SAND2020-12557C

# Low-cost MPI Multithreaded Message Matching Benchmarking

Whit Schonbein, Scott Levy, W. Pepper Marts, Matthew G.F. Dosanjh, and Ryan E. Grant
Center for Computational Research
Sandia National Laboratories*
Albuquerque, New Mexico, USA
Email: wwschon@sandia.gov, sllevy@sandia.gov, wmarts@sandia.gov, mdosanj@sandia.gov, regrant@sandia.gov

***Abstract*—The Message Passing Interface (MPI) standard allows user-level threads to concurrently call into an MPI library. While this feature is currently rarely used, there is considerable interest from developers in adopting it in the near future. There is reason to believe that multithreaded communication may incur additional message processing overheads in terms of number of items searched during demultiplexing and amount of time spent searching because it has the potential to increase the number of messages exchanged and to introduce non-deterministic message ordering. Therefore, understanding the implications of adding multithreading to MPI applications is important for future application development.**

**One strategy for advancing this understanding is through 'low-cost' benchmarks that emulate full communication patterns using fewer resources. For example, while a complete, 'real-world' multithreaded halo exchange requires 9 or 27 nodes, the low-cost alternative needs only two, making it deployable on systems where acquiring resources is difficult because of high utilization (e.g., busy capacity-computing systems), or impossible because the necessary resources do not exist (e.g., testbeds with too few nodes). While such benchmarks have been proposed, the reported results have been limited to a single architecture or derived indirectly through simulation, and no attempt has been made to confirm that a low-cost benchmark accurately captures features of full (non-emulated) exchanges. Moreover, benchmark code has not been made publicly available.**

**The purpose of the study presented in this paper is to quantify how accurately the low-cost benchmark captures the matching behavior of the full, real-world benchmark. In the process, we also advocate for the feasibility and utility of the low-cost benchmark. We present a 'real-world' benchmark implementing a full multithreaded halo exchange on 9 and 27 nodes, as defined by 5-point and 9-point 2D stencils, and 7-point and 27- point 3D stencils. Likewise, we present a 'low-cost' benchmark that emulates these communication patterns using only two nodes. We then confirm, across multiple architectures, that the low-cost benchmark gives accurate estimates of both number of items searched during message processing, and time spent processing those messages. Finally, we demonstrate the utility of the low-cost benchmark by using it to profile the performance impact of state-of-the-art Mellanox ConnectX-5 hardware support for offloaded MPI message demultiplexing. To facilitate further research on the effects of multithreaded MPI on message matching behavior, the source of our two benchmarks is to be included in the next release version of the Sandia MPI Micro-Benchmark Suite.**

## I. Introduction

The Message Passing Interface (MPI) standard includes support for allowing user-level threads to concurrently call into an MPI library [1]. While this level of thread support (`MPI_THREAD_MULTIPLE`) is rarely used in current practice, a recent survey of application developers in the United States' Department of Energy Exascale Computing Project indicates a majority (86%) are interested in taking advantage of the opportunities it affords [2]. Specifically, emerging memory technologies such as high-bandwidth memories (HBM) offer significantly higher memmory bandwidths at the cost of available storage, precluding the standard 'MPI everywhere' strategy of instantiating a distinct MPI process per core or thread context. A single, multithreaded MPI process is one solution. Therefore, understanding the implications of adding multithreading to MPI codes is important for future application development.

Contributing to this need are performance considerations. Since under `MPI_THREAD_MULTIPLE`, individual threads can issue sends and receives, this mode of operation may lead to significant increases in the number of messages exchanged. Furthermore, since many threads in a process may be issuing receives or sends concurrently, multithreaded communication may result in deviations to the expected ordering of messages, making the standard strategy of posting receives in the order incoming messages are anticipated to arrive ineffective. This suggests that utilizing multithreaded communication can result in more time spent in MPI message matching, with potential consequences for application performance. This calls for methods for assessing the impacts of pursuing multithreaded communication under MPI.

One such method is to *emulate* a multithreaded halo exchange, as in [3]. This approach has the benefit of being 'low-cost' in the sense it requires fewer resources to execute, e.g., two nodes rather than the 9 or 27 required by full exchanges. Consequently, the benchmark can be deployed on systems where acquiring a full allocation is difficult (e.g., because it is under high utilization) or even impossible (e.g., because it lacks the necessary number of nodes). However, to our knowledge, studies based on this approach were limited to a single architecture, and their authors made no attempt to confirm that such a strategy accurately represents the behavior of a multithreaded halo exchange involving a full complement of processes.

An alternative is to provide a benchmark implementing the complete, 'real-world' multithreaded halo exchange. In comparison to the low-cost approach, this strategy has the benefit of imposing more realistic demands on MPI and the underlying network, but also incurs additional resource costs. There exists published work based on this 'real-world' approach: Levy et. al. [4] used such a benchmark to collect MPI traces that were then input

*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.

into a simulator. However, the authors of that paper did not evaluate the matching behavior of the benchmark on actual hardware.

The purpose of the study we present in this paper is to bridge the gap between the low-cost and real-world approaches by quantifying how accurately the former captures the matching behavior of the latter. In the process, we advocate for the feasibility and utility of the low-cost benchmark. We present a 'real-world' benchmark implementing a full multithreaded halo exchange on 9 and 27 nodes, as defined by 5-point and 9-point 2D stencils, and 7-point and 27-point 3D stencils. Likewise, we present a 'low-cost' benchmark that emulates these communication patterns using only two nodes. We then confirm, across multiple architectures, that the low-cost benchmark gives reasonable estimates of both number of items searched during message processing, and time spent processing those messages. Finally, we demonstrate the utility of the low-cost benchmark by using it to profile the performance impact of state-of-the-art Mellanox ConnectX-5 hardware support for offloaded MPI message demultiplexing.

In this paper we make the following contributions:

- an evaluation of the low-cost benchmark relative to the baseline, across multiple architectures;
- a case study using the low-cost benchmark to examine the effect of message matching offloading on a multithreaded halo exchange;
- the public release of the source code for the low-cost and real-world multithreaded communication benchmarks that we designed and implemented.[1]

The remainder of this paper is structured as follows. In section II we describe the low-cost and real-world benchmarks, and the systems upon which they are executed. In section III we compare results obtained from the low-cost and real-world benchmarks. In section IV, we deploy the low-cost benchmark to evaluate performance using offloaded message matching. We provide a summary of related work and briefly conclude in Sections V and VI.

## II. Methodology

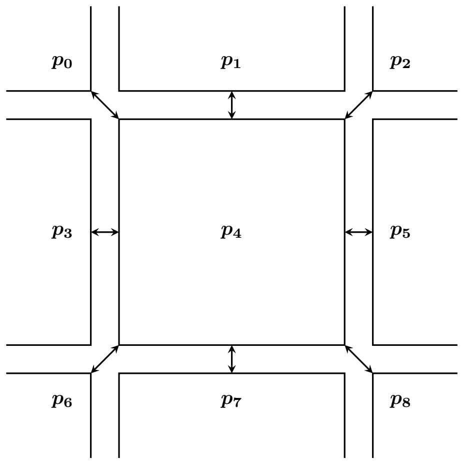

Fig. 1: A single-threaded, nine-point halo exchange.

