The Case for Semi-Permanent Cache Occupancy*†

Understanding the Impact of Data Locality on Network Processing

Matthew G. F. Dosanjh
Center for Computing Research
Sandia National Laboratories
mdosanj@sandia.gov

S. Mahdieh Ghazimirsaeed
ECE Department
Queen's University, Canada
s.ghazimirsaeed@queensu.ca

Ryan E. Grant‡
Center for Computing Research
Sandia National Laboratories
regrant@sandia.gov

Whit Schonbein§
Center for Computing Research
Sandia National Laboratories
wwschon@sandia.gov

Michael J. Levenhagen
Center for Computing Research
Sandia National Laboratories
mjleven@sandia.gov

Patrick G. Bridges
Computer Science Department
University of New Mexico
bridges@cs.unm.edu

Ahmad Afsahi
ECE Department
Queen's University, Canada
ahmad.afsahi@queensu.ca

ABSTRACT

The performance critical path for MPI implementations relies on fast receive side operation, which in turn requires fast list traversal. The performance of list traversal is dependent on data-locality; whether the data is currently contained in a close-to-core cache due to its temporal locality or if its spacial locality allows for predictable pre-fetching. In this paper, we explore the effects of data locality on the MPI matching problem by examining both forms of locality. First, we explore spacial locality, by combining multiple entries into a single linked list element, we can control and modify this form of locality. Secondly, we explore temporal locality by utilizing a new technique called "hot caching", a process that creates a thread to periodically access certain data, increasing its temporal locality. In this paper, we show that by increasing data locality, we can improve MPI performance on a variety of architectures up to 4x for micro-benchmarks and up to 2x for an application.

KEYWORDS

MPI, Message Passing, Performance

*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.

†This work was supported in part by the Natural Sciences and Engineering Research Council of Canada Grant RGPIN/05389-2016. Computations were performed on the GPC supercomputer at the SciNet HPC Consortium. SciNet is funded by the Canada Foundation for Innovation under the auspices of Compute Canada, the Government of Ontario, Ontario Research Fund - Research Excellence; and the University of Toronto.

‡Also with Computer Science Department, University of New Mexico.

§Also with Computer Science Department, University of New Mexico.

ICPP'18, August 2018, Eugene, OR, USA

ACM ISBN 978-1-4503-6510-9/18/08…$15.00
https://doi.org/10.1145/3225058.3225130

ACM Reference Format:
Matthew G. F. 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 *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3225058.3225130

1 INTRODUCTION

MPI, the dominant HPC communication API for over two decades, has been used with a wide range of hardware from small clusters to leadership-class systems. As we approach exascale, the ratio of network injection bandwidth to CPU throughput has decreased. BSP applications tend to use the network in uneven ways, having large regions of compute with very low network requirements followed by regions of communication that have very bursty network behavior. To mitigate this, a number of new MPI programming models have emerged to reduce the network bottleneck. These models often utilize a fine-grained messaging scheme which increase the volume and service time of network processing requests.

In this paper, we explore the implications of data locality in network processing by focusing on a key MPI processing bottleneck, message matching. By keeping message volume low and search depths shallow, well-tuned applications can minimize the impact of message matching on overall runtime. However, less well-tuned applications can suffer from a large number of outstanding requests and long searches. Furthermore, given the movement towards multi-threaded MPI communication [6], we will show the expected growth in volumes of MPI requests and longer average search depths when using traditional MPI message matching techniques. The techniques reported in this paper demonstrate how considering locality is important to engineering an efficient message matching solution handling these various cases.

Locality has two basic forms in modern systems, namely spacial (location in main memory) and temporal localities (placement in

cache). To examine the effect of different spacial locality levels, we introduce a matching architecture as a linked list of arrays. This structure allows us to contain an arbitrary number of MPI matching elements in contiguous memory, thereby increasing the ratio of entries to cache-lines, and enabling efficient hardware cache prefetching strategies.

To explore the implications of temporal locality, we developed hot caching. Hot caching increases temporal locality by creating a heating thread which periodically interacts with specified regions of memory. By updating the metrics used in cache eviction, the specified regions are prevented from being evicted. For example, a region that would otherwise be evicted under a least-recently-used cache evicition policy is retained. We can control the granularity of the induced temporal locality by adjusting the periodicity of the heating thread and by adjusting its binding to determine which level of hierarchical memory it gets refreshed into.

The results in this paper show a substantial increase in MPI performance with increasing data locality. The results are similar to other explorations of optimizing MPI message matching, which will be discussed further in section 5. Given this, we posit that, with explicit hardware-supported data-locality control for a portion of the data cache, a cache partition, or a dedicated network cache, MPI message matching performance can be improved for long lists without a cost to short list performance.

This paper makes the following contributions:

- A linked list of arrays matching architecture to explore the effects of increased spacial locality;
- A technique to increase temporal locality named 'Hot Caching'; and,
- A study of the effects of data locality on MPI using different x86 processor architectures.

The rest of this paper is structured as follows: in Section 2 we discuss MPI required matching semantics for messages and provide an overview of existing implementations with a discussion of current application behavior regarding matching. Section 3 describes the MPI library modifications that we have created as tools to study the impacts of data locality on MPI. Next, Section 4 discusses the design of our experiments and details the results of said experiments on both microbenchmarks as well as proxy applications and a real production MPI application. Finally, Section 5 discusses related work and Section 6 concludes the paper with final discussion of our findings.

2 BACKGROUND AND MOTIVATION

MPI message matching is a key feature of the API that has helped contribute to its success over the last two decades. Having matched send/receive semantics makes programming easier and helps guarantee communication isolation through source/destination information, MPI communicators, and user-defined tags. MPI message matching has been performant in MPI implementations since MPI's inception, and as a result has not been optimized as heavily as other aspects of MPI libraries. New proposals for thread handling in MPI have brought message matching back to the forefront as the load on a single match engine is expected to increase significantly (see below).

Note this paper does not aim to provide novel, optimized matching engine schemes. Rather, the motivation is to better understand the contributions of data locality to matching performance, and hence to provide insight into the merits of existing proposals, as well as guidance for the development of new ones.

