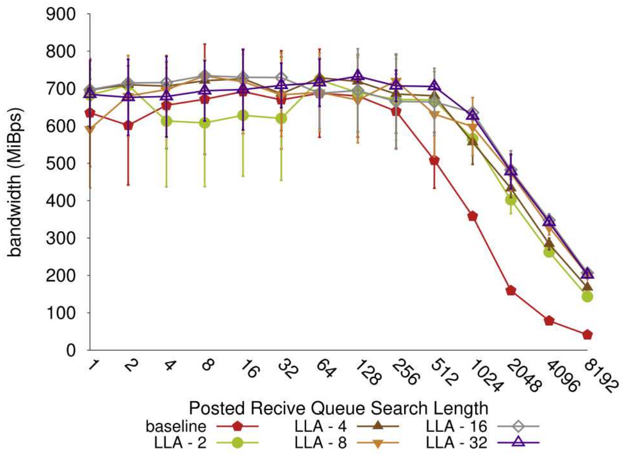


Figure 9. Spatial locality's impact on bandwidth

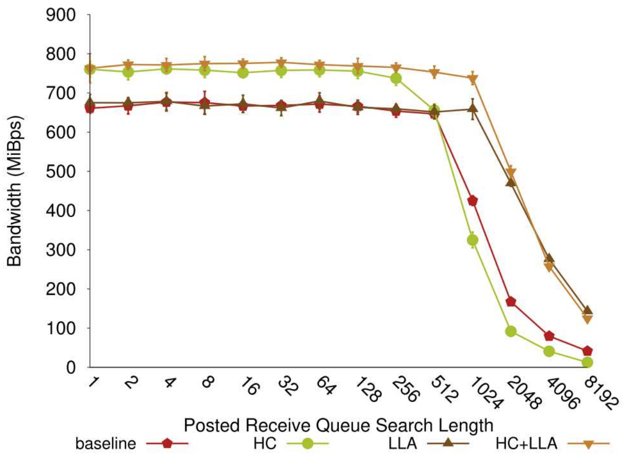


Figure 10. Temporal locality's impact on bandwidth

pact available from improving match-list locality. While we don't expect this to be representative of today's applications, FDS serves as a good proxy for future multithreaded applications. It should be noted that the HC Nehalem performance is tied to the overhead of externally managed memory. With explicitly managed memory, we can create a pool of elements that are consistently hot cached, removing the overhead of adding and removing elements. As the LLA implementation manages memory within the matching engine, HC is more effective when combined approach.

5. Changing Models

Perhaps the best way to avoid the coming storm is to adopt alternative models for message passing. For example, by allowing relaxations of current ordering constraints, avoid-

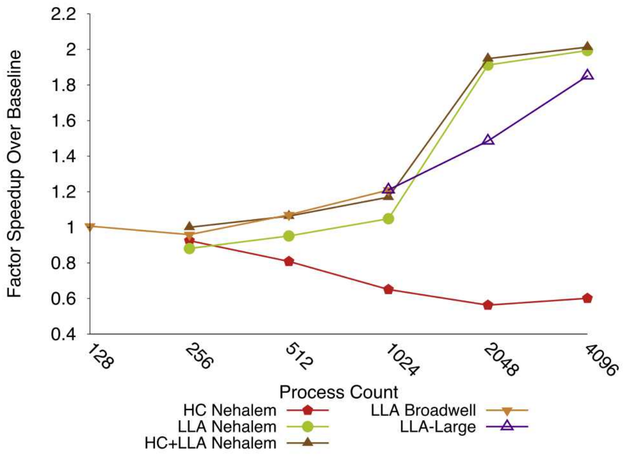


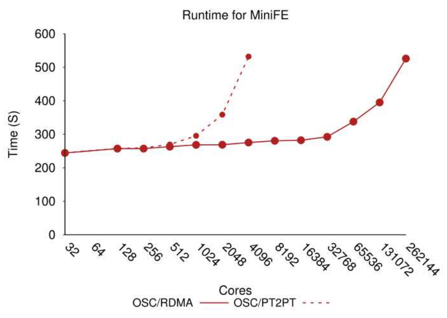
Figure 11. Locality effects on Fire Dynamics Simulator

ing traditional point-to-point communication, or finding other ways to exploit multi-threading besides promoting them to the same communicative role as traditional MPI processes. By designing new communication models, we can tailor the interface to match modern runtime environments, allowing for a holistic approach rather than creating a series of workarounds. The drawback of these approaches is they can require significant changes to applications. However, these approaches can allow for sustainable and portable multi-threaded performance. In this final section we discuss two possible alternative models, RMA and Finepoints.

