2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)

Fuzzy Matching: Hardware Accelerated MPI Communication Middleware

Matthew G. F. Dosanjh, Whit Schonbein and Ryan E. Grant
Sandia National Laboratories
Center for Computational Research
P.O. Box 5800, MS-1110
Albuquerque, New Mexico 30332–0250
Email: (mdosanj,wwschon,regrant)@sandia.gov

Patrick G. Bridges
University of New Mexico
Department of Computer Science
Albuquerque, New Mexico 87131
Email: bridges@cs.unm.edu

S. Mahdieh Gazimirsaeed and Ahmad Afsahi
Queen's University
Electrical and Computer Engineering
Kingston, Ontario K7L 3N6
Email: (s.ghazimirsaeed,ahmad.afsahi)@queensu.ca

***Abstract*— Contemporary parallel scientific codes often rely on message passing for inter-process communication. However, inefficient coding practices or multithreading (e.g., via `MPI_THREAD_MULTIPLE`) can severely stress the underlying message processing infrastructure, resulting in potentially unacceptable impacts on application performance. In this article, we propose and evaluate a novel method for addressing this issue: 'Fuzzy Matching'. This approach has two components. First, it exploits the fact most server-class CPUs include vector operations to parallelize message matching. Second, based on a survey of point-to-point communication patterns in representative scientific applications, the method further increases parallelization by allowing matches based on 'partial truth', i.e., by identifying *probable* rather than exact matches. We evaluate the impact of this approach on memory usage and performance on Knight's Landing and Skylake processors. At scale (262,144 Intel Xeon Phi cores), the method shows up to 1.13 GiB of memory savings per node in the MPI library, and improvement in matching time of 95.9%; smaller-scale runs show run-time improvements of up to 31.0% for full applications, and up to 6.1% for optimized proxy applications.**

I. Introduction

MPI message matching is a critical performance component in the MPI communication library. Increased message counts from multiple threads and fine-grained tasking models create difficult challenges for MPI message matching engines. In particular, communication by large numbers of threads dramatically increases the number of messages that need to be matched, as well as the length of the match lists. Most importantly, multi-threaded communication with MPI creates a partially non-deterministic match list, and therefore, the queue depth at which matches will be found for incoming messages will grow significantly [1]. This increases MPI memory usage and time spent in critical sections in the MPI library, stealing critical resources from applications.

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

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

To address these issues, we introduce two novel techniques that leverage SIMD parallel instructions to improve matching performance on many-core and traditional big-core architectures: vector matching and Fuzzy Matching. Vector matching reorganizes matching data – rank and tag – into individual vectors. This allows an implementation to quickly iterate through the matching engine to find the result. Fuzzy Matching extends vector matching by using a compressed version of the matching data to create a small 'fast-match' identifier utilizing small bit-width instructions. This results in greater search parallelism but can introduce false positives. Fortunately, false positives can be verified with minimal overhead for the continuing list search. Furthermore, for 10 different important applications and mini applications spanning multiple scientific domains, we show that with well-chosen tags and communication patterns, false positives can be eliminated.

The Intel Xeon Phi introduced much longer vector operations than its contemporary Xeon server CPU brethren. While the Xeon Phi line has been discontinued, it is still in operation in many major supercomputers throughout the world. This allows us to explore the benefits of using cutting edge long vector units at scale for our vector matching solutions. Recent Intel Xeon Skylake CPUs have added support that enables a solution that is even more beneficial than the Xeon Phi vector units. We have evaluated our approach on a Skylake system to assess these new capabilities as well.

In this paper we make the following contributions:

- A vector-based MPI message matching engine for Open MPI that provides high-performance with significant memory savings.
- A study of MPI application rank and tag use that shows the viability of using a high-pass filter on matching data.
- Fuzzy matching, an optimistic matching approach based on rank and tag space compression and that accelerates vector matching. This is demonstrated at large scale (256K cores).
- Initial results showing the benefit of new vector operations for fuzzy matching on Intel Xeon Skylake CPUs.

The rest of the paper is structured as follows: Section II

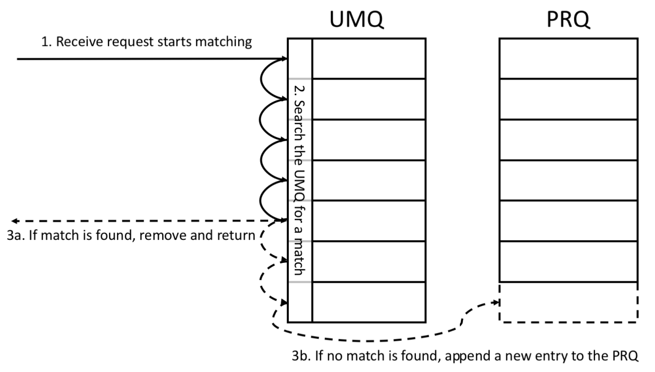

UMQ
PRQ
1. Receive request starts matching
2. Search the UMQ for a match
3a. If match is found, remove and return
3b. If no match is found, append a new entry to the PRQ

Fig. 1: Traditional Matching in MPI

describes the background of matching and the challenges associated with providing a performant implementation. Section III presents an analysis of the matching problem and the future issues that it poses for multithreaded codes. Section IV presents our matching techniques and the details of the implementations. Section V explores the viability of Fuzzy Matching by examining the rank and tag usage of different applications. Section VI presents a performance study of vector matching and Fuzzy Matching. Section VII presents additional discussion of this work. Section VIII presents the related work. Finally, Section IX presents our conclusions.

II. Background

MPI message matching has been a feature of MPI since its inception. Message matching semantics are part of the most popular functions in MPI and have been used in many code over the past two decades. However, message matching is not without challenges.

A. MPI Matching Constraints

The message matching engine is one of the major synchronous data structures in MPI. It is used to match incoming data to a user buffer specified in a receive function call. This matching is done by comparing two identifiers, a user defined tag, and the identifier of the sender (rank). MPI matching is complex because it must both ensure sequential matching order for sends from a single process, and it must support wild-card receives. For example, AMG2013 [2] uses wild-cards to establish communication outside of its regular halo exchange. There have been proposals that would allow communicators to be created without the wild-card constraint, as it limits optimizations for applications that do not use the feature.

Traditionally, major open-source MPI implementations have used a pair of linked lists to implement matching: a posted receive queue (PRQ) and an unexpected message queue (UMQ). Figure 1 illustrates the process. To post a receive, the process must search through the UMQ. If a matching message is found, the receive is fulfilled, the message is removed from the list, and the matching engine returns. If a matching message is not found, the process creates and appends a new element to the PRQ. An incoming message follows the same process but searches through the PRQ and appends to the UMQ. Due to the complexity of guaranteeing order and wild card usage, most multi-threaded implementations treat the matching engine as a singe large critical section.

B. Messaging in Exascale Applications

New programming models and algorithms have been proposed to decrease networking bottlenecks, especially as projections show parallel network access and data movement increasing. Many of these approaches rely on fine-grained messaging techniques that leverage many smaller non-blocking messages to distribute messages in non-bulk synchronous ways. For example, communication and computation overlap proposes to use fine grained messaging to distribute the communication to BSP computation phases where the network has traditionally been idle. Similarly, asynchronous algorithms and task-based approaches attempt to increase communication-computation overlap by decomposing computation into tasks that can be executed independently.