2.1 Matching Semantics

MPI matching works on three key elements: a source address, a matching tag, and a communicator. An MPI communicator is a special isolation mechanism that allows a defined set of processes to send messages to each other. Each process in a communicator has a corresponding address, called a rank. Each message a process sends must be given a matching identification number, called a tag. Each process keeps two matching lists, a posted receive queue for messages that are expected to arrive, and an unexpected message queue for messages that have been received but did not find a corresponding match in the posted receive list. The unexpected message queue is useful for cases where the receiving process is not ready for the incoming message.

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

2.2 Match List Implementations

MPI implementations take different approaches to implementing MPI matching. Implementations based on the open source MPICH implementation [23] typically use a single linked list for all communicators. Newer approaches like CH4 in MPICH, however, use more than one list [22]. Other more complex approaches like hash tables are also possible [13].

Of the open source MPI implementations, Open MPI [24] has the most complex match list, a hierarchical list with the communicator as the first level and source as the second level. Each communicator has an array of linked lists for searching the ranks and tags. In this approach, requests bearing rank i are positioned in the ith element of the array. This allows the short list for a particular communicator/source to be reached in O(1) time. Consequently, the queue search time in this approach is significantly faster than the linked list data structure. The Open MPI approach, however, is not scalable in terms of memory consumption, since for a communicator comprising N processes, each process must maintain an array of size N. This results in a total of $O(N^2)$ memory usage.

Some hardware will perform matching so that MPI does not have to. Examples of such hardware include Intel's OmniPath PSM2 [7] devices that handle matching in software layer messaging, and Atos-Bull's BXI interconnect [11] which performs MPI-style message matching entirely in hardware. Such solutions will only benefit from software MPI matching improvements when list lengths are longer than that which can be supported in hardware. Understanding the effects of data locality in such cases is still valuable, and can even inform future hardware matching designs that rely on more traditional CPUs for offloading matching.

Decomp.	Stencil	t_r	t_s	Length	Search depth
32×32	5pt	124	128	128	32.51
64×32	5pt	188	192	192	48.22
32×32	9pt	124	132	380	85.18
64×32	9pt	188	196	572	127.24
$8 \times 8 \times 4$	7pt	184	256	256	65.85
$1 \times 1 \times 128$	7pt	128	514	514	132.27
$1 \times 1 \times 256$	7pt	256	1026	1026	259.08
$8 \times 8 \times 4$	27pt	184	344	2072	410.02
$1 \times 1 \times 128$	27pt	128	1042	3074	596.85
$1 \times 1 \times 256$	27pt	256	2066	6146	1294.49

Table 1: Queue lengths and mean search depths for 2d and 3d decompositions.

2.3 Features of Common Matching Patterns

To study locality inside of an MPI implementation, we must first examine the behavior of MPI under different workloads. As our focus for this study is on the matching engine, it is important to understand how common application patterns affect list lengths, search depths, and the degree to which the matching engine impacts MPI application performance. To this end, the SST simulation system [21] contains motifs describing communication patterns for different categories of applications. SST was modified to collect queue data for three of these patterns at large scale: an adaptive mesh refinement pattern, AMR, at 64K processes; a 3D sweep pattern at 128K processes;and a Halo-exchange pattern at 256K processes.

Figure 1a shows the number of items in a given process' match list on the x-axis and the number of times that such a sample occurred throughout the simulation on the y-axis. Samples are taken during each communication phase in the simulation, such that all list additions and deletions are captured. AMR shows that most list lengths maintain zero to mid-hundreds of elements for the majority of the application run; however, extremes do occur out to the mid 400s. Therefore we conclude from these results that the most important queue lengths occur in the mid 100s as they are abundant and intensive to search. It is also important to consider the performance ramifications of several hundreds of list elements at a time to capture performance drop off with regards to longer lists.

Sweep3D patterns in Figure 1b show similar results to AMR, with the exception of the length of exceptionally long queues. Sweep3D needs good performance for queue lengths into the low hundreds of elements. Halo3D shows the expected queue lengths for a neighbor halo exchange, i.e., relatively few elements in the queue and many very small queue length operations. Consequently, applicaitions of this sort require good short list length performance.

The MPI standard permits multithreaded communication (i.e., `MPI_THREAD_MULTIPLE`). While this approach is currently uncommon in applications, it is expected to be more widely adopted as we approach exascale [6]. Since multithreaded communication increases message counts while introducing nondeterminacy through scheduling and lock contention, list lengths and search depths are anticipated to grow.

To gain insight into how multithreaded communication impacts MPI matching, we developed a benchmark emulating multithreaded MPI communication for different 2d and 3d thread decompositions. In this benchmark, a receiving MPI process is decomposed into threads, each of which posts receives during a BSP communication phase. Assuming intra-process communication is handled independently of the MPI matching engine (e.g., through shared memory), some subset of these threads will receive messages from those of similarily-decomposed neighboring processes. The number of entries added to the receiver's match list is thus a function of the number of receiving threads, the decomposition, and the type of stencil. Threads are assumed to enter the communication phase concurrently, so the order in which entries are added depends on scheduling and lock contention. A second multithreaded MPI process serves as a proxy for the multiple sending processes of an actual application. After the receives are posted, the proxy issues the required number of sends (also in a multithreaded region), and search depth information gathered on the receiving side.

The benchmark was executed on a Cray XC40 with an Aires interconnect and KNL nodes (68 cores per node, four hardware threads per core). In some cases, the sending process may be oversubscribed, but this never occurs in the receiving process. Table 1 shows results for several 2d and 3d thread decompositions. t_r is the number of threads posting receives, t_s the number of sending threads, and average search depths are averaged over 10 trials. While some list lengths stay within the bounds seen in the SST results, those for decompositions with many sending neighbors, or with communication-intense stencils, go beyond those bounds by an order of magnitude. Likewise, for 27pt stencils, average search depth easily exceeds the maximum list lengths observed in the SST results. Consequently, to cope with future matching requirements, a matching engine should also exhibit good performance for searches of lists with thousands of entries, of which hundreds to thousands must be inspected.