[1]The benchmarks will be included in the next release version of the Sandia MPI Micro-Benchmark Suite (SMB) [5].

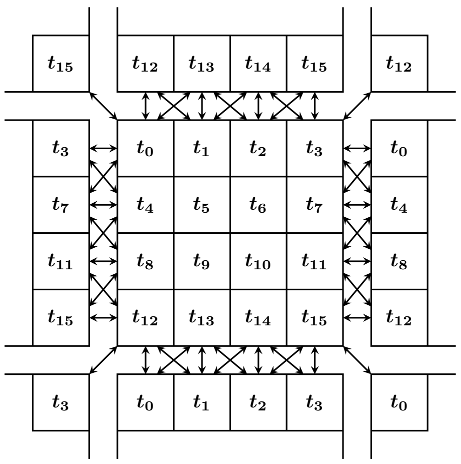
$t_{15}$ $t_{12}$ $t_{13}$ $t_{14}$ $t_{15}$ $t_{12}$
$t_3$ $t_0$ $t_1$ $t_2$ $t_3$ $t_0$
$t_7$ $t_4$ $t_5$ $t_6$ $t_7$ $t_4$
$t_{11}$ $t_8$ $t_9$ $t_{10}$ $t_{11}$ $t_8$
$t_{15}$ $t_{12}$ $t_{13}$ $t_{14}$ $t_{15}$ $t_{12}$
$t_3$ $t_0$ $t_1$ $t_2$ $t_3$ $t_0$

Fig. 2: A multithreaded, nine-point halo exchange.

The halo exchange communication pattern is common among scientific applications [6], [7]. Figure 1 illustrates a simple single-threaded, 9-point, two-dimensional halo exchange between process $p_4$ and its neighbors. In a typical bulk-synchronous processing (BSP) application using this stencil, after each process completes its assigned work, it exchanges data with its eight nearest neighbors. Once this exchange is complete, it continues into a new work phase.

Figure 2 represents a straightforward multithreaded version using the same stencil communication pattern. In this scenario, work on $p_4$ is divided between 16 threads ($t_0$ through $t_{15}$). During the communication phase, each thread is responsible for exchanging messages with its neighboring threads, based on the same 9-point stencil pattern shown in Figure 1.

Assuming that inter-thread communication within the same process is handled outside of the MPI matching engine (e.g., via shared memory), one can calculate the number of sending and receiving threads, and the total number of messages exchanged between processes. Specifically, a decomposition can be viewed as a collection of 'faces' of different dimensions. For example, a square (2D) decomposition comprises 4 faces of dimension 0 (corners)[2], 4 faces of dimension 1 (edges)[3] and 1 face of dimension 2 (interior)[4]. If we assume, for purposes of exposition, that all dimensions have the same length, $x$, then the total number of threads, $t(k)$, on faces of dimension $k$ can be expressed as a variation on the standard equation for calculating the faces of a hypercube of dimension $d$:

$$t(k) = (x-2)^k 2^{d-k} \binom{d}{k} \tag{1}$$

for $x \geq 2$. In this equation, the first factor ($(x-2)^k$) is the volume of the $k$-dimensional face, expressed as the number of threads it contains. The remainder of the equation computes the number

[2]Each corner comprises a face containing a single thread (i.e., a zero-dimensional point).

[3]Each edge comprises a face containing a one-dimensional line of threads (e.g., $t_1$-$t_2$ in Figure 2).

[4]The interior comprises a face containing a two-dimensional grid of threads (e.g., $t_5, t_6, t_9$, and $t_{10}$ in Figure 2

of $k$-dimensional faces. The total number of threads is simply the sum of the number of threads for each possible dimension:

$$\sum_{k=0}^{d} t(k) \tag{2}$$

The number of threads involved in inter-process communication depends on the dimensionality of the stencil used to define the communication. For this work, we limit our analysis to standard 5 and 9-point 2D stencils, and 7 and 27-point 3D stencils. If the dimensionality of the stencil ($d_s$), is greater than the dimension of the thread decomposition ($d$), then the set of participating threads includes every thread in the decomposition. Otherwise, the number of participating threads is equal to Equation 2, where the upper bound of the summation is reduced to $d-1$, i.e., the single $d$-dimensional face is excluded. The case where $d_s < d$ is beyond the scope of this paper.

The number of messages exchanged by each thread depends on the face it belongs to and the stencil used. For communication limited to the Von Neumann neighborhood (5 and 7-point stencils), the number of messages is given by Equation 3. For communication within the Moore neighborhood (9 and 27-point stencils) the number of messages is given by Equation 4.

$$m(k) = 2d_s - (d + k) \tag{3}$$

$$m(k) = 3^{d_s} - 3^k 2^{d-k} \tag{4}$$

That is, the number of messages exchanged by a thread is equal to the total number of messages ($2d_s$ for the Von Neumann neighborhood, and $3^{d_s}$ for the Moore neighborhood) minus the number of intra-process messages. The total number of messages exchanged is therefore the sum of the messages processed by each thread on each face:

$$\sum_{k=0}^{d} t(k)m(k) \tag{5}$$

If $d_s = d$, the upper bound of the summation is reduced to $d-1$.

This analysis can be straightforwardly extended to handle the case where the number of threads in each dimension is not equal, e.g., in two dimensions, a rectangular decomposition instead of a square decomposition. Table I summarizes the number of messages processed by the receiver for each of the decompositions and stencils considered in the experiments reported in this paper. These data illustrate how multithreading can significantly increase the number of messages exchanged.

The *low-cost* multithreaded benchmark we consider in this study emulates a full halo exchange using two MPI processes: a sender and a receiver (cf. [3]). The number of OpenMP threads that the receiver uses is determined by a version of our Equation 2. The number of OpenMP threads that the sender uses is equal to the number of threads issuing sends to the receiver, across all communicating neighbor processes; this number can be calculated with a minor modification to Equations 1 and 2. That is, while the receiver corresponds to the center process in an actual halo exchange (e.g., $p_4$ in Figure 1), the threads from surrounding processes (e.g., $p_0$-$p_3$ and $p_5$-$p_8$ in Figure 1) are collected onto a single sending process.

Following a bulk synchronous processing pattern, the receiving threads post their receives, and then, after a barrier, the sending threads issue sends. No 'work' is done between barriers; the benchmark focuses solely on communication. To enforce ordering, each message has a unique tag, and each thread posts its receives in the same order as the corresponding sends are issued. The low-cost benchmark supports 5 and 9-point two-dimensional stencils, and 7 and 27-point three-dimensional stencils. For all experiments reported in this paper, regardless of benchmark, thread affinity is `OMP_PROC_BIND=spread`.

|  | 1x1 | 2x1 | 2x2 | 4x2 | 4x4 | 8x4 | 8x8 | 16x8 | 16x16 |
|---|---|---|---|---|---|---|---|---|---|
| 5pt | 4 | 6 | 8 | 12 | 16 | 24 | 32 | 48 | 64 |
| 9pt | 8 | 14 | 20 | 32 | 44 | 68 | 92 | 140 | 188 |
|  | 1x1x1 | 2x1x1 | 2x2x1 | 2x2x2 | 4x2x2 | 4x4x2 | 4x4x4 | 8x4x4 | 8x8x4 |
| 7pt | 6 | 10 | 16 | 24 | 40 | 64 | 96 | 160 | 256 |
| 27pt | 26 | 50 | 92 | 152 | 272 | 464 | 728 | 1256 | 2072 |
|  | 1x1x1 | 1x1x2 | 1x1x4 | 1x1x8 | 1x1x16 | 1x1x32 | 1x1x64 | 1x1x128 | 1x1x256 |
| 7pt | 6 | 10 | 18 | 34 | 66 | 130 | 258 | 514 | 1026 |
| 27pt | 26 | 50 | 98 | 194 | 386 | 770 | 1538 | 3074 | 6146 |

TABLE I: Number of messages processed by receiver under different thread-level decompositions and stencils.