These changes reduce or eliminate the need to transmit data during synchronous application communication phases, but create longer matching lists and more matching events because there are more messages in flight at a given time. This is straightforward to handle in single threaded MPI programs as programmers can provide some determinism with regards to received message ordering. However, threaded applications are more complex and message ordering may change between individual runs due to lock contention and performance variation. Threaded applications leveraging these models will require a thread safe way to access MPI, such as $MPI\_THREAD\_MULTIPLE$.

C. Matching in Exascale Architectures

Due to their limited cache size and out-of-order processing many-core processors make MPI matching non-performant by making cache misses more frequent and expensive [3], [4]. This is problematic for BSPs as the communication phase performance is degraded by these caching effects. While other application models may be less latency sensitive they require a performant $MPI\_THREAD\_MULTIPLE$ implementation. Even with fine grained locks in MPI, large critical sections, like the matching engine, would still negatively impact performance through blocking and lock contention. To adapt MPI to these new architecture paradigms, MPI must be able to handle requests quickly for larger volumes of much deeper list searches than it does today.

D. Vector instructions

Modern processors rely on vector instructions to process large amount of data quickly and efficiently. Intel's Advanced Vector Instructions (AVX), for example, are an extension of the x86 architecture. These were first supported on the Sandy Bridge architectures, with the Intel Xeon Phi architecture introducing AVX-512, a 512-bit version of AVX instructions. Similarly, ARM's Scalable Vector Extensions are vector processing extensions to the AArch64 architecture. While this architecture is early in its life-cycle, it proposes the ability to incorporate up to 2048-bit vectors. This would allow for

64 32-bit integers to be simultaneously evaluated. As with the AVX-512 Bytes Words module, SVE allows for different subdivisions of the vectors from 8-bit to 128-bit.

III. Does MPI Matching Matter?

While modern applications are optimized for shallow MPI message matching search depths [5], developers of scientific applications targeted at exascale systems have indicated that they anticipate taking advantage of send/recv semantics in multi-threaded communication patterns [6]. To explore this we leveraged an existing multithreaded benchmark that explores the effect of multithreaded communication patterns on MPI message matching [1].

Our experiments with this benchmark can be seen in Figure 2. We ran these experiments for naïve decompositions of 9pt and 27pt stencils in addition to a 512 message queue representing an adaptive mesh refinement code. Average search depths rapidly increase into the hundreds as thread counts grow. More importantly, the average time to drain the queue represents the impact of multi-threaded matching on a single compute-plus-communicate iteration in an application. This quickly becomes problematic as many applications, such as Molecular Dynamics simulations [7], [8], have targets of running 10 to 1,000 iterations per second. This means that any queue drain time above $10^3\mu$s in Figure 2 is completely unsuitable for an entire class of scientific applications, all due to the overhead of MPI matching.

IV. Matching Techniques

To address matching issues in many core architectures, we designed a vector matching approach that leverages the AVX-512 vector unit on Intel's Knight's Landing architecture. This approach, presented in Section IV-A, increases the performance of matching significantly. Furthermore, next-generation architectures extend vector support by providing smaller bit-width integer elements, allowing applications to gain additional SIMD parallelism. Section IV-B presents Fuzzy Matching, our optimistic matching approach that leverages these additional features to increase matching performance. To test these architectures, we implemented both vector matching and Fuzzy Matching within Open MPI.

A. Vector Matching

To take advantage of vector instructions we developed a vector matching engine. This approach creates an auxiliary data structure containing separate vectors for both rank and tag, where each set of vectors is stored in a linked list element. These vectors are then loaded into the vector unit and compared via a parallel "integer-compare-equals" operation. The outcome is a bit-mask where each bit represents the result at a different offset. Once the compare is done, we check the mask to see if any bits returned true.

One challenge of this auxiliary data structure is that removing an entry from the queue can leave holes in the queue. For this initial implementation, we simply ignore empty elements by setting their tag and rank entries in the vectors to the unmatchable default values. An element is removed once it does not contain any matching entries. This maps well to many common matching patterns, where the queue is cleared between iterations of an application. This could be problematic in the corner case where there are a number of long-lived entries that are not co-located in the array. However, all of our results include the inefficiency of the "holes" and for all of the applications studied this is not a significant factor in performance. There are a number of ways "holes" can be handled in the future, such as a clean-up step aided by new SIMD operations, such as a vector shift, which we plan to explore in future work leveraging FPGAs.

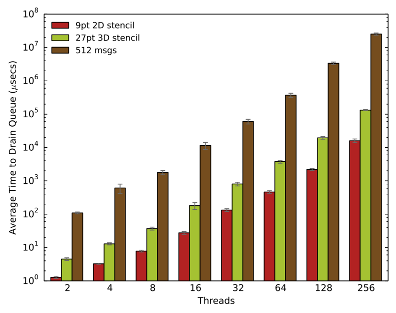
To search the posted receive queue in this structure, we use the `set1` function to initialize target vectors for both rank and tag. These target vectors are used for the entire search. To support wild-cards, the implementation then applies a vector bit-mask using a vector bitwise `and` operation. Finally, the implementation uses a vector equality operation to test for a match. This gives us a 16 field bit-mask of match information.

B. Fuzzy Matching

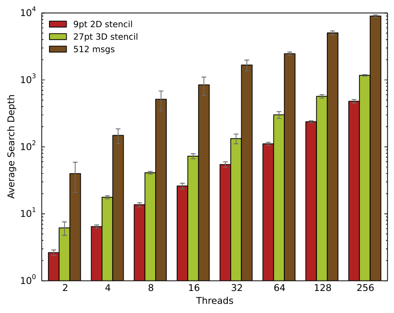
Fuzzy Matching is an approach that leverages smaller vector elements to increase the parallelism of our vector matching architecture. Applications rarely need 32 bit integers for tag and rank space as these support over 4 billion different values.To develop this concept, we designed an optimistic matching architecture. This design creates a fast match value that combines tag and rank into a single number. For the initial implementation, the fast match number uses a window of the least significant bits of each, and combines them into a single integer by offsetting the windowed tag bits to avoid overlap. This bit-wise operation is: $fast\_match_x = ((tag_x \& tag\_window\_bitmask) << rank\_window\_size) | (rank_x \& rank\_window\_bitmask)$
There are other ways to generate a fast match id such as selectively choosing bits, or using a map structure to apply different heuristics, which we will explore in future work.

This allows us to do many comparisons in parallel, but comes with the caveat of having to verify positive results to prevent false positives. This exposes a trade-off space based on the size of the tag window and rank window; we can reduce the window size to increase parallelism which should increase the throughput of our fast match path, however, verification performance is decreased by the number of false positives which increase as the windows decrease. We explore this Section V.