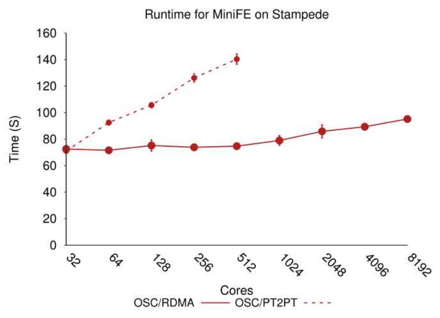
5.1. One Sided Models

RMA is MPI's one-sided communication model. Under this model, each process exposes a buffer as a window through which other processes can put, get, and accumulate data. This approach provides limited message processing, reducing message-passing overheads. However, since this technique can leverage hardware level Remote Direct Memory Access (RDMA), the out-of-band communication that occurs with RDMA must be managed to ensure that the memory performance of the application is not impacted. For some systems, this Network-induced Memory Contention (NiMC) can have significant impact on application performance [21,22].

RMA can be a challenging model to understand and use. Unlike two-sided model, message delivery is not explicit so the user must use methods to explicitly synchronize the RMA windows with the remote processes. This means that communication in RMA typically occurs in epochs. In the MPI-3 RMA model there are two classes of communication synchronization; active-target, and passive target. Active target provides global memory fence (MPI_Win_fence) and group communication (Post-Start-Complete-Wait) completion semantics. Niether of which may be well optimized in MPI implementations. For passive target, using the MPI_Win_lock and MPI_Win_lock_all APIs enable the use of unlock or the more efficient flush operations to synchronize windows. Users may see, and not totally understand, suboptimal performance when opening and closing windows frequently in applications. Additionally, RMA requires sender side knowledge of data



(a) MiniFE on KNL



(b) MiniFE on Stampede

Figure 12. RMA-MT results for MiniFE on two major computing platforms

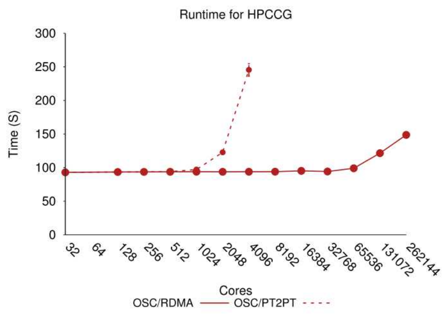


Figure 13. RMA-MT results for MiniFE on a KNL system

placement, particularly when multiple processes are interacting with the same window. MPI-3 RMA also has two memory models that an applications needs to be aware of. The unified memory model and the seperate memory model. In the seperate memory model it is necessary for the application to call MPI_Win_sync to ensure the application memory is updated to reflect changes made to the private RMA window data.

The performance of RMA in a multi-threaded environment has not been a focus of MPI library implementors in the past. This was partially due to a lack of codes that used RMA in a multi-threaded way. Recent efforts on developing codes that use multi-threaded RMA (RMA-MT) [23,24] have lead to further efforts to improve MPI library performance. These efforts have resulted in significant performance improvements to the multi-threaded support in MPI RMA implementations [25].

Figure 12 and 13 highlight performance of two mini-applications from the RMA-MT benchmark suite. These were run using the new improvements to Open MPI and were run on LANL's Trinity and TACC's Stampede. These results show a significant improvement in runtime at scale, allowing for full use of the system.

5.2. Partitioned Communication

The overhead of modifying existing applications to use alternative models like one-sided communication can be significant. Changes to alternative models can require restructuring data layouts for communication and in some cases require new algorithm design to fit the new communication model [24,25]. Therefore it is desirable to have solutions that are as close as possible to the most popular two-sided send/recv communication model. However, the existing multi-threaded interface for MPI does not differentiate between processes and threads, allowing each to interact with MPI in the same way, with the same interfaces and similar privilege.

One way to address these problems is to develop new MPI interfaces that differentiate between threads and processes. In addition, these interfaces can be aligned with existing multi-threading methods/APIs. In order to provide a two-sided communication interface that leverages existing multi-threading programming models, we have developed MPI partitioned communication [26].

In partitioned communication, threads have a specialized interface that matches multi-threading programming model semantics that also allows for greater efficiency in MPI. The model is "partitioned" in that each thread contributes a subset of the data for a two-sided communication call. This can be conceptualized as many threads working on a common data buffer, with notification to MPI on what data is available and when.