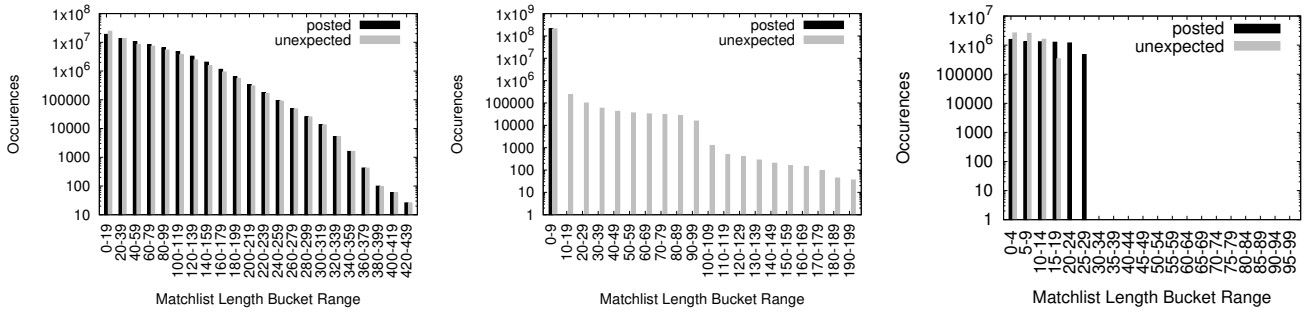
3 TOOLS TO STUDY MATCHING LOCALITY

In this section we present the two techniques that we use to explore spacial and temporal locality. Subsection 3.1 presents a linked list of arrays data structure that allows control of the degree of spacial locality in main memory. Subsection 3.2 presents hot caching, a method that facilitates the placement and retention of data in the shared CPU cache by adjusting temporal locality.

3.1 Linked List of Arrays

Our linked list of arrays structure stores an array of matching entries within a linked list element. This provides control over the degree of spacial locality within the entire list by changing the ratio of contiguous elements to non-contiguous elements. To increase the spacial locality, we add more entries to each array.

Each queue element for the posted receive queue contains 24 bytes of information, 4 bytes for the tag, 2 bytes each for the rank and context id, 8 bytes of bit masks for matching, and an 8 byte pointer to the request. The unexpected message queue does not require masks, so it only requires 16 bytes per entry. There are also 3 per array items that are stored: a pointer to the next array and indexes to the array indicating the start and end of the used section. We manage holes in the array (from deletions in the middle of the



(a) AMR match list sizes - 64K (b) Sweep3D match list sizes - 128K (c) Halo3D match list sizes - 256K

Figure 1: Queue Lengths for Common Matching Patterns

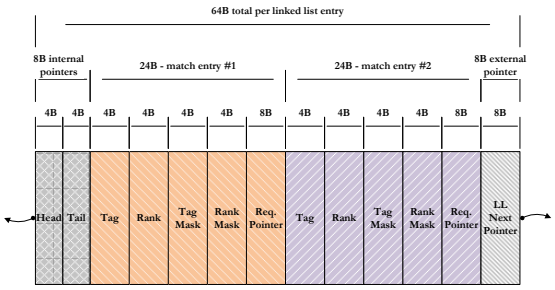


Figure 2: Packing data structures into 64 byte cache lines

list) by ensuring tags and sources are invalid and all bitmask fields are set.

For the purposes of this study, the linked list of arrays technique is tuned to optimize the efficiency of each memory lookup. To accomplish this, the first logical spacial locality increase that we have explored aims to fill a cache-line as shown in Figure 2. For readability, this figure combines rank and context id into a single 4 byte field. This results in each PRQ linked list element containing two entries and each UMQ element containing 3. From there we increase spacial locality by doubling the number of elements to perform an exponential sweep.

3.2 Hot Caching

Hot caching is a technique that leverages otherwise idle hardware to manipulate the temporal locality of specified regions of memory. This technique increases the temporal locality of a specified region of memory by periodically accessing said region. Our basic implementation is a thread that iterates through a list of regions, specified as a virtual memory address and a size, and adds the first four bytes of each cache line to a throwaway summation variable. It then sleeps for an arbitrary number of nanoseconds and repeats the process. To apply this to MPI, we add those memory regions associated with the matching engine to the list of regions for the hot caching thread.

There are a number of challenges in implementing this technique. First, when running alongside an MPI implementation it is necessary to pin the extra thread to a core that shares a level

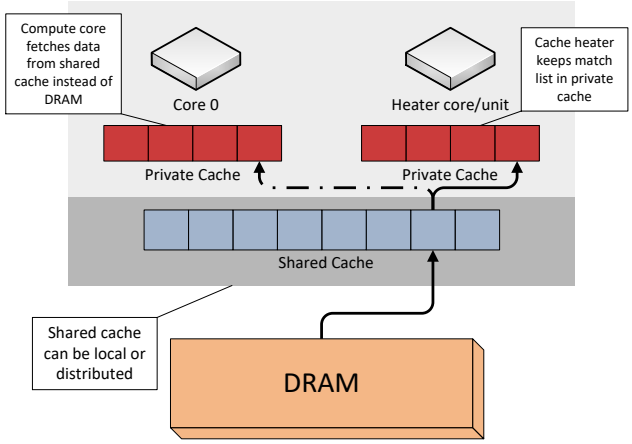


Figure 3: Hot Caching

of cache with the communication process. Intel Sandy Bridge and Broadwell processors have a shared L3 cache. This allows us to pin the thread to any other core on the socket.