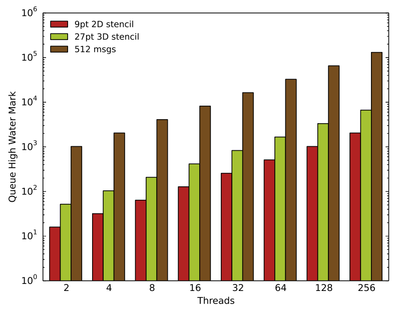
Because this technique creates the potential for false positives, an implementation must include a verification step where a 'fast matched' element is checked against the full matching information. Fuzzy Matching acts as an initial filter allowing for quick searching of a list. The verification step then does a traditional matching comparison on each fast matched entry. The overhead for this step is low, adding a small number of instructions for each fast matched entry. This could be problematic when false positive rates are high, however, as we show in Section V, many applications can leverage this



(a) Average Time to Drain Queue



(b) Average Search Depth



(c) Queue High Water Mark

Fig. 2: Multithreaded Scalability of Message Matching

technique and see few, if any, false positives. Additionally, this two phase process can be parallelized to mask the additional overhead by running the verification step while the SIMD search continues. Our results include all overheads of verifying Fuzzy Matched entries and this is done on the searching CPU core.

1) Software Implementations: We implemented Fuzzy Matching as an extension of our vector matching engine architecture. Instead of using two vectors, we combine the tag and rank into a single integer with variable bit-windows, and replace the multi-phase vector matching with a single comparison. Open MPI was selected as the MPI implementation for development as it had direct support for the Cray's Aries network. To compare against traditional matching engines, we created a specially modified Open MPI that collapsed the per-peer lists into per-communicator lists, to mimic a structure similar to MPICH's implementation. This allowed us to compare both major open source MPI implementation matching methods on a Cray Aries network (MPICH does not have Aries support and Cray MPI which is based on MPICH is not open source).

a) KNL: We designed and implemented a fuzzy matching architecture for the KNL processors to test our methods at scale on a Cray XC40 supercomputer. Because KNLs only have support for 32 bit integer vector operations, we focused on packing both tag and rank into 32 bits. For this prototype, we took the least significant 8 bits of the tag, and the least significant 24-bits of the rank. This decision was based on the idea that rank is a more distinguishing feature. While this means the rank-space is preserved, up to 16 million ranks, tag space has more potential for false positives.

b) Skylake: Intel's Skylake architecture introduced the AVX-512BW vector extensions with support for 8-bit and 16-bit integer vectors. In exchange for their smaller bit-width, these vectors increase the amount of parallelism to 64 and 32 concurrent integer compares respectively. To leverage these instructions, we designed and implemented fuzzy matching schemes for each available bit width. Efficiently preserving the most information becomes increasingly important. To preserve more information these matching implementations use full bit-width windows for both tag and rank and combine them using an bitwise `xor` operation. This preserves a large amount information with an efficient operation but has a few caveats: First, it can introduce false positives. For example, a message from rank 2 with a tag of 1 becomes a false positive with a message from rank 1 with a tag of 2. Secondly, wild cards become more expensive. Because the combination function modifies all of the fast-match bits based on both tag and rank, a wild-card in one field must be fast matched as a wild card in both.

2) Hardware: Fuzzy matching shows the potential for reducing the rank and tag space of MPI matching. With a fuzzy matching discrete design, simple very wide vectors can perform matching at high-speed and hardware can complete the verification of the partial matches. Integrated NICs can be designed to use vector units that they can share with the CPU when not in use, allowing for mutually beneficial co-design. In addition, fuzzy matching is useful from a software perspective even in the presence of message matching hardware. Fuzzy matching is useful for shared memory message passing and host-side matching from matching hardware that exceeds its memory capacity and must spill over into host memory to continue operation. Utilizing fuzzy matching in hardware can help to avoid such spill over events as matching data in the critical hardware components (like ternary content addressable memory, as used in today's switches) can be reduced in size, with full data held in larger traditional memories.

V. Application Tag and Rank Characteristics

Fuzzy Matching is feasible when rank and tag space can be compressed effectively, e.g., by truncation of rank and tag bits. To assess this, we collected tag and rank data from 15 proxy and full applications at 256, 512, and 1024 processes (216, 512, and 1000 for LULESH). For each rank, we recorded (i) the tags appearing in sends to that rank (the *tag set* for the rank), (ii) the number of times each tag was used for communication with the rank (the *tag weights* for the rank), (iii) the ids of peers issuing these sends (the *peer set* for the rank), and (iv) the number of times each id appears (the *peer weights* for the rank).

In general, we find application tag-spaces fall into three categories. First, 'zero-tag' applications use a single tag for all communication at all scales, effectively requiring no tags at all. Second, scale-invariant applications use the same number

TABLE I: App Categories

Category	Apps/Proxies
Zero-Tag	HPCG [9], LAMMPS [7], MCCK [10], MiniMD [9]
Scale-Invariant	Castro [11], Lulesh [12], MILC [13], MiniGhost [9], Pennant [14], AMG2013 [2], MiniAMR [9], MiniFE [9]
Scale-Sensitive	Kripke [15], NEKbone [16], FDS [17]

of tags regardless of scale. The number of tags can still vary widely, e.g., from three (LULESH, MiniFE, Pennant) to over 10,000 (Castro). Finally, the tag sets of 'scale-sensitive' applications grow with the number of processes. Of the applications surveyed, 27% were zero-tag, 53% scale-invariant, and 20% scale-sensitive (table I).

The compressibility of tag or rank-space depends on the tags and ranks used in inter-process communication, not on the global spaces. E.g., an application may utilize thousands of tags, but if no individual process has a tag set of cardinally greater than two, a single bit is sufficient to guarantee perfect discrimination.

To assess the trade-offs between the number of bits considered and false positives, we calculated the percentage of false positives for each process as a function of the number of bits used to match, weighted by the frequency of each tag in the processes' tag set. These percentages reflect the *potential* for false positives; it may be actual communication patterns are such that tags that could collide rarely do so.

The results for the worst-performing rank in each scale-invariant and scale-sensitive app for the 1024 process runs are shown in figure 3. The tag-spaces of Castro and MILC are nearly identical, so are plotted together. With a matching window of eight bits, half have perfect discrimination. Of the remaining five, Castro and MILC remain below 1% false positives (0.38% and 0.29%), NEKbone is below 5% (4.15%), and FDS is at 12.5%. Kripke (30%) and LULESH (66.67%) are the worst performers. Of these two, LULESH is scale-invariant: the high percentage is due entirely to the arbitrary choice of large tag values. Consequently, false positives can be avoided not only by increasing the matching window size, but also by choosing different tags.

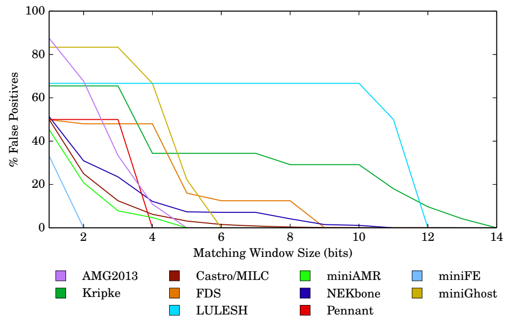


Fig. 3: False positives at 1024 procs.

For scale-sensitive applications, false positives vary with scale. Figure 4 shows this variation for FDS, NEKbone, and Kripke at 256, 512, and 1024 processes. The impact of scaling on false positives can differ greatly across applications. For example, in Kripke and FDS, a doubling in scale can significantly impact potential errors (e.g., with an eight bit window, FDS goes from 0.2% to 12.5% false positives between 512 and 1024 processes). In contrast, inter-process communication in NEKbone scales 'gracefully' insofar as false positives avoid dramatic increases (and can even decrease) as the number of bits required for perfect matching grows. For instance, at eight bits, NEKbone has false positive percentages of only 1.35%, 1.62%, and 4.15% at 256, 512, and 1024 processes, respectively.