MPI message matching is commonly implemented using two queues: a Posted Receive Queue (RQ) for unsatisfied receive requests, and an Unexpected Messages Queue (UQ) for messages that have not yet been matched to a receive request. When an application posts a receive request, it is first compared to all of the messages that are currently waiting in the UQ to determine whether an existing message will satisfy the request. If no message in the UQ satisfies the request, then the receive request is appended to the RQ. Similarly, when an incoming message arrives, it is compared to all of the receive requests that are currently in the RQ. If the message satisfies none of the requests in the RQ, then the message is appended to the UQ.

Recent versions of Open MPI include an option to utilize a traditional dual-queue matching engine, optimized for SIMD operations (see [8]). To collect data on the number of items searched and time spent searching, we instrumented this traditional dual-queue matching engine without enabling these optimizations.[5] Our instrumented Open MPI reports: (i) the number of items searched in the RQ before a match is found for each incoming message; and (ii) the total amount of time spent searching the RQ. Because all of the receives are pre-posted in this benchmark, all received messages will have a match in the RQ.

The *real-world* benchmark performs an actual multithreaded halo exchange. Specifically, nine processes are used for 2D stencils, and 27 processes are used for 3D stencils. Each process is partitioned into $x \times y$ or $x \times y \times z$ threads, again depending on the dimensionality of the process-level decomposition. Threads on the perimeter of the decomposition participate in inter-process message exchanges with those of neighboring processes, posting receives and issuing sends. Communication is non-periodic (i.e., non-toroidal). Like the low-cost benchmark, each MPI process in this benchmark resides on a distinct node. All receives are pre-posted in the same order as the corresponding sends, and data is collected from the center process via the modified Open MPI described above.

Note that an earlier iteration of the real-world benchmark appears in [4]. However, in that case the benchmark was used

[5]Open MPI hash dd74c6252f8947e213e83f470024f3a4ce78b10b

solely to generate MPI profiling-layer traces for input into a simulator. Here, we assess the actual MPI message processing behavior incurred by a multithreaded halo exchange.

To compare the performance of the real-world and low-cost benchmarks, we ran both benchmarks on a Cray XC40. This system comprises two compute node partitions. In the first, each compute node has two sockets, each of which contains an Intel Xeon ES-2698 v3 (Haswell) processor operating at 2.3GHz. Each processor has 16 cores with 2 hardware threads per core, for a total of 64 thread contexts. In the second partition, each node has a single socket containing a 68-core Intel Xeon Phi 7250 (Knights Landing) processor operating at 1.4GHz. Each core has 4 hardware threads, for a total of 272 thread contexts. The nodes are connected via a Cray Aries network. All tests were conducted on a single cabinet from one of the Cray XC40's partitions. Therefore, the results are for a fully connected network (not a dragonfly, as would be the case for communication that spanned multiple cabinets).

There are multiple ways that an application developer can decompose their problem with multiple threads. We consider three possible decompositions: two-dimensional 'square' (e.g., $1\times1$, $2\times1$, $2\times2$, $4\times2$, $\ldots$), three-dimensional 'cube' (e.g., $1\times1\times1$, $2\times1\times1$, $2\times2\times1$, $2\times2\times2$, $\ldots$), and three-dimensional 'linear', which scales only along the $z$ axis (e.g., $1\times1\times1$, $1\times1\times2$, $1\times1\times4$, $\ldots$). Square and cube decompositions represent typical ways that applications decompose problems. The linear decomposition represents a less efficient corner case.

The low-cost and real-world benchmarks were executed on each partition, using one MPI process per node, with each of the three types of decomposition. Each decomposition was scaled up until the point oversubscription would occur on the receiver side. For the low-cost benchmark, this means sender-side oversubscription is permitted. For the real-world benchmark, this means that oversubscription is not permitted, since all processes are both senders and receivers. For the square decomposition, data was collected for 5 and 9-point stencils; for the cube and linear decompositions, data was collected for 7 and 27-point stencils. Because internal nodes in a regular halo exchange have an equal communication overhead, and assessing this overhead is the goal of the benchmark, scaling beyond 9 nodes (for 2D decompositions) or 27 nodes (for 3D decompositions) is not necessary. For each configuration, the benchmarks were run 50 times, with each run executing two trials, i.e., two halo exchanges. All reported results are for the second of the two trials. All messages contained 8-byte payloads.

For the case study – assessing the potential impact of dedicated offloading message matching hardware – we ran the low-cost benchmark experiments on an ARM-based testbed with offloaded matching disabled and then repeated the experiments with offloaded matching enabled. On this system, each compute node has two sockets. Each socket contains a 28-core Cavium ThunderX2 CN9775 processor operating at 2GHz. The ARM testbed's network utilizes Mellanox ConnectX-5 interfaces, which provide dedicated hardware assistance for MPI message matching [9]. This capability is accessed via OpenUCX [10]. For the experiments in this paper, we used UCX 1.7.0, configured with optimizations (`--enable-optimizations`) and multithreaded support (`--enable-mt`) enabled. We used the same version of MPI as in the previous experiments. However, because Open MPI's message matching is bypassed by UCX, our MPI instrumentation was also bypassed. Therefore, for these results, we report the time spent processing messages as measured in the benchmark code. This includes a barrier (to ensure the sender cannot begin issuing messages before the timer is started) and a call to `MPI_Waitall`. In addition to supporting hardware offloading, UCX also includes message matching optimizations in software (in the form of binning messages into 1021 bins according to tag and source), and these optimizations are reflected in the reported times. Hardware offloading of message matching is controlled via the `UCX_RC_MLX5_TM_ENABLE` environment variable. The default threshold recommended by Mellanox for engaging offloading is 1024 bytes. For our experiments, we kept this default value, meaning that even when offloading is enabled, it is not used for messages whose size is less than 1KiB.

## III. Comparison

In this section, we compare results derived from the two benchmarks, confirming that the low-cost benchmark provides a reasonable approximation of a genuine multithreaded halo exchange.