The MPI library can leverage this information to optimize data movement, moving data earlier than a traditional two-sided model can and leverage aggregation to optimize wire efficiency. There are other advantages to moving data over a longer period of time, reducing bursty traffic which in turn lowers network contention [27,28]. Partitioned communication allows for what we have termed"early-bird communication". This allows data transmission whenever data becomes available, rather than waiting for monolithic transmissions on a per process basis like traditional MPI.

However, simply performing more communication operations would present additional problems in message volume for processing at the MPI level [15]. Therefore partitioned communication provides a method of moving data whenever it is available, but provides high efficiency remote notification of completion. At the simplest level matching overheads and MPI-level message volume are reduced to single-threaded levels, with only completion of the entire buffer being promoted to the MPI level.

Early implementation of partitioned communication have shown promising results. It significantly reduces overhead compared to multithreaded communication and with efficient aggregation and early bird communication, partitioned communication can outperform thread single improving application performance.

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.

6.1. Efficient MPI Message Matching

MPI message matching has been explored in the context of evaluating its performance [29] and investigating the impact of match list length [30] and performance on different types of CPU architecture [31]. 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 [29]. Underwood and Brightwell were the first to build MPI message matching microbenchmarks to study the impact of long list length and unexpected messages [30]. 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 [31]. 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 [32]. 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 [33,34,35]

6.2. Matching Techniques

Several matching techniques have been proposed to avoid performance degradation. Dang, Snir and Gropp proposed [36] 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 [37]. 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 [38].

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 phased, a scan phase and a reduce phase that could take advantage of the large amount of concurrency available [39]. 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. 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 [40,41]

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 [42] 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 [43]. The Seastar interconnect [44] 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 [45] and the Myrinet network [46]. More recent examples of message matching NICs include the Bull-Atos BXI NIC [47] that implements the Portals interface and ConnectX-5 NICs from Mellanox that also perform message matching [48]

7. Conclusion

In this article we have shown that the non-determinism resulting from multithreading scalable system software can have a significant negative impact on application performance and highlighted several key techniques to approach this problem. Using our case study of MPI Message Matching, we demonstrated that the change in access behavior of threaded applications can create performance problems via request ordering. We then highlighted three areas of research that have been alleviating this problem. These include; leveraging parallelism through parallel progress and vector parallelism, modifying memory behavior, and building new interfaces to fully avoid these problems. Each of these solution areas have different trade-offs; Leveraging parallelism requires the least changes to the programming environment but is somewhat limited in the amount of improvement it can offer. Changing memory behaviors offers decent performance improvement but will require hardware vendors to expose caching controls. Additionally, the impact of changing the caching model on other parts of code requires a more thorough examination. Finally, changing communication models offers the most promising improvements but requires more application changes than the other approaches.

Acknowledgments

We would like to thank Kurt B. Ferreira and Kevin Pedretti for their assistance in collecting the data in Section 2.2 on the average search depth of current scientific workloads. We would also like to thank Taylor Groves for his contributions to the RMA-MT work in Section 5.1 and Patrick G. Bridges and Dorian C. Arnold for their input on many of these projects.

Sandia National Laboratories is a multi-mission 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] David E. Bernholdt, Swen Boehm, George Bosilca, Manjunath Venkata, Ryan E. Grant, Thomas Naughton, Howard Pritchard, and Geoffroy Vallee. A survey of mpi usage in the u. s. exascale computing project. *Concurrency and Computation: Practice and Experience*. in press.

[2] Kyle B Wheeler, Richard C Murphy, and Douglas Thain. Qthreads: An API for programming with millions of lightweight threads. 2008.

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

[4] Sandia National Laboratories. LAMMPS molecular dynamics simulator. `http://lammps.sandia.gov`, Apr. 10 2013.

[5] Lawrence Livermore National Laboratory. Co-design at lawrence livermore national lab : Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH). `http://codesign.llnl.gov/lulesh.php`, 2015. Retrieved 10 June 2015.

[6] Indiana University. HPCG benchmark. `http://physics.indiana.edu/~sg/milc.html`. Retrieved September 2017.

[7] J.M. McGlaun, S.L. Thompson, and M.G. Elrick. CTH: A three-dimensional shock wave physics code. *International Journal of Impact Engineering*, 10(1):351–360, 1990.

[8] Jr. E. S. Hertel, R. L. Bell, M. G. Elrick, A. V. Farnsworth, G. I. Kerley, J. M. McGlaun, S. V. PetneY, S. A. Silling, P. A. Taylor, and L. Yarrington. CTH: A software family for multi-dimensional shock physics analysis. In *Proceedings of the 19th International Symposium on Shock Waves*, 1993.