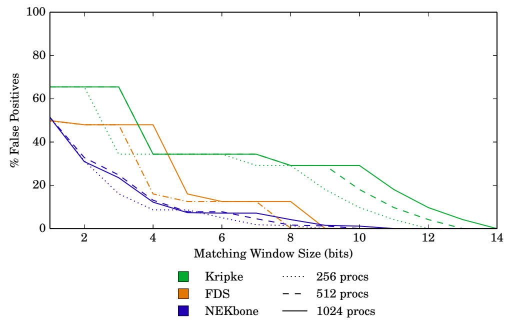


Fig. 4: False positives for scale-sensitive apps.

As expected, the rank-spaces of most (14) of the surveyed applications are scale-sensitive. Furthermore, of these applications, the majority (11) exhibited significant (greater than 5 percentage points) jumps in false positives across the same matching window size as scale increased. While all applications have perfect rank matching with 16 bit windows, and at 8 bits 11 remain below 5% at 1024 processes, these results indicate rank matching is more susceptible to false positives at scale than tag matching.

These results indicate Fuzzy Matching is a reasonable strategy, especially for tag matching. Of the 15 applications investigated, nine achieve perfect tag matching with a window of only eight bits, and the same number remain below 10% potential false positives with a window of four bits. False positives for scale-invariant applications can be addressed by choosing different tags. Finally, the potential penalties imposed on scale-sensitive applications depend on how tags and rank ids are distributed across point-to-point communications; scale-sensitivity is not itself proof the approach will yield unacceptable impacts on performance as the result of false positives.

VI. Experimental Results

To evaluate the viability of our vector matching and Fuzzy Matching approaches, we examine both its memory usage in a modern MPI implementation and its performance on a range of modern hardware platforms at a variety of scales.

A. Experimental Setup

Our experimental platform consists of two types of systems: a Cray XC40 using the KNL architecture and a testbed cluster using the Intel Xeon Sky Lake architecture. Cray XC40 runs used the Intel 17.0.1 compiler and default Cray programming environment. Sky Lake runs used the Intel 16.0 compiler. Our Cray XC40 platform, Trinity has two different node types, a dual socket node with Haswell E5-2698v3 CPUs and 128GB RAM, and a single socket node with a Knights Landing (KNL) Xeon Phi 7260 many-core CPU with 96GB RAM and 16GB MCDRAM. We have used the KNL partition of Trinity for these tests during its open science period in 2017. Trinity is the number ten ranked supercomputer in the world based solely on the performance of its Haswell partition as of June 2017.

The KNL architecture provided a testing platform with AVX-512 instructions and the Sky Lake architecture provided a platform to leverage AVX-512BW to test Fuzzy Matching on 16-bit and 8-bit integer vectors. Micro-benchmark results in this section represent the average of 10 iterations with error bars for standard deviation. We used a modified version of the OSU micro-benchmarks [18] for MPI bandwidth and latency. The micro-benchmarks were modified in three ways: First, we added an MPI barrier to ensure that receive requests were pre-posted. This allowed us to test the fast path of the underlying MPI implementation. This also required changing the receive calls to be non-blocking in latency to ensure pre-posting. Second, we cleared the cache between each iteration. This simulates a computation phase in a bulk synchronous application and allows us to do fine grained performance evaluation in that context. Finally, we added unmatched entries to the queue to evaluate performance with different receive queue lengths. We retain the communication pattern found in the original benchmarks (latency performs a ping-pong communication and bandwidth sends a series of non-blocking sends from one architecture to another). For the KNL experiments these benchmarks were run using two nodes leveraging the network for communication. The Sky Lake experiments were run as two processes on a single node utilizing shared memory communication.

B. Memory Savings in Open MPI

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

TABLE II: Open MPI Memory Overhead for different Run Scenarios for Trinity's KNL Partition

To evaluate MPI memory usage of our approach, we examined the memory costs of implementing our system in Open MPI. Open MPI's matching engine creates a 'proc' object for every other processor in the communicator. This object requires roughly 244 bytes. The per-peer matching structure contained within consists of two Open MPI list objects, an posted receive queue and an unexpected message queue. An empty Open MPI list structure requires 64 bytes of memory, 128 for both. This is good for performance as it separates out traffic from given peers and results in much shorter lists to search than a monolithic approach when receiving traffic from many peers at the same time.

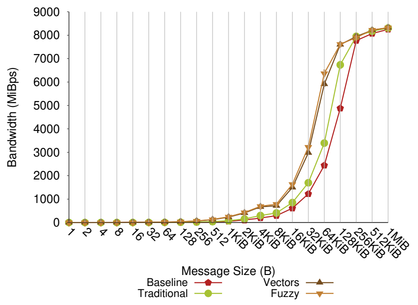
From a memory perspective, this becomes problematic with many core systems. In many-core systems we have more processes and more processes per node per application run versus a traditional CPU. This requires more MPI contexts per node, and for each MPI context to maintain state on more peers. This can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

Given this, the memory overhead for Open MPI grows as $O(number\_of\_nodes * processes\_per\_node^2)$ per full communicator per node. From this, we can calculate the memory overhead for different scenarios for just having the default communicator. Table II shows the memory overhead for Open MPI given four different scenarios; half scale and full scale MPI everywhere applications as they would appear on Trinity, and two forward looking scenarios with a doubling or quadrupling of the number of cores per node. This last scenario could also be mapped to MPI endpoints [19], where each endpoint has its own matching engine and the application is running with an endpoint per hyper-thread. As we can see in the table, the memory requirements of an MPI everywhere application grow by $O(processes\_per\_node^2)$ if the number of nodes is constant.

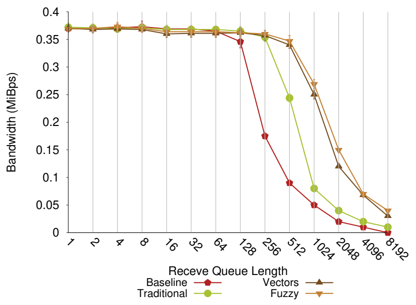
C. Knight's Landing - Small Scale Performance

Our initial evaluation focused on a fine-grained performance analysis of Fuzzy Matching and vector matching on a Xeon Phi system. These tests compare four different matching architectures: Open MPI's array of linked lists, a traditional MPICH style list (one large linked list), vector matching, and Fuzzy Matching. To evaluate these, we examined the bandwidth performance using micro-benchmarks for various message sizes and list lengths and three computational codes including two proxy applications (miniMD and AMG2013) and one full application (FDS). We use bandwidth instead of latency tests as the throughput of the matching engine is our primary concern for large codes. Bandwidth tests, like message rate tests, stress the matching engine to it's limit with a stream of messages, and are a common method of determining match engine performance [3].