### A. *Resource usage*

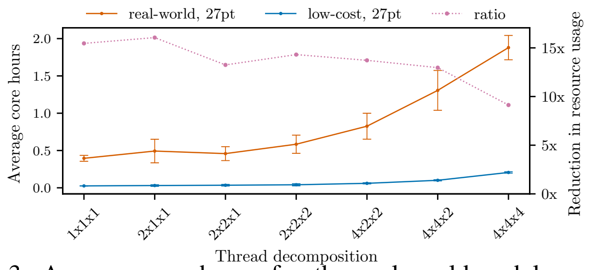

Fig. 3: Average core hours for the real-world and low-cost multithreaded halo exchange benchmark executing a 27-point 3D halo exchange.

Figure 3 compares the average number of core hours (over 30 runs) for the real-world and low-cost benchmarks executing a multithreaded 27-point, 3D halo exchange. Results were obtained on 2.3GHz Intel Xeon ES-2698 v3 processors using Cray MPICH v7.7.6. As the size of the thread decomposition increases, the low-cost benchmark remains under 0.25 core hours, while the real-world benchmark approaches 2 core hours. The reduction in resource usage afforded by the low-cost benchmark ranges from 9.1x to 16.1x relative to the real-world benchmark. This reduction in resource usage decreases as the number of threads increases, because the amount of work each process in the low-cost benchmark has to do increases faster than the work done by each of the processes in the real-world benchmark. However, we expect the low-cost benchmark to be used at small scale, mostly by MPI researchers and for MPI performance regression testing. MPI halo exchange performance is often tested/monitored on supercomputers, which can now use a lower-cost benchmark to achieve this goal.

### B. *Items searched*

The total number of items searched during a halo exchange and the search depths for the processing of individual messages

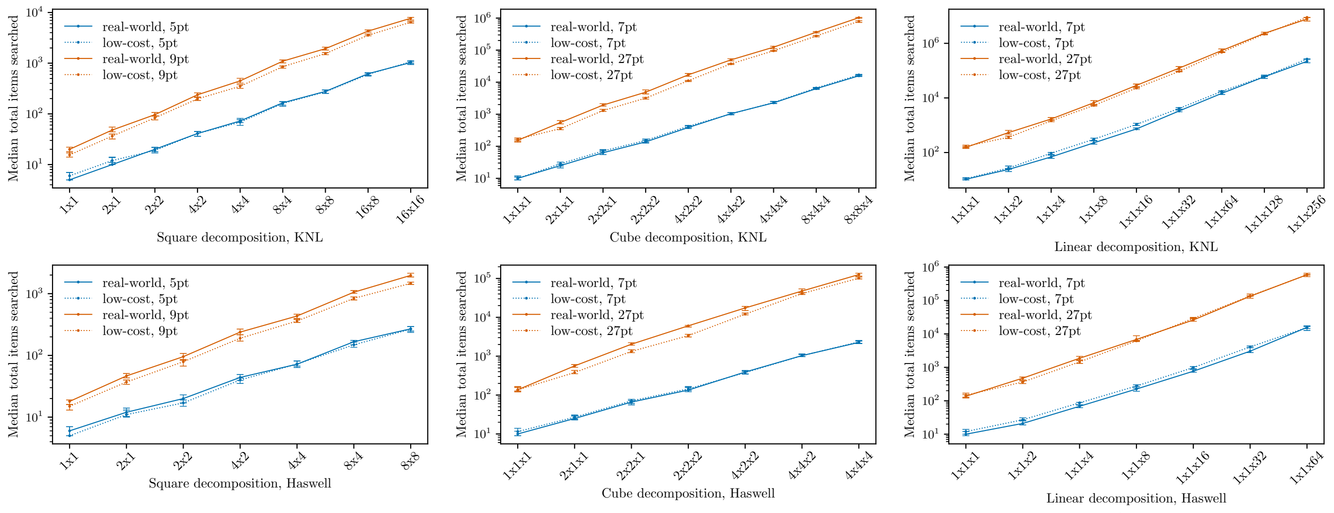

Fig. 4: Median total number of items searched under the low-cost and real-world benchmarks, for 2D (5 and 9-point stencils) and 3D (7 and 27-point stencils) decompositions, on Haswell and KNL architectures.

are useful for understanding the overheads due to the non-deterministic behavior introduced by multi-threaded communication [3]. Consequently, we begin by contrasting the total number of items searched reported by the low-cost and real-world benchmarks. Figure 4 summarizes this data for square, cube, and linear decompositions for KNL and Haswell architectures. Data points represent median search depth (over 50 runs), and error bars extend to the first and third quartiles. Note that the $y$-axis is logarithmic and the range differs between the plots of data from the KNL and Haswell architectures. Furthermore, the decomposition for the KNL architecture has two extra data points over Haswell due to the greater number of available execution contexts.

Both benchmarks confirm the hypothesis that the non-deterministic ordering introduced by multithreaded communication leads to increased search depths [3]. For instance, under the real-world benchmark on Haswell, a 4x4x4 cube decomposition with 27-point stencil communication results in a total number of items searched that, on average, is 170.5 times larger than the ideal, where each search matches on the first item.

Comparing the two benchmarks, the mean absolute error across all stencils and decompositions, expressed as a ratio to the median reported by the real-world benchmark, is 16.4% ($\sigma = 11.1$). We observe that disagreement between the low-cost and real-world benchmarks occurs under larger (9 and 27-point) stencils, and this typically involves the low-cost benchmark underestimating the total number of items searched. For instance, the average error of 5-point stencils versus 9-point stencils for square decompositions, across both architectures, is 7.6% and 16.6%, respectively. Likewise, for 7-point stencils versus 27-point stencils and cube decompositions, these values are 6.3% and 27.7%.

Figure 5 offers a more detailed view of the data for three different cube decompositions on the Haswell architecture, selected because they clearly manifest this trend. In these histograms, the $x$-axis specifies bins of numbers of items searched during the processing of each incoming message during an exchange, and the $y$-axis the number of searches of that depth, averaged across all 50 trials. In all three cases, there is strong agreement between low-cost and real-world benchmarks for 7-point stencil communication. In contrast, for 27-point stencils, the low-cost benchmarks exhibits greater numbers of shallower searches, and correspondingly fewer deep searches, in comparison to the real-world benchmark. Furthermore, the maximum search depths for the real-world benchmark consistently exceed that of the low-cost benchmark.

### C. Matching Overheads

Determining search times can give a user an idea of how expensive MPI overhead is for multiple decomposition strategies. Even close agreement between low-cost and real-world benchmarks with respect to mean or median number of items searched does not guarantee that the temporal overhead of searching the match lists is similar because the temporal costs of searching may not be linear. Consequently, to assess how the low-cost benchmark relates to the real-world version regarding processing times, we compare time to process all items in the posted receive queue (i.e., 'queue drain times') for each architecture, stencil, and decomposition. Results are summarized in Figure 6. As before, data points represent medians and error bars are first and third quartiles. The $y$-axis is logarithmic and differs between architectures and the plots of KNL data have two additional data points.

Previous work has observed that the costs of multi-threaded communication can be prohibitively expensive, requiring more time for message processing than currently allocated by current scientific applications for an entire compute-plus-communication cycle [3]. The results reported here confirm this observation across both benchmarks and architectures. For example, on Haswell, 27-point communication on a 4x4x4 cube decomposition exceeds 1 millisecond for queue processing.