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

The third challenge is reducing application interference. The hot caching thread utilizes processor resources, occupying both cycles on a core and lines in cache. This has the potential to affect the computation phases of some applications. While our initial implementation focuses on understanding locality via hot caching, there are a number of potential mitigation strategies if hot caching was used in practice. First, the heater can collaborate with the application to pause when needed. The challenge with this approach is to resume the heater in time to ensure the match list is in cache

before the first access in a communication phase; this should be easily achievable in current generation bulk synchronous algorithms, where communication phases are explicitly separated from computation. Another option assuming is to gain access to defective cores on the die that still have the potential to load data from memory into a shared cache; this would allow the heater to leverage otherwise un-utilized hardware. In such a case, we need a core that is turned off for yield purposes, that is still capable of load/store operations but might otherwise be faulty, for example a bad FPU or bad register. Alternatively, this could also be supported with relative ease by device manufacturers by adding a small 1-2KiB network specific cache to the core design.

4 EXPERIMENTAL RESULTS

In this section we present the results of our locality study. First, we describe our experimental setup. Next, we present results of an in-depth locality study on two processor architectures leveraging our modified matching engine and microbenchmarks. Next, we examine the effect of MPI Matching locality on real codes using two proxy applications. Finally, we present a study of locality on a full application.

4.1 Experimental Goals and Design

To test how spacial and temporal locality affect the MPI match engine we designed a series of experiments that explore match queue length, caching behavior, and memory subsystem behavior. The goal of these experiments is to identify aspects of locality that impact performance, as well as the degree to which different aspects of locality matter. We designed the match queue length experiments to better understand the amount of memory needed to hold all of the relevant MPI data. This helps in sizing caches and understanding requirements for layout in main memory. Our caching behavior experiments were conceived to answer two main questions, "how much does caching network data help?" and "how much could caching potentially hurt application performance?". Finally, we designed experiments to understand the impacts of memory subsystem behavior on network processing. Namely, the layout of data in memory can have positive or negative impacts on performance based on the ease with which hardware prefetching units can effectively fetch data from memory before it is needed, reducing the number of cycles wasted while stalled on pending memory loads.

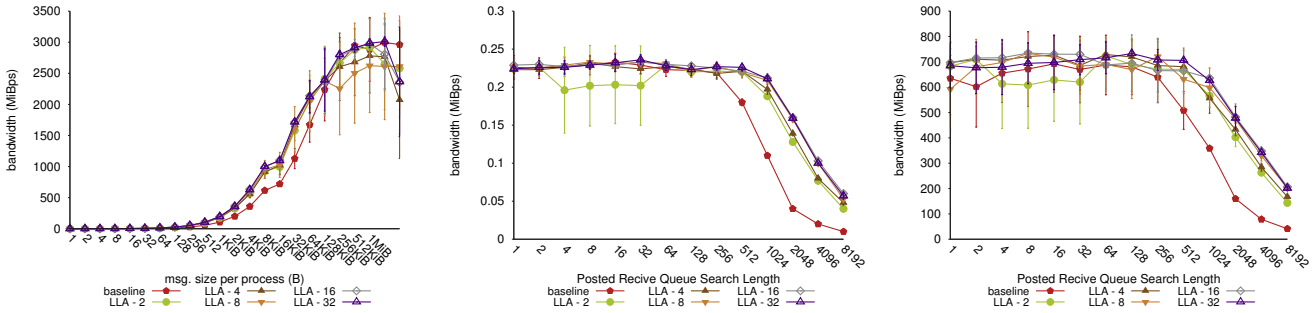
To explore this space, we identified and modified several well-known benchmarks. In particular, we tested using the OSU microbenchmarks [19] for MPI bandwidth and latency, the Mantevo mini-application [15] miniFE, the APEX benchmark AMG2013 [1], and the SPEC benchmark Fire Dynamic Simulator (FDS) [18]. The microbenchmarks were modified in four ways: First, we added an MPI barrier to ensure that recvs were preposted. This allowed us to test the fast path of the underlying MPI implementation. Second, we cleared the cache between each iteration. This emulates a computation phase in a bulk synchronous application and allows us to do fine grained performance evaluation in that context. This also allows us to tightly control temporal locality effects, as tight loop repeated communication like that in a microbenchmark will introduce beneficial temporal locality that would not be present in real applications. Third, we pinned the master thread to a specified core. This allowed for experiments utilizing shared caching behaviors on our target platforms. Finally, we added unmatched entries to the queue to evaluate performance with different receive queue lengths. The mini-apps were modified to allow different receive queue lengths to assess the impact of locality on future communication patterns utilizing finer grained messaging. We expect the use of finer grained messaging in the future for asynchronous algorithms, communication and computation overlap, and over-decomposition [4]. The majority of experiments in this section utilize two systems: A Xeon Sandy Bridge system and a Broadwell system. The Sandy Bridge system has two 2.6 GHz 8-core processors and 64 GB of RAM per node and is connected by a QLogic InfiniBand QDR network. The Broadwell system has two 2.1 GHz 18-core processors and 128 GB of RAM per node and is connected by an OmniPath EDR network. Additionally, A Xeon Nehalem system was used for larger application scaling experiments. This system was has two 2.53 GHz 4-core processors and 16 GB of RAM per node and is connected with a Mellanox QDR network. All of the micro-benchmark results in this section are presented as averages and standard deviations of 10 runs. The application results are an average and standard deviation for three runs.

4.2 Spacial Locality

To test the effects of spacial locality of the matching engine on message rate, we ran our modified OSU_bw benchmark with a variety of linked list of arrays configurations. The linked list of arrays modification to MPI leverages data placement that increases the spacial locality of sequential matching entries. We examine the effect of increasing the number of contiguous entries to determine what impact increasing spacial locality has on performance.