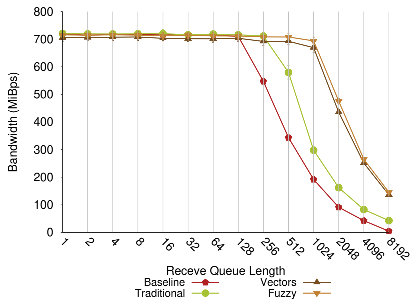
Figures 5a to 5c present the results of running our modified OSU bandwidth on the KNL architecture. Figures 5b and 5c show that the differences between the approaches with an empty queue is minimal. This demonstrates that Fuzzy Matching maintains very high zero-length queue performance, one of the most common modes of operation. As the queues grow in size, to long, yet realistic lengths (1024 elements) several interesting trends emerge: first, traditional performs better than the Open MPI approach. Examining this further, this is understandable as Open MPI's data structure separates entries by processes, which does not map well to this particular test, as it essentially creates one very long list as only one peer process communicates all of the data for the test. This means that the Open MPI approach behaves similarly to the single list, but with additional lookups to find the long single list. Secondly, the vector approach performs up to 3.18× better



(a) Queue Depth 1024



(b) 1 Byte Messages



(c) 4 KiB Messages

Fig. 5: KNL Bandwidth Performance

than the two baselines. Finally, while the trends between Fuzzy Matching and the vector approach are similar, Fuzzy Matching performs 6.8% better than the vector approach. This shows the potential for vector matching as the degree of vector-based parallelism is the same and the only benefit from Fuzzy Matching in these tests is a reduction in the number of vector operations required.

Figures 6a to 6c show the improvements to matching performance in different applications. These graphs show the maximum time an MPI process spends in the matching engine over the course of a run, averaged over three runs. This is representative of application impact in synchronous HPC codes as one process can delay computation. MiniMD, as shown in Figure 6a, represents a highly optimized communication pattern, doing a halo exchange each time step. Fuzzy Matching shows up to a 51.88% improvement over traditional and 77.91% over Open MPI. AMG 2013 (figure omitted for space), represents a typical scientific code, where much of its communication happens in a halo exchange but with occasional additional communication patterns required to move the data. Fuzzy Matching shows up to a 73.44% matching performance improvement over a traditional matching engine and 91.06% over Open MPI for AMG2013. AMG 2013 with a Laplace solver, as shown in Figure 6b, is a less communication optimized solver in AMG 2013, with higher communication requirements as observed in the figure. Fuzzy Matching for this version of AMG 2013 shows up to a 81.03% matching performance improvement over a traditional matching engine and 95.95% over Open MPI.

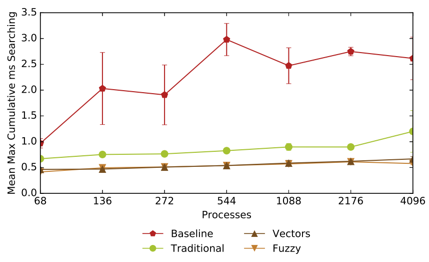
Both AMG2013 and MiniMD are scalable codes designed to interact with the matching engine in a scalable way. They spend a limited amount of time in the matching engine, and thus don't see a runtime performance impact. Fire Dynamics Simulator (FDS), as shown in Figure 6c, is a full application that has an occasional communication pattern that is more like a multi-threaded match queue than a traditional single-threaded one in both size and search depth. It is important to note that this graph has logarithmic y-axis and the communication bottleneck is increasingly problematic for the traditional approach as the problem scales. The vector and Fuzzy Matching approach both increase the scalability of this application. FDS shows up to a 67.98% matching performance improvement over a traditional matching engine and we observe up to a 31.0% runtime speedup. Open MPI, in this case, performs 19x better than our approaches. This is unsurprising as FDS's bottleneck is an application phase where every processes sends messages to rank 0 and this behavior aligns optimally with Open MPI's data structure. As noted in other papers, FDS spends an increasingly significant amount of its run time in the matching engine, so this can translate to a significant drop in overall runtime.

The small scale KNL results show the potential impact of vector and Fuzzy Matching. In the micro-benchmark tests we observed a significant increase in bandwidth from Fuzzy Matching and the vector approach over the linked list approaches. In all four of our application tests, Fuzzy Matching and the vector approach improve matching performance over a traditional linked list implementation. The benefits of Fuzzy Matching vs. vectors are not significant in these tests, as the level of SIMD parallelism does not increase over the vector approach. It is promising that we don't observe a overhead for the extra logic required for generating fast match values, the verification step, or any false positives in the applications.

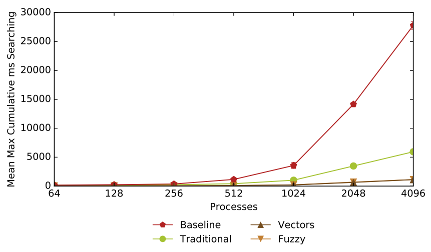
D. *Knight's Landing - Scaling Performance*

To test our approach at scale, we leveraged KNL partition of Trinity, a top 10 supercomputer. This allowed us to test up to 278,528 cores for four different applications. To test our performance impact at scale, we used four highly scalable codes from the US Department of Energy's Exascale Computing Project's designated Mini Applications [20]. When available, we used the best measured threading levels for each application [21]. The exception to this is Kripke; because detailed performance data was not available at the time of our large scale system time, we used 64 processes per node. These tests took a total of 26,738,688 core-hours.

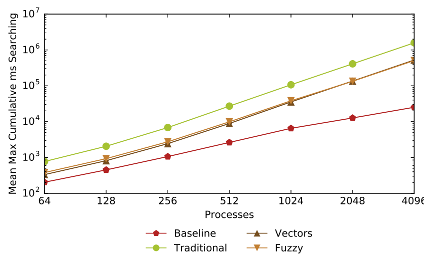
Each graph in this section presents a comparison between Fuzzy Matching and Open MPI's array of linked lists style matching engine and calculated memory savings. We ran each application 3 times per scale, displaying mean and standard deviation for all of the data points. All of the memory savings data is calculated based on Open MPI overhead generated in the same way as Section VI-B and a Fuzzy matching overhead



(a) MiniMD



(b) AMG2013 Laplace Large



(c) FDS (log scale)

Fig. 6: Mean maximum cumulative time spent searching in applications

that is based on an assumption that maximum of concurrent match list entries for these applications is 8192 entries.

Figure 7a shows the results for MiniMD. MiniMD is a proxy application for the Lennard-Jones molecular dynamics solver. This was run as an MPI everywhere application with one process per core (68 processes per node). Given the number of MPI processes it is unsurprising that the memory savings is up to 1.1 GiB at 2048 nodes. The performance does not show a significant difference, and the standard deviation is high, meaning that users would see the same performance but with lower memory overhead using Fuzzy Matching.

Figure 7b shows the results for LULESH. LULESH is a proxy application for Lagrangian explicit shock hydrodynamics codes. The threading level for these runs was a hybrid MPI+OpenMP, with 8 MPI processes per node using 8 threads per process. As only a limited number of MPI Processes share each node, the threading level for LULESH limits the amount of memory saving to 0.01 GiB. Like MiniMD and MiniFE, the performance difference in negligible with one anomaly at 1024 nodes. This anomaly is likely due to natural variance of Trinity during the busy open science period in which these tests were run.