[9] Sandia National Laboratories and University of Tennessee Knoxville. MIMD lattice computation (MILC) collaboration. `http://www.hpcg-benchmark.org`, 2017. Retrieved September 2017.

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

[11] Scott Levy and Kurt B Ferreira. Using simulation to examine the effect of MPI message matching costs on application performance. In *Proceedings of the 25th European MPI Users' Group Meeting (EuroMPI)*. ACM, 2018.

[12] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. LogGOPSim - simulating large-scale applications in the LogGOPS model. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 597–604. ACM, Jun. 2010.

[13] Kurt B. Ferreira, Scott Levy, Kevin Pedretti, and Ryan E. Grant. Characterizing MPI matching via trace-based simulation. *Parallel Computing*, 77:57 – 83, 2018.

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

[15] Whit Schonbein, Matthew GF Dosanjh, Ryan E Grant, and Patrick G Bridges. Measuring multithreaded message matching misery. In *European Conference on Parallel Processing*, pages 480–491. Springer, 2018.

[16] Steve Plimpton, Paul Crozier, and Aidan Thompson. LAMMPS-large-scale atomic/molecular massively parallel simulator. *Sandia National Laboratories*, 18, 2007.

[17] Erik Lindahl, Berk Hess, Szilárd Páll, and Alfredo Metere. Gromacs 5.0 benchmarks, 2017.

[18] Matthew G. F. Dosanjh, Ryan E. Grant, Whit Schonbein, and Patrick G. Bridges. Tail queues: A multi-threaded matching architecture. *Concurrency and Computation: Practice and Experience*. in press.

[19] Matthew GF Dosanjh. Improving hpc communication library performance on modern architectures. 2017.

[20] Matthew GF Dosanjh, S Mahdieh Ghazimirsaeed, Ryan E Grant, Whit Schonbein, Michael J Levenhagen, Patrick G Bridges, and Ahmad Afsahi. 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*, page 73. ACM, 2018.

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

[22] T. L. Groves, R. E. Grant, A. Gonzales, and D. Arnold. Unraveling network-induced memory contention: Deeper insights with machine learning. *IEEE Transactions on Parallel and Distributed Systems*, 29(8):1907–1922, Aug 2018.

[23] PJ Mendygral, Nick Radcliffe, Krishna Kandalla, David Porter, Brian J O'Neill, Chris Nolting, Paul Edmon, Julius MF Donnert, and Thomas W Jones. Wombat: A scalable and high-performance astrophysical magnetohydrodynamics code. *The Astrophysical Journal Supplement Series*, 228(2):23, 2017.

[24] Matthew GF Dosanjh, Taylor Groves, Ryan E Grant, Ron Brightwell, and Patrick G Bridges. RMA-MT: A benchmark suite for assessing MPI multi-threaded RMA performance. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 550–559. IEEE, 2016.

[25] Nathan Hjelm, Matthew GF Dosanjh, Ryan E Grant, Taylor Groves, Patrick G Bridges, and Dorian Arnold. Improving MPI multi-threaded RMA communication performance. In *International Conference on Parallel Processing (ICPP)*, 2018.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] Brian W Barrett, Ron Brightwell, Ryan 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.

[32] 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.

[33] 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.

[34] R. Brightwell, S. Goudy, and K. Underwood. A preliminary analysis of the MPI queue characteristics of several applications. 2005.

[35] Ron Brightwell, Kevin Pedretti, and Kurt 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.

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

[37] 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.

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

[39] Benjamin Klenk, Holger Froning, 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.

[40] S Mahdieh Ghazimirsaeed, Seyed H Mirsadeghi, and Ahmad Afsahi. Communication-aware message matching in MPI. *Concurrency and Computation: Practice and Experience*, page e4862.

[41] S Mahdieh Ghazimirsaeed, Ryan E Grant, and Ahmad Afsahi. A dedicated message matching mechanism for collective communications. In *Proceedings of the 47th International Conference on Parallel Processing Companion*, page 26. ACM, 2018.

[42] Brian W. Barrett, Ron Brightwell, Ryan E. Grant, Scott Hemmert, Kevin Pedretti, Kyle Wheeler, Keith Underwood, Rolf Riesen, Arthur B. Maccabe, and Trammell Hudson. The portals 4.1 networking programming interface, 2017.

[43] Keith D Underwood, K Scott Hemmert, Arun 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.

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

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

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

[47] 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.

[48] Understanding MPI tag matching and rendezvous offloads (ConnectX-5). `https://community.mellanox.com/docs/DOC-2583`. Accessed: 2018-07-25.