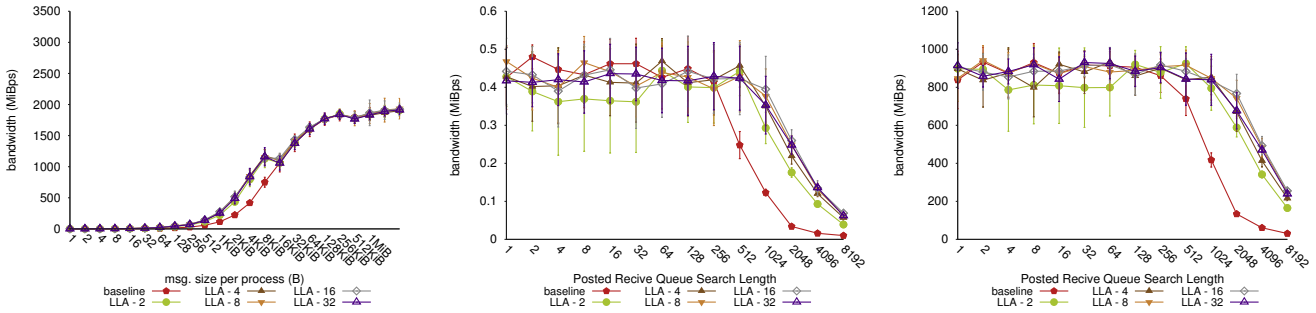
Figures 4a and 5a show bandwidth results with a 1024 queue search depth for Sandy Bridge and Broadwell respectively. On both systems, we observe up to a 2x performance benefit for small and medium message sizes by increasing the spacial locality. However, this appears to be limited for large messages and the network's data transfer speed becomes the bottleneck. These tests demonstrate that even with significant message processing compute time elapsed, spacial locality is important to performance. In this case, the traditional linked list cannot be as easily exploited to predict memory load behavior as the linked list of arrays approach. This is understandable as the traditional linked list requires information embedded in the list entries themselves for determining the next memory load address. The use of arrays in the linked list provides an easily recognizable relationship between the data, and allows for multiple entries to be fetched and prefetched at once.

Our spacial locality results are further broken out in figures 4b and 5b, which showcase the effect of data locality on small messages of 1 Byte. This allows us to explore the effects of matching engine locality as the matching engine search depth scales. We observe a large jump from the baseline to the first linked list of arrays, and a slight increase as we increase the number of entries within an array. This effect seems to be limited as the performance gain stops once we reach 8 elements per array. This is very interesting as it lines up with our experimental design regarding packing data



(a) Queue Depth 1024 (b) 1 Byte Messages (c) 4 KiB Messages

Figure 4: Impact of Spacial Locality for Sandy Bridge Architecture



(a) Queue Depth 1024 (b) 1 Byte Messages (c) 4 KiB Messages

Figure 5: Impact of Spacial Locality for Broadwell Architecture

into cache lines. Each cache line contains 2 matching elements, and both the Sandy Bridge and Broadwell architectures have four different types of prefetch units. One of the L2 level prefetch units specializes in fetching cache line pairs for adjacent data. Therefore, this observation of 8 data items being a performance peak aligns with a common prefetch unit gathering a single adjacent cache line from the original load for the array, followed by a specialized cache prefetch unit meant to fetch the adjacent cache line pair, 128 Bytes, or 4 more data items. This means in total we observe 4 cache line loads per load operation due to prefetching; which at 2 entries per cache line equates to 8 items fetched per load. Interestingly, this also carries over to the L1 cache, where prefetching occurs on a per line basis rather than pairs, but as the data is accessed in a sequential manner, the L1 prefetcher has enough time to fetch adjacent cache lines from the L2 to feed the CPU.

Figures 4c and 5c show the same experiments with more realistic 4KiB messages. While the bandwidth of the network is higher at larger message sizes, we observe similar behavior of the spatial locality test of a performance peak at 8 elements in an array.

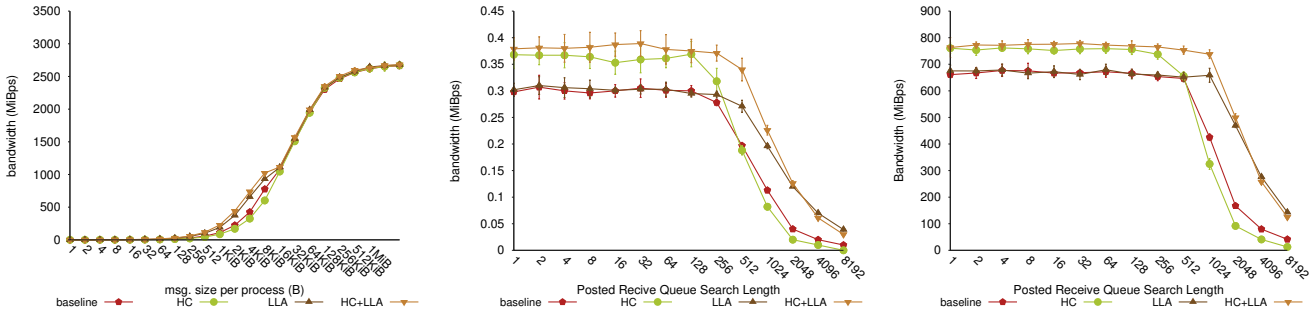
These results indicate that spacial locality matters in two ways to the matching engine. The biggest effect we observed was the difference between the unmodified baseline and the linked list of arrays posted receive queue. This is partially due to the reorganization and packing of the data. The new structure contains all the information needed to represent to matching entries in a single cache line, while the unmodified baseline requires more than a cache line for a single entry. The other part of the performance change we observed was correlated with the increase of contiguous elements, that was amenable to cache prefetch behavior.

4.3 Temporal Locality

To test the effects of temporal locality of the matching engine on message rate, we ran our modified OSU_bw benchmark with another set of modified MPI implementations.[1] These tests included one that used our cache heater technique and a combination of our LLA structure with a cache heater. The modified LLA structure allowed us to run experiments where we have more control over memory allocation. This tighter control allows us to reduce the locking overhead of our cache heater by using a dedicated element pool.