Figure 7c shows the results for Kripke. Kripke is a proxy application for 3D Sn deterministic particle transport, specifically the ARDRA application. The threading level for these runs was MPI everywhere with highest power of 2 MPI processes per node available (64 processes per node). Since the threading level is similar to MiniMD, we observe similar memory savings of 1GiB on 2048 nodes. Unlike the other miniapps, there was a performance benefit to using Fuzzy Matching. Fuzzy Matching reduces the runtime by up to 6.12% for some scales. 5 out of the 6 core counts tested showed speedup with Fuzzy Matching.

Comparing the memory requirements of Open MPI's matching engine and the Fuzzy Matching approach, we can conclude that there is significant benefit to using Fuzzy Matching for MPI Everywhere applications for modern extreme scale systems. Given the applications results in this section, it is clear that the approach doesn't negatively impact highly scalable codes, even when running a hybrid MPI+X mode. We expect that the traditional matching would preform similar to the other two approaches used in these tests.

E. Sky Lake

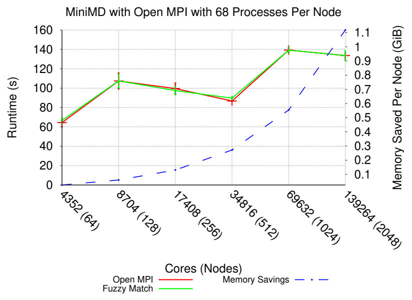
To examine the benefits of Fuzzy Matching, we ran tests on a small testbed system using Intel Xeon Sky Lake architecture. At the time of testing, Skylake Xeon Server CPUs were not available at scale; therefore we have used a small testbed of some of the first generally available Skylake CPUs for our non-scale dependent testing (bandwidth). This new CPU architecture introduces the AVX-512BW extensions, which allows us to test Fuzzy Matching for 16-bit and 8-bit integer vectors that provide an increased level of SIMD parallelism. In addition to the matching implementations we examined on the KNL, these tests include a 16-bit Fuzzy Matching and 8-bit Fuzzy Matching combining the tag and rank via a bitwise XOR. These tests were run using a shared memory transport to increase the load on the matching engine. This results in the fastest data transmission times, representing a worst case for matching as the deadline for performing a match is shorter than with lower bandwidth inter-node traffic.

Figures 8a to 8c show the bandwidth results from Sky Lake. Bandwidth performance for single element queues is slightly lower for Fuzzy Matching than for traditional or baseline approaches for medium to large size messages. This can be observed in Figure 8c. Figure 8a illustrates that at 1024 element lists, Fuzzy Matching has a distinct advantage in bandwidth over both MPICH and Open MPI style lists.

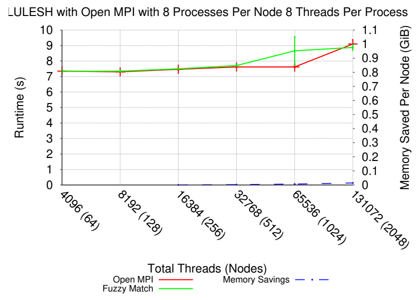
By varying the queue length for a fixed size message, we can see a slight degradation of performance for small list lengths in Figure 8b, however, traditional and Open MPI quickly drop in performance for medium to long length lists while Fuzzy Matching maintains its performance well until very large list lengths are reached (>1K). For 4KiB messages in Figure 8c, Open MPI and MPICH style lists are better until 64 elements are reached, then vector and Fuzzy Matching approaches maintain much better performance than non-vector approaches as queues grow.

We see a large benefit from each implementation of Fuzzy Matching. 8-bit Fuzzy Matching shows a 28$\times$ bandwidth improvement over Open MPI for short 1 byte messages with a 1024 search depth and 45$\times$ bandwidth improvement for medium 4KiB messages at 8196. This showcases a large performance benefit of Fuzzy Matching in an environment with few false positives.

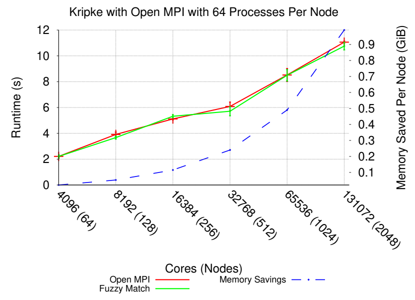
These results show a significant improvement using Fuzzy



(a) MiniMD

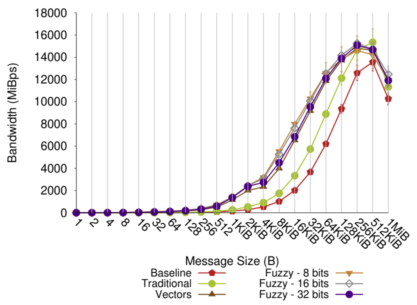


(b) LULESH

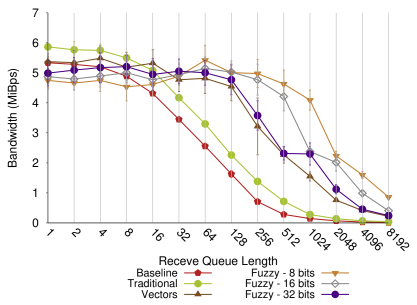


(c) Kripke

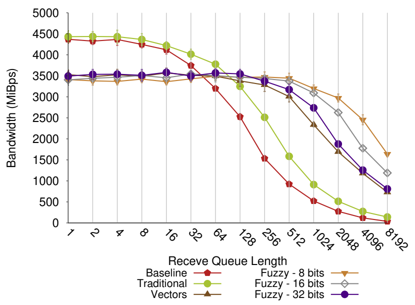
Fig. 7: Applications at Scale



(a) Queue Depth 1024



(b) 1 Byte Messages



(c) 4 KiB Messages

Fig. 8: Skylake Bandwidth Performance

Matching over exact vector matching. At 1024 messages in the list, we observe a 17.28% improvement with just the 32-bit Fuzzy Matching and 37.02% with the 8-bit Fuzzy Matching. As scales increase we see a larger difference for 8-bit Fuzzy Matching, where improves performance by 124.98% over vector matching at 8192 messages.

VII. Discussion

Why are existing Hash Table approaches insufficient? Hash tables have been sparsely adopted as a potential solution to the Matching Problem. They typically create a hash based on Rank or Tag to sort matching elements into a limited number of buckets. These solutions face several challenges. First, ordering and wildcards can be difficult to handle [22], leading to a relaxed version of the MPI Standard [23], or to a slow path for wildcard matching. Second, initial tests show that sublists are difficult for the prefetcher to interact with, causing significant slowdowns for medium search depths. This is particularly problematic when looking at phased applications as bucket sizes are unevenly distributed and can create laggard processes.

Are false positives an issue? We've shown (Section V) false positives are typically rare and often avoidable. Furthermore, because our tests examined conflicts across the entire run of the application, the data presented there is a strict upper bound and the observed false positive rate will likely be significantly lower.

Is this a memory locality issue? While much of the performance of vector matching and Fuzzy Matching can be attributed to the parallel vector operations, some of the performance benefit can be attributed to better interactions with memory locality. Our previous work exploring data locality for matching [24] shows a significant impact in packing two matching entries into a single cache line, but a limited benefit for increasing locality beyond that. While we have observed a significant performance improvement from using vector operations over a locality-improved approach, Fuzzy Matching does provide the benefit of increasing the number of matching (fast match) elements available per cache line.