Comparing the benchmarks, the mean absolute error across all stencils and decompositions (calculated as in Section III-B) is 24.9% ($\sigma = 18.6$). However, as shown in the subfigures, discrepancies between benchmarks are more pronounced for decompositions and stencils with fewer numbers of messages; as the number of messages increases, disagreement decreases. For example, whereas the average error across 5-point stencils is 36.0%, the average error decreases to 17.6% for 27-point cube decompositions, and to 16.7% for 27-point linear decompositions.

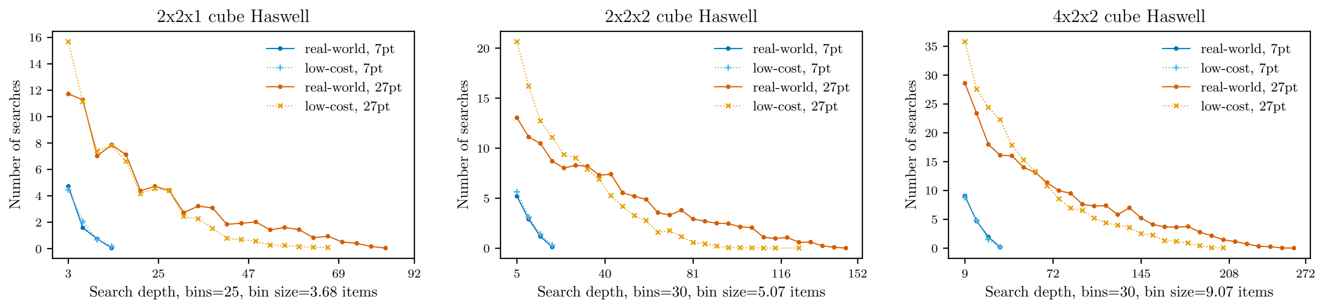

Fig. 5: Histograms showing average number of searches at each bin of items searched.

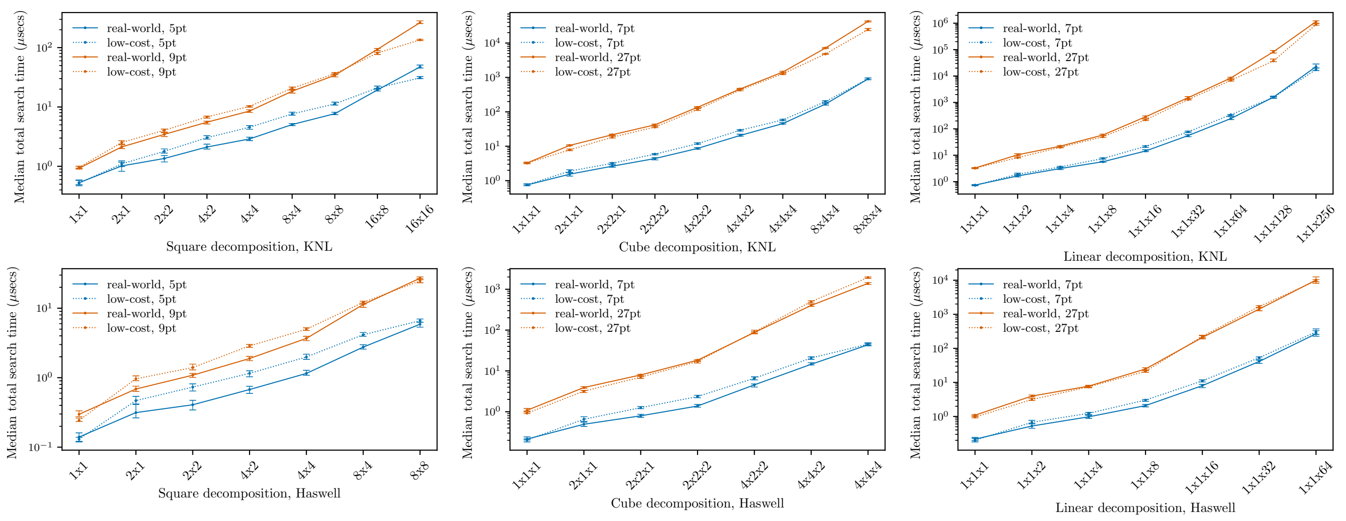

Fig. 6: Median queue drain times for 2D (5 and 9-point stencils) and 3D (7 and 27-point stencils) decompositions.

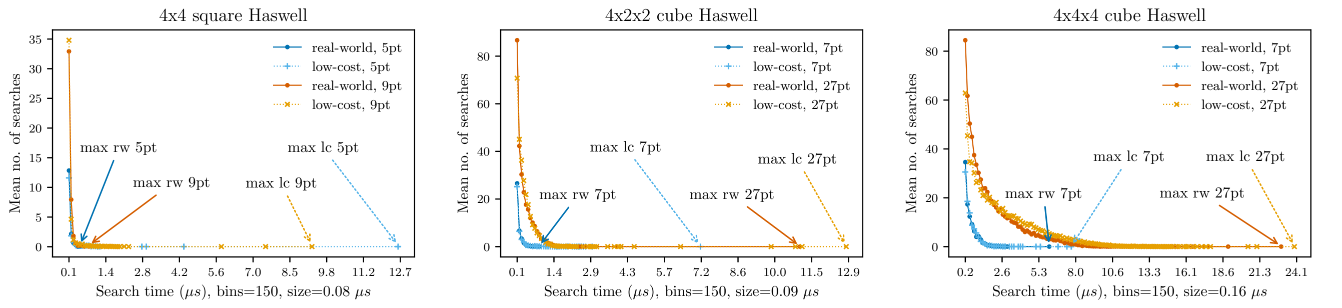

Fig. 7: Histograms showing average time per search for selected decompositions. Annotations show the minimum and maximum search times for real-world (rw) and low-cost (lc) benchmarks, for each stencil.

Figure 7 shows results from a square and two cube decompositions, all executed on Haswell, chosen because they are representative of the trend towards agreement. As these figures illustrate, a notable contributor to discrepancies between low-cost and real-world results are outliers: on all stencils, for lower message counts, the low-cost benchmark has more extreme slow outliers than the real-world benchmark, and this gap closes as the number of messages increases. For example, for the 4x4 decomposition, the maximum real-world search times under 5 and 9-point stencils are 436ns and 828ns, respectively, while the corresponding low-cost values are 12686ns and 9367ns, exhibiting gaps of one or two orders of magnitude. For the 4x2x2 cube decomposition, the maximum real-world 7-point stencil is 909ns while the low-cost is 7213ns, and for 27-point these are 11051ns and 12941ns. Finally, for the 4x4x4 cube decomposition, the maximums are 6251ns vs. 7817ns (7-point) and 23080ns vs. 24061ns (27-point), indicating that by this point, the gaps have closed significantly.

## *D. Discussion*

Previous work has studied the impact of multithreading on message queues and processing times on a KNL system [3]. As part of the present work, we have reproduced that study. While our results support the conclusion that multithreaded communication leads to increased search depths and potentially problematic message processing times, observed message processing times trend higher than in that earlier work. This is likely due to the Spectre/Meltdown exploit, a variant of which affects look-ahead pointer dereferencing [11]. In contrast to the data presented here, results gathered for the earlier work were obtained prior to the exploit being patched. Unfortunately we were unable to test this hypothesis due to the unavailability of unpatched nodes.