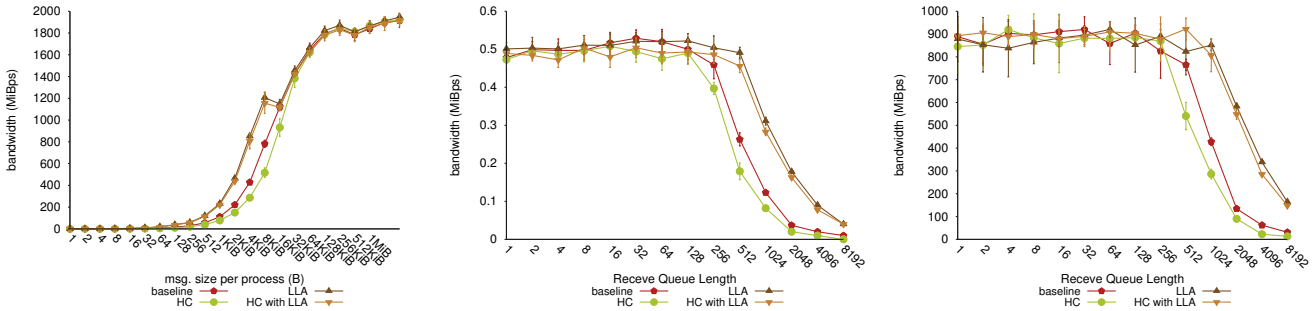
Figures 6a and 7a show bandwidth results with a 1024 queue search depth for Sandy Bridge and Broadwell respectively. On Sandy

[1] It should be noted that because these were run as separate experiments on a production cluster, there were differences in the performance of the unmodified baseline in contrast to the spacial locality experiments. These were run at different times and while there were some variance in our experimental set up (the results in the spacial locality section used the Intel 2018 compilers while the ones in the temporal locality section used Intel 2016) we have verified that these didn't significantly impact the results. The variance in results is inline with what we expect for our systems based on differences in system load and placement during these two days and each graph is presented with its own baselines to ensure a fair comparison.



(a) Queue Depth 1024 (b) 1 Byte Messages (c) 4 KiB Messages

Figure 6: Impact of Temporal Locality on Sandy Bridge Architecture



(a) Queue Depth 1024 (b) 1 Byte Messages (c) 4 KiB Messages

Figure 7: Impact of Temporal Locality on the Broadwell Architecture

Bridge, we observed an increase in performance corresponding to increased temporal locality, particularly when we avoid the synchronization overhead required by using cache heating with the original matching implementation. However, as with spacial locality, impact on large messages is limited as the network's data transfer speed becomes the bottleneck. For Broadwell, we see a negative result from cache heating, indicating that the cache refreshing is interfering with normal operation. Interestingly, there are some differences in cache architecture between Sandy Bridge and Broadwell that can possibly account for these differences. Sandy Bridge L3 cache ran at the speed of the core, the clock domains were unified, while in Broadwell (the change happened at the Broadwell predecessor, Haswell) the cache clock speed was decoupled from the core speed, which increased L3 cache latency. Cache bandwidth was significantly increased for Haswell/Broadwell, but MPI matching is inherently latency bound.

The experimental results are detailed in Figures 6b and 7b, which showcase the effect of temporal data locality on small messages of 1 Byte. In contrast to spacial locality, we see a performance increase from small to medium list lengths on Sandy Bridge. This indicates that there an increased temporal locality allows the system avoids significant overhead when using the match lists. As with the previous results, we observe a slight performance drop on Broadwell. Figures 6c and 7c show the same experiments with more realistic 4KiB messages, the results are similar to the other results.

The Sandy Bridge results indicate that temporal locality matters when entering the matching engine. As the impact can be seen in very short queues, it is likely that the increase in performance is due to avoiding cache misses that the prefetcher has trouble predicting. This is indicated by the convergence of the cache heating results with their baselines at large queue lengths.

This is not replicated in the Broadwell results. To explore this difference, we used a custom cache heater benchmark. When we run a simple cache heating benchmark on Broadwell with a random access pattern, we observe nearly a doubling of throughput (reducing the iteration runtime from 38.5 ns to 22.8 ns) which is similar to the Sandy Bridge results (which reduce 47.5 ns to 22.9 ns). Therefore, we would expect similar performance improvements to those observed for Sandy Bridge, but these latencies are for random accesses, which cannot be easily helped by prefetching. Therefore, the lack of performance gain from temporal locality on Broadwell is a result of the access pattern, higher latency lowest level cache, and overheads from the cache heating technique itself (cache heating requires holding a lock when removing elements from the list).

4.4 Application Study

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

times, and strong-scaling applications require sufficient scale before communication becomes the dominant runtime component. This work examines AMG2013, MiniFE, and MiniMD, all of which are common proxy apps for codes of interest at large capability class installations. Prior to examining these results one should note that matching is not a significant part of the runtime for today's highly tuned extreme scalable applications as many applications are built to avoid long matching lists [12]. We expect this to change as applications developers move to leverage MPI from multiple threads at exascale [6] and this will result in longer queues and less deterministic search depths (section 2.3). To this end, we present one additional application, FDS, that due to its design, utilizes the matching engine in way similar to that MPI will need to support to enable this application behavior. The application study focuses on one platform for miniapps, our Broadwell system. For FDS we run the application on an older CPU architecture, Nehalem, as exclusive access to a large more recent system was unavailable. All of these experiments leverage the first level of increased spacial locality, containing two PRQ entries or 3 UMQ entries in a linked list element. All mini-apps were run 10 separate times with many compute iterations per run. Error bars are provided for minimum and maximum runtimes observed, although some may not be visible in all figures as the runtime variance observed was low.

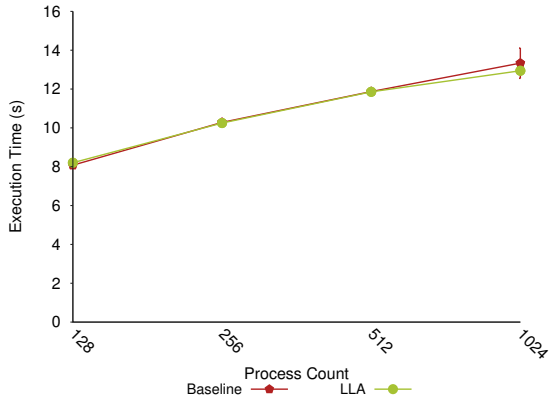


Figure 8: AMG2013 Scaling Results for Broadwell

4.4.1 AMG2013. AMG2013 is an adaptive multi-grid solver that is designed for unstructured grids. AMG is a weak-scaling code that has relatively trivial load balancing in that the grid is evenly split between all processes and increasing scale corresponds with proportionally larger problems/grids. AMG is very memory intensive and requires occasional large message bandwidth. While AMG can be forced to send many small messages if configured in an unrealistic manner, we have used the configuration recommended by the US DOE when AMG has been used for acceptance testing on new systems. Therefore, AMG is more bandwidth sensitive than message rate sensitive.