Doesn't network offload handle this? Offload matching has been designed to handle current applications and is currently limited to a small number of entries per node. If this threshold is reached matching is run in software. Fuzzy Matching can be applied here, either as the software or as a hardware back up; implementing small bit width capabilities, more matching elements can fit in the same storage space. We discus this in depth in Section VIII.

Can I use Fuzzy Matching? The source code is currently available in Open MPI at https://github.com/open-mpi/ompi.

VIII. Related Work

There have been many recent efforts to improve MPI matching, for example using matching entries by communicator [25] or creating separate lists for each peer in the communicator along with a separate wildcard list [26]. In this later approach, each communicator has an array of linked lists that can be indexed by the communication peer. A search must check both elements in both the specific list for specified peer and the wild card list, compare the order of any matches and return the first match.

There have been several previous proposals to accelerate the matching engine without incurring memory penalties. Zounmevo and Afsahi [27], for example, proposed a matching engine that turns the search into a look-up in a 4-dimensional table. Rodrigues et al. [28] leveraged custom Processing in Memory (PIM) architectures to process multiple MPI match operations in a single ALU operation. In contrast, our approach uses commodity processor capabilities that are already present in virtually all of modern processors.

Flajslik, et al. [22] proposed a hash-bucket approach. In this approach, the match lists are fixed-size hash bucket that uses matching data as a key to separate the linked lists. Using 256 hash bins in this approach significantly reduces the number of match attempts per message. In addition, this paper solves the traditional issues with hash table approaches, allowing wild cards to be correctly handled with the hash table via a new marking scheme. However, having multiple hash buckets in this approach has memory overhead that can motivate using fewer buckets and consequently having lower performance. Bayatpour, et al. [29] extend the hash-table approach by dynamically switching between hashing and traditional matching. This approach avoids the overheads of the hashing scheme in short match lists. Unlike the hashing approach, fuzzy matching does not need to switch between techniques to provide good short search length performance.

Klenk, et al. [30] developed a GPU based message matching algorithm that is optimized by relaxing the constraints of MPI in terms of ordering and wild card support. This algorithm is designed with two phases, scan and reduce to leverage the available threads on a GPU. Their evaluation shows that this algorithm could provide a 10x to 80x increase in matching rate. This approach has three major caveats: it relies on a simplified set of MPI requirements, the existence of a GPU style accelerator card, and having a sufficiently long matching lists to make use of all the threads in a GPU warp. It also assumes that a GPU-based MPI implementation exists, which is not the case as of mid 2018.

Several works have explored the MPI+X tasking models that will create large matching lists. Stark et al. [31] and Barrett et al. [32] focus on exploring MPI+X task models that allows over-decomposition of a problem in to a number of tasks, which results in larger numbers of messages when all of the tasks communicate.

Vectorization has been used in databases and horizontal and vertical vectorization techniques have used SIMD instructions to search for results in bucketized hash tables [33], [34]. However, such solutions only need to search a single key, and have no ordering requirements like MPI imposes. While SIMD operations have been used to speed up set intersections [35], to the author's knowledge, this is the first work that leverages incomplete data to utilize small bit-width, many-element vectors.

Hardware support for message matching is provided some high-end modern interconnects like Bull-Atos's BXI NIC [36] based on the Portals 4 networking API [37] and Mellanox Connect-X 5 NICs [38]. Previous generation NICs have provided message matching capabilities as well such as Quadrics QSNet [39], Cray Seastar [40], and Myricom Myrinet adapters [41]. However, hardware solutions typically only support a limited number of match elements due to the size restrictions of high speed matching units. Therefore, such solutions can benefit from software assisted matching in cases of hardware resource exhaustion that may occur with future list lengths [1].

Improvements to the concurrency of MPI matching have been proposed that show excellent performance and scalability. Work by Dang et el. [23] proposed eliminating wild cards to allow for highly parallel hashing schemes that enable thread counts in the millions. While many codes can be adapted to not use wildcards, some scientific codes have valid reasons as to why they are needed, making such approaches non-universal.

Currently proposed MPI extensions could increase the rankspace and volume of matches. Endpoints [19], [42] allows for per-thread addressing in MPI. This can expand the rank space to equal the number of threads in a given job rather than the number of processes. Another approach to handling threading in MPI is the finepoints [43] proposal that seeks to avoid this per-thread rank space expansion as well as avoiding expansion of the number of messages that need to be matched by composing per-thread traffic into single operations. However, finepoints still requires matching that is consistent with today's requirements, and therefore can benefit from fuzzy matching.

IX. Conclusions

We have demonstrated the benefits of vector operation based MPI message matching engines and introduced an optimistic matching scheme that uses partial truth in matching elements to accelerate matches. Fuzzy matching is effective at very large scale on many-core architectures and has been demonstrated to be particularly beneficial on new CPUs capable of fine-grained vector comparisons. We have shown that with long lists, fuzzy matching with byte-level compares can provide 45$\times$ performance over traditional matching engines. In addition Fuzzy Matching improves matching time by 95.9% with a runtime improvement of up to 31.0% for small-scale runs of full applications and up to 6.1% for optimized proxy applications at scale. We have demonstrated that by using a Fuzzy Matching approach, we can provide similar or superior performance to Open MPI matching lists with significant memory savings.

References

[1] W. Schonbein, M. G. Dosanjh, R. E. Grant, and P. G. Bridges, "Measuring multithreaded message matching misery," in *Proceedings of the International European Conference on Parallel and Distributed Computing*, 2018.

[2] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang, "Multigrid smoothers for ultraparallel computing," *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2864–2887, 2011.

[3] B. W. Barrett, R. Brightwell, R. Grant, S. D. Hammond, and K. S. Hemmert, "An evaluation of MPI message rate on hybrid-core processors," *The International Journal of High Performance Computing Applications*, vol. 28, no. 4, pp. 415–424, 2014. [Online]. Available: http://dx.doi.org/10.1177/1094342014552085

[4] M. G. Dosanjh, S. M. Ghazimirsaeed, R. E. Grant, W. Schonbein, M. J. Levenhagen, P. G. Bridges, and A. 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*. ACM, 2018, p. 73.

[5] K. B. Ferreira, S. Levy, K. Pedretti, and R. E. Grant, "Characterizing mpi matching via trace-based simulation," in *Proceedings of the 24th European MPI Users' Group Meeting*. ACM, 2017, p. 8.

[6] D. E. Bernholdt, S. Boehm, G. Bosilca, M. Venkata, R. E. Grant, T. Naughton, H. Pritchard, and G. Vallee, "A survey of mpi usage in the u. s. exascale computing project," *Concurrency and Computation: Practice and Experience*, 2018, in press.

[7] S. Plimpton, P. Crozier, and A. Thompson, "LAMMPS-large-scale atomic/molecular massively parallel simulator," *Sandia National Laboratories*, vol. 18, 2007.

[8] E. Lindahl, B. Hess, S. Páll, and A. Metere, "Gromacs 5.0 benchmarks," 2017.

[9] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, vol. 3, 2009.

[10] K. G. Felker and A. R. Siegel, "MCCK user manual version 1.0 mathematics and computer science division argonne national laboratory," 2013.