As noted above, the results indicate a user of the low-cost benchmark should be aware that the benchmark may, under larger message counts, underestimate the number of items searched during message processing. This is not unexpected: whereas in the low-cost benchmark, the non-deterministic order of message arrivals at the receiver is due entirely to thread-level competition at the single sender, the real-world benchmark adds network-induced non-determinism, i.e., variation in message arrivals because the messages may take different routes through the network. We hypothesize that as adaptive routing gains traction in HPC, this divergence between benchmarks will decrease, assuming processes participating in the low-cost benchmark are sufficiently far removed in the network topology.

Finally, the user should also keep in mind that the low-cost benchmark can overestimate message processing time for smaller message counts in comparison to the real-world benchmark. Again, this is not unexpected: because there is only a single sender in the low-cost benchmark, messages arrive at a higher rate than in the real-world benchmark, where arrival times are diluted by having multiple senders. This means in the low-cost benchmark, more memory accesses are occurring in a smaller window of time, introducing contention and leading to the outliers noted in Figure 7. As the number of messages increases, the real-world benchmark trends towards the behavior exhibited by the low-cost benchmark.

## IV. Case Study

In this section, we deploy the low-cost benchmark to establish initial expectations regarding multithreaded message matching performance on an ARM system with Mellanox ConnectX-5 network interfaces. Specifically, as shown in Section III, multithreaded communication implies increased search depths and time spent searching. How might the message matching offloading offered by ConnectX-5 hardware affect these results? This case study also illustrates the utility of the low-cost benchmark. Our target system is often under high utilization: acquiring 27 nodes can take on the order of hours. Given the runtime of the benchmarks (seconds) and the number of configurations to be explored (see below), executing the full-cost benchmark is not feasible.

### *A. The impact of offloading*

As described in Section II, we ran experiments with the low-cost benchmark on a ConnectX-5 system with hardware matching enabled and then repeated those experiments with hardware matching disabled. Because this system bypasses our instrumented MPI, we record the time spent processing incoming messages, within the benchmark rather than within MPI. Furthermore, since offloading is only active above a certain threshold of message size (1024B, the default), we also vary message sizes to span this transition point (from 8B to 1MiB). For each set of parameters, the benchmark was executed 50 times, with 11 emulated halo exchanges per run; the data presented here discards the first exchange from each run, for a total of 500 trials.

Results for three message sizes – small (512B), medium (16KiB), and large (1MiB) – are presented in Figure 8. The left column shows results for square decompositions, and the right for cube decompositions. Plotted values are medians, and error bars extend to the first and third quartiles.

As expected, when message sizes are below the threshold for engaging offloaded matching (<1024B), processing time is not affected by hardware matching being enabled (Figure 8, row 1). When message sizes are modestly above the threshold (e.g., 16KiB), hardware matching accelerates processing time across most square and cube decompositions and stencils (row 2). Within each type of decomposition, speedup is more pronounced for larger stencil sizes (larger numbers of messages). For square decompositions, the average speedup for the 5-point stencil is $1.13\times$, and for the 9-point stencil it is $1.23\times$. Similarly, for cube decompositions, the average speedup for the 7-point stencil is $1.21\times$ while that for the 27-point is $1.39\times$.

However, our benchmark also reveals (Figure 8, row 3) that when messages become large (e.g., 1MiB), hardware-assisted message matching may actually slow down message processing relative to software message matching. For 1MiB messages, the average slowdown is remarkably consistent across types of decomposition and stencils: $1.85\times$ for 5-point square, $1.87\times$ for 9-point square, $1.87\times$ for 7-point cube, and $1.88\times$ for 27-point cube. These results suggest that, on this system, there exists a 'window of effectiveness' where hardware offloading can reduce the overhead of message matching for multithreaded communication patterns.

To better characterize this window, we plot processing time across all message sizes for selected square and cube decompositions in Figure 9 (note the gap between 8B and 512B). We observe that offloaded matching typically reduces processing time for message sizes beginning near the default threshold of 1024B, but this benefit disappears between 32KiB and 64KiB, at which point offloaded matching incurs additional overheads in comparison to not using offloading. This effect threshold is observed regardless of the number of messages being processed.

### *B. Discussion*

These results confirm the potential benefits of hardware-assisted message matching in handling the increased overhead of multithreaded applications. However, they also suggest that there are situations where offloading may be detrimental to application performance (e.g., when message sizes exceed 32KiB).

Previous work by Marts et al. [9] has also considered the impact of ConnectX-5 message matching offloading. The results presented in this paper are consistent with this earlier study. Marts et al. saw similar performance benefits that were limited to a window of message sizes between 1KiB and 16KiB. For messages larger than 16KiB, they showed that hardware-assisted message matching was slower than software message matching. The same effect is seen in Figure 9. These results are also consistent with Mellanox's default threshold which limits hardware-assisted message matching to messages that are larger than 1024B.

Marts et al. also observed that even for message sizes within this window, UCX tag binning collisions can decrease the perfor-