The results in Figure 8 shows scaling results from using the recommended large problem. These runs are not of large enough scale to show any clear trends for run-time, but show runtime improvements for with increased spacial locality at 2.9%. This is unsurprising as the results from microbenchmarks on Broadwell indicate that queue sizes need to be close to 512 to see large benefits. It is still important to note, however, that with regards to spacial and temporal locality impacts, good performing MPI matching proposals show similar improvements of single percentage points in runtimes for applications due to MPI. Therefore, the impacts of data locality are within the expected performance enhancement range when compared to existing practices.

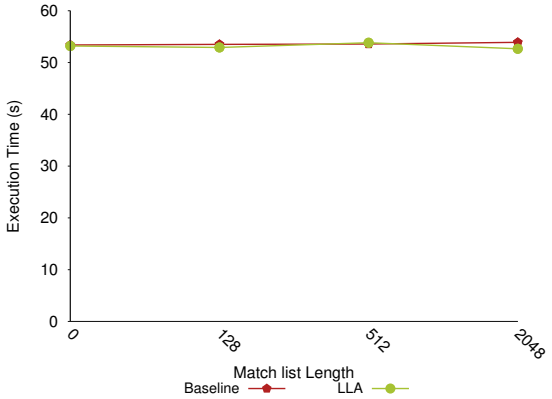


Figure 9: MiniFE Results at 512 processes with varying match list lengths for Broadwell

4.4.2 MiniFE. To explore any potential effects locality might have on optimized large scale codes, we examined two mini-applications in the Mantevo suite [15]. MiniFE is an unstructured implicit finite elements simulation mini-application that's primary computation is a conjugate gradient solver. This mini-application is representative of the common bulk-synchronous halo-exchange communication pattern.

The results in Figure 9 show MiniFE on a Broadwell system at a fixed size of 512 processes and a problem size of 1320^3, for various posted receive queue lengths. The results here are similar to the AMG2013 results, in that there is a small but not insignificant improvement from improving data locality. Using LLA at 2048 queue sizes results in a 2.3% improvement to runtime. This experiment does not show much effect from locality. This is due to the communication pattern requiring a limited number and frequency of messages with a relatively predictable ordering allowing for optimizations to reduce search depth.

4.5 Full Application Study

Typical MPI message matching lists have been shown to be in the hundreds of matching list elements for many application communication patterns, however some applications can build list sizes to much larger lengths. In addition, the manner in which these potentially large lists are searched can lead to further performance bottlenecks in MPI. One application that exhibits these behaviors is the Fire Dynamics Simulator (FDS) [18], a benchmark in the SPEC MPI benchmarking suite.

Figure 10 shows results from three different tests: testing spacial locality on Broadwell, spacial and temporal locality on a Nehalem

cluster with additional experiments to examine the limit of spacial locality on a Nehalem cluster. Each result is presented as a factor improvement compared to its baseline. For Broadwell, we ran from 128 to 1024 processes, and see a marked performance increase at 1024 with 1.21x for linked list of arrays. To examine this at larger scale, we ran on a Nehalem cluster. The results on this cluster show the effects of both spacial and temporal locality. Examining spacial locality we see a further divergence as we scale up resulting in a 2x speedup at 4k processes. While we see a slowdown from temporal locality with hot caching, temporal and spacial locality when combined shows a significant speedup at smaller scale. This is due to lock contention as we must remove elements from the hot caching list before MPI can deallocate them. Hot Caching shows a significant improvement at smaller scales. At 1024 processes, linked list of arrays with hot caching performs 14.5% better than the baseline and has a 10.4% improvement over the linked list of arrays alone. To further optimize at scale, we examined an early linked list of large arrays approach using MVAPICH2 2.0. We can observe that the large arrays results speed up FDS at 8192 MPI processes, where we observe a 2x speedup.

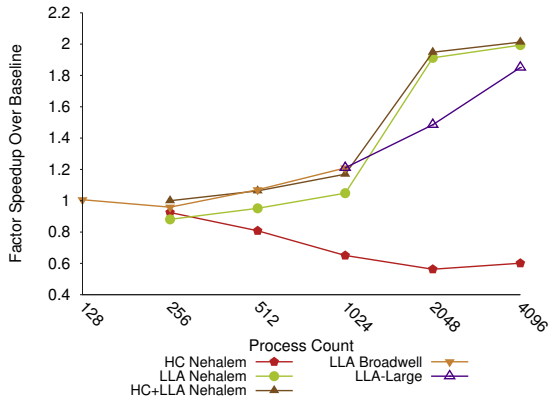


Figure 10: Fire Dynamics Simulator Scaling Results for Nehalem

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

4.6 Discussion

In this section we examined two types of data locality. From an architectural stand point, the reason to increase these forms of locality is to reduce the number of cache misses on modern CPUs. Spacial locality tries to reduce the number of cache misses by leveraging contiguous memory and data packing to reduce the number of cache lines needed and to interact better with the prefetcher. Our technique to modify temporal locality focuses on keeping data near the core in the cache hierarchy by manipulating the cache eviction policy. We observed that the spacial and temporal locality tests highlighted key architectural features that are of interest to matching engines, the depth and behavior of prefetching and the latencies and cache coherency policies in multiple x86 CPU generations.

The results in this section show that problems like MPI matching are often data locality problems. Because our MPI speedup is similar to the proposed techniques of current research, we posit that the bottleneck in MPI matching is memory look-up speed rather than computational speed. While most of the new matching techniques focus on reducing the number of comparisons needed to find an element, what these techniques are actually doing is reducing cache misses by limiting list iteration.