[11] A. Almgren, V. Beckner, J. Bell, M. Day, L. Howell, C. Joggerst, M. Lijewski, A. Nonaka, M. Singer, and M. Zingale, "CASTRO: A new compressible astrophysical solver. i. hydrodynamics and self-gravity," *The Astrophysical Journal*, vol. 715, no. 2, p. 1221, 2010.

[12] I. Karlin, J. Keasler, and R. Neely, "LULESH 2.0 updates and changes," Tech. Rep. LLNL-TR-641973, August 2013.

[13] J. A. Bailey, C. Bernard, C. DeTar, M. Di Pierro, A. El-Khadra, R. Evans, E. Freeland, E. Gamiz *et al.*, "Fermilab lattice and MILC collaborations," *PoS LATTICE*, vol. 311, p. 2010, 2010.

[14] C. R. Ferenbaugh, "PENNANT: an unstructured mesh mini-app for advanced architecture research," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 4555–4572, 2015.

[15] A. J. Kunen, T. S. Bailey, and P. N. Brown, "KRIPKE - a massively parallel transport mini-app," in *American Nuclear Society Joint International Conference on Mathematics and Computation, Supercomputing in Nuclear Applications, and the Monte Carlo Method*, Jun 2015. [Online]. Available: http://www.osti.gov/scitech/servlets/purl/1229802

[16] I. Ivanov, J. Gong, D. Akhmetova, I. B. Peng, S. Markidis, E. Laure, R. Machado, M. Rahn, V. Bartsch, A. Hart *et al.*, "Evaluation of parallel communication models in Nekbone, a Nek5000 mini-application," in *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*. IEEE, 2015, pp. 760–767.

[17] K. McGrattan, S. Hostikka, R. McDermott, J. Floyd, C. Weinschenk, and K. Overholt, "Fire dynamics simulator, user's guide," *NIST special publication*, vol. 1019, p. 6th Edition, 2013.

[18] D. Panda *et al.*, "OSU microbenchmarks v5. 1," *URL http://mvapich. cse. ohio-state. edu/benchmarks*.

[19] J. Dinan, R. E. Grant, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur, "Enabling communication concurrency through flexible MPI endpoints," *The International Journal of High Performance Computing Applications*, vol. 28, no. 4, pp. 390–405, 2014.

[20] E. C. Project, "Exascale project - proxy applications," *URL: https://exascaleproject.github.io/proxy-apps/all-apps/*, 2017.

[21] M. Rajan, D. W. Doerfler, and S. D. Hammond, "Trinity benchmarks on Intel Xeon Phi," Sandia National Laboratories, Tech. Rep., 2015.

[22] M. Flajslik, J. Dinan, and K. D. Underwood, "Mitigating MPI message matching misery," in *International Conference on High Performance Computing*. Springer, 2016, pp. 281–299.

[23] H.-V. Dang, M. Snir, and W. Gropp, "Towards millions of communicating threads," in *Proceedings of the 23rd European MPI Users' Group Meeting*. ACM, 2016, pp. 1–14.

[24] M. G. F. Dosanjh, S. M. Ghazimirsaeed, R. E. Grant, W. Schonbein, M. J. Levenhagen, P. G. Bridges, and A. Afsahi, "The case for semi-permanent cache occupancy," in *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP 2018, 2018, pp. 73:1–73:11.

[25] M. D. Team, "Ch4," 2016, Last accessed: 1/4/2017. [Online]. Available: https://www.mpich.org/2016/11/13/mpich-3-3a2-released

[26] O. D. Team, "Openmpi," 2017, Last accessed: 28/3/2017. [Online]. Available: https://www.open-mpi.org

[27] J. A. Zounmevo and A. Afsahi, "A fast and resource-conscious MPI message queue mechanism for large-scale jobs," *Future Generation Computer Systems*, vol. 30, pp. 265–290, 2014.

[28] A. Rodrigues, R. Murphy, R. Brightwell, and K. D. Underwood, "Enhancing NIC performance for MPI using processing-in-memory," in *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2005, pp. 8–pp.

[29] M. Bayatpour, H. Subramoni, S. Chakraborty, and D. K. Panda, "Adaptive and dynamic design for MPI tag matching," in *IEEE International Conference on Cluster Computing*. IEEE, 2016, pp. 1–10.

[30] B. Klenk, H. Froning, H. Eberle, and L. Dennison, "Relaxations for high-performance message passing on massively parallel SIMT processors," in *31st International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017.

[31] D. T. Stark, R. F. Barrett, R. E. Grant, S. L. Olivier, K. T. Pedretti, and C. T. Vaughan, "Early experiences co-scheduling work and communication tasks for hybrid MPI+X applications," in *Proceedings of the 2014 Workshop on Exascale MPI*. IEEE Press, 2014, pp. 9–19.

[32] R. F. Barrett, D. T. Stark, C. T. Vaughan, R. E. Grant, S. L. Olivier, and K. 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*. ACM, 2015, pp. 30–39.

[33] O. Polychroniou, A. Raghavan, and K. A. Ross, "Rethinking simd vectorization for in-memory databases," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1493–1508.

[34] M. Zukowski, S. Héman, and P. Boncz, "Architecture-conscious hashing," in *international workshop on Data management on new hardware*. ACM, 2006, p. 6.

[35] B. Schlegel, T. Willhalm, and W. Lehner, "Fast sorted-set intersection using SIMD instructions," in *ADMS@ VLDB*, 2011, pp. 1–8.

[36] S. Derradji, T. Palfer-Sollier, J.-P. Panziera, A. Poudes, and F. A. Wellenreiter, "The BXI interconnect architecture," *In Proceedings of the 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects, HOTI*, pp. 18–25, 2015. [Online]. Available: http://dl.acm.org/citation.cfm?id=2861514

[37] B. W. Barrett, R. Brightwell, R. E. Grant, S. Hemmert, K. Pedretti, K. Wheeler, K. Underwood, R. Riesen, A. B. Maccabe, and T. Hudson, "The portals 4.1 networking programming interface," 2017.

[38] M. Corp, "Understanding MPI tag matching and rendezvous offloads (ConnectX-5)," https://community.mellanox.com/docs/DOC-2583, accessed: 2018-07-25.

[39] F. Petrini, W.-c. Feng, A. Hoisie, S. Coll, and E. Frachtenberg, "The quadrics network: High-performance clustering technology," *IEEE Micro*, vol. 22, no. 1, pp. 46–57, 2002.

[40] R. Brightwell, K. T. Pedretti, K. D. Underwood, and T. Hudson, "Seastar interconnect: Balanced bandwidth for scalable performance," *IEEE Micro*, vol. 26, no. 3, pp. 41–57, 2006.

[41] C. Keppitiyagama and A. Wagner, "Asynchronous mpi messaging on myrinet," in *Parallel and Distributed Processing Symposium., Proceedings 15th International*. IEEE, 2001, pp. 8–pp.

[42] S. Sridharan, J. Dinan, and D. D. Kalamkar, "Enabling efficient multithreaded mpi communication through a library-based implementation of mpi endpoints," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 487–498.

[43] R. E. Grant, A. Skjellum, and P. V. Bangalore, "Lightweight threading with MPI using persistent communications semantics." Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2015.