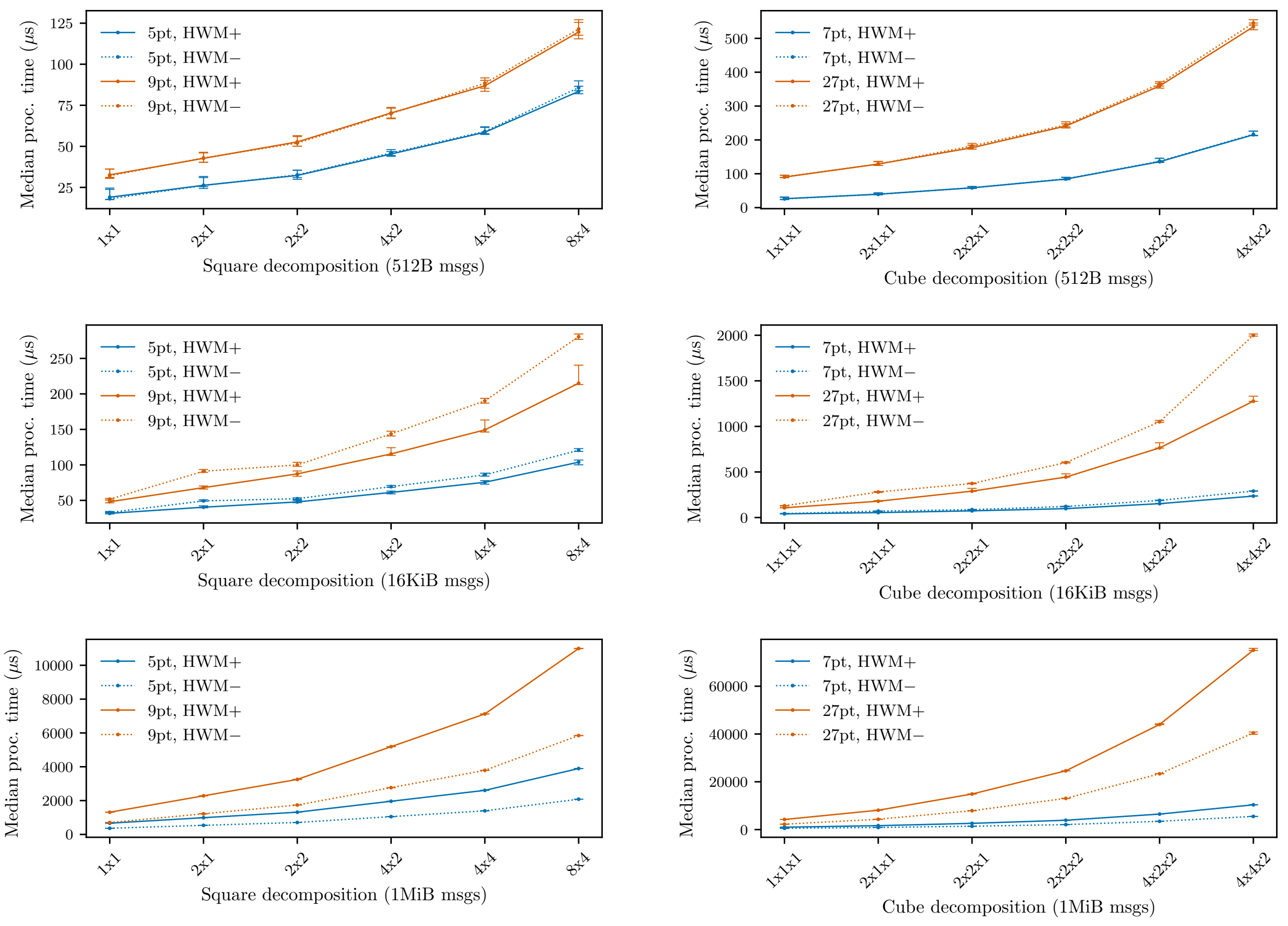

Fig. 8: Median time spent processing messages with hardware matching enabled (HWM+) and disabled (HWM−), for square and cube decompositions and small (512B), medium (16KiB), and large (1MiB) messages.

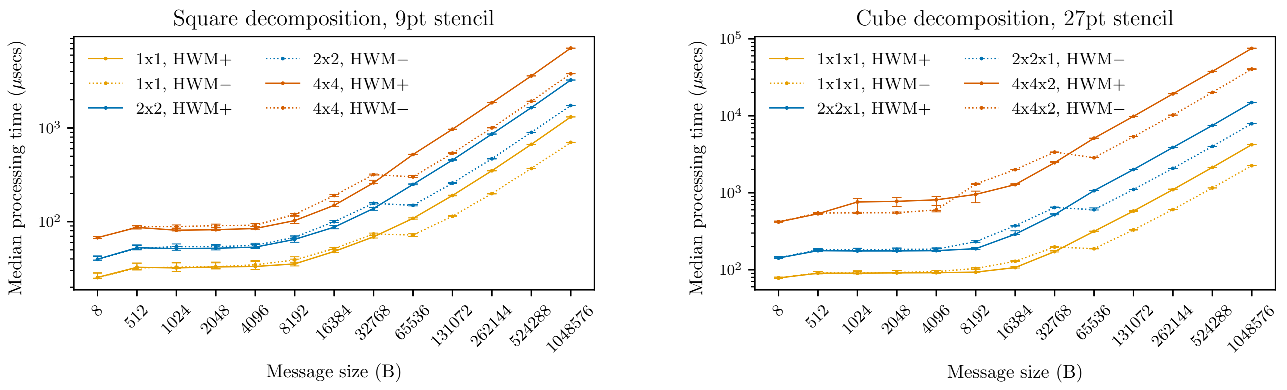

Fig. 9: Median time spent processing messages by message size, square and cube decompositions, with hardware matching enabled (HWM+) and disabled (HWM−).

mance of hardware message matching. Therefore, the fact that we observe a speedup for these medium-sized messages suggests that our benchmark is balancing its tags across the bins effectively. Provided that a user sends messages in this ideal size window and ensures that their tag use is consistent with low binning collisions, Mellanox's hardware offloading has the potential to alleviate some of the aforementioned concerns in MPI matching overhead.

## V. Related Work

Levy et al. [4] used a benchmark with similar communication characteristics to gather MPI traces to simulate a multithreaded halo exchange. In contrast to the work described in our paper, a multithreaded halo exchange trace was the desired output; they did not collect search depth or timing information on real hardware. In addition, Levy et al. used a full number of processes for the halo exchange like the comparison benchmark that we use to assess our low-cost benchmark.

More generally, a number of researchers have considered the potential overhead of MPI queue searches. For example, Balaji et al. [12], and Underwood and Brightwell [13] characterized queue search overheads, illustrating the costs of engaging with larger queues. Brightwell et al. [14] investigated queue features such as maximum size and search depth for several applications and benchmarks. Similarly, Ferreira et al. [15] observed that, for a variety of scientific applications, search depths tend to remain relatively shallow. In contrast to these results, the benchmarks presented here highlight the potential impact of non-determinism in message ordering implied by multithreaded communication. Attempts to model such behaviors have been made as well, *see e.g.*, [16]. Efforts have also been made to evaluate MPI message matching performance on a variety of system architectures, *see e.g.*, [17].

Considerable effort has been dedicated to optimizing message matching. Strategies include dedicating a hardware thread [18], leveraging cache behavior [19], incorporating hash tables or other data structures [20]–[22], utilizing GPUs [23] or vector units [8], and offloading matching to the NIC [24]–[27]. Most of these solutions do not consider or are not designed for the sorts of larger queue search depths exhibited by the multithreaded benchmarks given here. This is expected to remain the case for emerging smartNIC designs because recent proposals that utilize hardware message matching have similar matching list capacities as previous approaches to message matching, *cf.* [28], [29]. Previous work on new matching engines on ConnectX-5 have shown reasonable performance, *see* [9], [30], but have not been evaluated with MPI multithreading. Additionally, Dosanjh et al. [31] recently revisited the overall benefits of using hardware MPI message matching.

Several works have addressed multithreading support in MPI by improving implementation internals [32]–[34], and proposing new interfaces [35]–[37]. In addition to traditional send/receive multithreading and matching overheads, work has also examined multithreading in the context of MPI one-sided communication [38], [39]. Other MPI multi-threaded benchmarks have been proposed for basic MPI functionality and performance testing [40]–[42].

## VI. Conclusion

The potential implications of multithreaded MPI motivated us to explore benchmarks for assessing the performance impact of multithreaded communication under communication patterns common to scientific computing. While 'low-cost' and 'real-world' benchmarks have been proposed, reported results have been limited to a single architecture or derived indirectly through simulation, and to our knowledge, no attempt has been made to confirm that a low-cost benchmark accurately captures features of full (non-emulated) exchanges such as the number of items searched or time spent searching. In this paper, we have described the design and implementation of these benchmarks, and compared data acquired from each across multiple architectures. We found that the low-cost benchmark provides an accurate estimate of the number of items searched and the time spent searching with the real-world benchmark, with some qualifications. We hypothesize these differences are a consequence of substituting communication between two nodes for the many-node communication present in the real-world benchmark; further investigation into this hypothesis is a topic for future work.