Given these results, we propose that CPU support for networks processing can help solve problems like MPI matching. Through allowing users to either interact with cache management or providing a dedicated networks cache, CPU developers could alleviate these problems. In addition, these caches could include custom prefetching units that can be used by middleware such as MPI to ensure consistent intergenerational performance.

5 RELATED WORK

The most related work to ours in the MPI community centers on techniques proposed for message matching, and the study of MPI message matching behavior. Our work differs from all of these previous works in that we are providing insight into how and why these approaches work and how they relate to the underlying memory subsystems that they seek to optimize. Our work does not propose a brand new message matching scheme, but instead provides the tools to better understand how to assess existing schemes and engineer better schemes in the future.

Several papers have examined MPI message matching performance [2, 25, 27]. For example, Barrett, et al. [2] examines many-core matching rates. Underwood and Brightwell [25] introduce some microbenchmarks with the aim of analyzing the behavior of MPI implementation in presence of long priority (PRQ) and unexpected (UMQ) message queues. Rodrigues et al. [20] explored the use of wide ALU operations to process multiple MPI match operations in a single ALU operation. Vetter and Yoo [27] analyze the scalability and performance characteristics of some scientific applications and show that communication overhead is one of the important limiting factor in some MPI applications.

Other studies have focused on evaluating message match list lengths for various applications [8–10, 16]. For example, Keller and Graham [16] study the characteristics of UMQ of three MPI applications (GTC, LSMS and S3D). These characteristics include the size of UMQ, the required time for searching the UMQ and the length of time such messages spend in these queues. The evaluation results show that message queue matching is one of the scalability bottlenecks for some applications. Others [8–10] have investigated

the impact of latency and message queue length on the performance of some specific applications.

Zounmevo and Afsahi [28] proposed a new message queue mechanism called a 4-dimensional data structure for large scale applications that is scalable in terms of both speed and memory consumption. This approach decomposes ranks to multiple dimensions to reduce the number of MPI queue operations. The main goal of this data structure is to skip portions of the match list for where no match can be found. This data structure decomposes ranks into a 4D lookup.

Flajslik, et al. [13] propose a data structure that uses a hash-map keyed using the full set of matching criteria. In this approach, the match lists are replaced by a fixed hash map that maps matching data to separate linked lists. The number of linked lists and the hash function are configurable parameters. The evaluation results comparing this data structure with the linked list data structure show that the proposed design with 256 bins reduce the number of match attempts per message significantly. Moreover, this data structure has a constant overhead in queue selection, which slows down the most common case of a very short list traversal. Bayatpour, et al. [5] extend the hash-table approach by creating a dynamic runtime approach to swap between hashing and traditional matching when appropriate.

In contrast to the message matching approaches that have been designed based on CPU architecture, Klenk, et al. [17] introduce a new message matching algorithm taking advantage of GPU features. This algorithm is designed based on two phases (scan and reduce) leveraging a large number of threads on GPUs. The evaluation results show that the proposed algorithm could provide 10x to 80x improvement in matching rate. The caveat with this approach is that it is dependent on the existence of an GPU style accelerator card and being able to do enough matches to mitigate the overhead of transferring control to the GPU.

The state of the art proposed software matching techniques focus on reducing search depth. With the insights of data locality presented in this paper, these techniques can be further refined to further optimize their interactions with memory behaviors. As of the writing of this paper these software techniques [5, 13, 17, 28] are closed source and unavailable for testing.

MPI matching list improvements have been proposed for hardware, with the Portals networking API [3] enabling such offloads, and investigations into hardware designs that can efficiently perform MPI message matching with wildcarding [26]. MPICH implementations are in the process of adding a new channel CH4 in order to help address scalability issues and better handle hardware supported message matching [14].

6 CONCLUSIONS AND FUTURE WORK

In this paper we have designed, implemented and used two new tools for understanding the role of data locality in the performance of MPI message matching. We have studied both data locality effects in the main memory subsystem as well as exploring the impacts of the cache memory hierarchy and the temporal relationship therein.

We have shown the impact of data locality on modern applications at different scales and have explored a full real-world application case in which it is possible to achieve large application time speedup through informed data locality design. In addition, we explored the underlying architectural reason behind this performance increase, providing insight into the benefits that new matching engine design can exploit as well as providing new information on which to base future designs. We demonstrated the speedups in matching performance that are possible and can typically produce 2X-5X speedups for common message sizes. This emphasizes how important data locality is in the design of MPI message matching schemes, and demonstrates that the tools that we have designed and implemented in this work can be of use to MPI implementors.

REFERENCES

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

[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 Network Programming Interface. Technical Report SAND2017-3825.

[4] Richard F Barrett, Dylan T Stark, Courtenay T Vaughan, Ryan E Grant, Stephen L Olivier, and Kevin T Pedretti. 2015. 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*. ACM, 30–39.

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

[6] 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, in press.* ([n. d.]). in press.

[7] Mark S. Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovettt, Todd Rimmer, Keith D. Underwood, and Robert C. Zak. 2015. Intel Omni-Path Architecture Enabling Scalable, High Performance Fabrics. *Annual Symposium on High-Performance Interconnects (HOTI)* (2015), 1–9. http://ieeexplore.ieee.org/abstract/document/7312660/

[8] R. Brightwell, S. Goudy, and K. Underwood. 2005. A Preliminary Analysis of the MPI Queue Characteristics of Several Applications. In *International Conference on Parallel Processing, 2005. ICPP 2005.* IEEE, 175–183.

[9] 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/

[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] 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

[12] 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.

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

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

[15] 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. 2009. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574* 3 (2009).

[16] 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.

[17] 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.

[18] 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.

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

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

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

[22] MPICH Development Team. 2016. CH4. https://www.mpich.org/2016/11/13/mpich-3-3a2-released Last accessed: 1/4/2017.

[23] MPICH Development Team. 2017. MPICH. https://www.mpich.org Last accessed: 30/3/2017.

[24] OPENMPI Development Team. 2017. OPENMPI. https://www.open-mpi.org Last accessed: 28/3/2017.

[25] 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.

[26] 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.

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

[28] 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.