We also used the low-cost benchmark to assess the effect of hardware support for offloaded message matching provided by state-of-the-art Mellanox ConnectX-5 network interfaces. Using the low-cost benchmark in this case allowed us to collect data that we could not feasibly collect with the real-world benchmark because of the high utilization of the host system. These data show that offloading can help alleviate the impact of multithreaded MPI communication for a particular range of message sizes, although for smaller messages with large message counts (1KiB-4KiB messages, 4x4x2 decomposition, 27-point stencils), and for large messages regardless of number of messages (>32KiB messages), it is preferable to utilize software-based matching, as offloading may incur additional overheads. Finally, we have publicly released the source for these two benchmarks as part of the Sandia MPI Micro-Benchmark Suite benchmark suite.

## References

[1] Message Passing Interface Forum, *MPI: A Message-passing Interface Standard, Version 3.1*. High-Performance Computing Center Stuttgart, University of Stuttgart, 2015.

[2] D. E. Bernholdt, S. Boehm, G. Bosilca, M. G. Venkata, R. E. Grant, T. Naughton, H. P. Pritchard, M. Schulz, and G. R. Vallee, "A survey of MPI usage in the US exascale computing project," *Concurrency and Computation: Practice and Experience*, vol. 0, no. 0, p. e4851, 2018. [Online]. Available: http://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4851

[3] W. Schonbein, M. G. F. Dosanjh, R. E. Grant, and P. G. Bridges, "Measuring Multithreaded Message Matching Misery," in *Euro-Par 2018: Parallel Processing*, ser. Lecture Notes in Computer Science, M. Aldinucci, L. Padovani, and M. Torquati, Eds. Springer International Publishing, 2018, pp. 480–491.

[4] S. Levy, K. B. Ferreira, W. Schonbein, R. E. Grant, and M. G. F. Dosanjh, "Using simulation to examine the effect of MPI message matching costs on application performance," *Parallel Computing*, vol. 84, pp. 63–74, May 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819118303272

[5] D. Doefler, B. W. Barrett, R. E. Grant, M. G. Dosanjh, and T. Groves, "Sandia MPI Micro-Benchmark Suite (SMB)," *Sandia National Laboratories*, 2009–2020. [Online]. Available: http://www.cs.sandia.gov/smb/index.html

[6] M. Bianco, "An Interface for Halo Exchange Pattern," Swiss National Supercomputing Centre, Tech. Rep., 2013.

[7] P. G. Raponi, F. Petrini, R. Walkup, and F. Checconi, "Characterization of the Communication Patterns of Scientific Applications on Blue Gene/P," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, 2011, pp. 1017–1024.

[8] M. G. F. Dosanjh, W. Schonbein, R. E. Grant, P. G. Bridges, S. M. Gazimirsaeed, and A. Afsahi, "Fuzzy Matching: Hardware Accelerated MPI Communication Middleware," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2019, pp. 210–220.

[9] W. P. Marts, M. G. F. Dosanjh, W. Schonbein, R. E. Grant, and P. G. Bridges, "MPI Tag Matching Performance on ConnectX and ARM," in *EuroMPI'19: Proceedings of the 26th European MPI Users' Group Meeting*, Zurich, Switzerland, 2019.

[10] OpenUCX, "UCX: Unified Communication X." [Online]. Available: http://www.openucx.org/

[11] V. G. V. Larrea, M. J. Brim, W. Joubert, S. Boehm, M. Baker, O. Hernandez, S. Oral, J. Simmons, and D. Maxwell, "Are we witnessing the spectre of an HPC meltdown?" *Concurrency and Computation: Practice and Experience*, vol. 0, no. 0, p. e5020. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5020

[12] P. Balaji, A. Chan, W. Gropp, R. Thakur, and E. Lusk, "The Importance of Non-Data-Communication Overheads in MPI," *The International Journal of High Performance Computing Applications*, vol. 24, no. 1, pp. 5–15, Feb. 2010. [Online]. Available: https://doi.org/10.1177/1094342009359528

[13] K. D. Underwood and R. Brightwell, "The impact of MPI queue usage on message latency," in *International Conference on Parallel Processing, 2004. ICPP 2004.*, Aug. 2004, pp. 152–160 vol.1.

[14] R. Brightwell, S. Goudy, and K. Underwood, "A preliminary analysis of the MPI queue characterisitics of several applications," in *2005 International Conference on Parallel Processing (ICPP'05)*, Jun. 2005, pp. 175–183.

[15] 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*, ser. EuroMPI '17. New York, NY, USA: ACM, 2017, pp. 8:1–8:11. [Online]. Available: http://doi.acm.org/10.1145/3127024.3127040

[16] P. G. Bridges, M. G. Dosanjh, R. Grant, A. Skjellum, S. Farmer, and R. Brightwell, "Preparing for exascale: modeling MPI for many-core systems using fine-grain queues," in *Proceedings of the 3rd Workshop on Exascale MPI*, 2015, pp. 1–8.

[17] B. W. Barrett, R. Brightwell, R. Grant, S. D. Hammond, and K. S. Hemmert, "An evaluation of MPI message rate on hybrid-core processors," *International Journal of High Performance Computing Applications*, vol. 28, no. SAND-2014-20472J, 2014.

[18] K. Vaidyanathan, D. D. Kalamkar, K. Pamnany, J. R. Hammond, P. Balaji, D. Das, J. Park, and B. Joó, "Improving concurrency and asynchrony in multithreaded MPI applications using software offloading." ACM Press, 2015, pp. 1–12. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2807591.2807602

[19] 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: Understanding the Impact of Data Locality on Network Processing," in *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP 2018. New York, NY, USA: ACM, 2018, pp. 73:1–73:11. [Online]. Available: http://doi.acm.org/10.1145/3225058.3225130

[20] M. Flajslik, J. Dinan, and K. D. Underwood, "Mitigating MPI Message Matching Misery," in *High Performance Computing*, ser. Lecture Notes in Computer Science. Springer, Cham, 2016, pp. 281–299.

[21] 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. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X13001489

[22] S. M. Ghazimirsaeed, R. E. Grant, and A. Afsahi, "A dynamic, unified design for dedicated message matching engines for collective and point-to-point communications," *Parallel Computing*, vol. 89, p. 102547, 2019.

[23] B. Klenk, H. Fröening, H. Eberle, and L. Dennison, "Relaxations for High-Performance Message Passing on Massively Parallel SIMT Processors," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 855–865.

[24] N. Tanabe, A. Ohta, P. Waskito, and H. Nakajo, "Network Interface Architecture for Scalable Message Queue Processing," in *2009 15th International Conference on Parallel and Distributed Systems*, Dec. 2009, pp. 268–275.

[25] S. Derradji, T. Palfer-Sollier, J. P. Panziera, A. Poudes, and F. W. Atos, "The BXI Interconnect Architecture," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, Aug. 2015, pp. 18–25.

[26] Mellanox, "Understanding MPI Tag Matching and Rendezvous Offloads (ConnectX-5)," 2018. [Online]. Available: https://community.mellanox.com/s/article/understanding-mpi-tag-matching-and-rendezvous-offloads–connectx-5-x

[27] K. D. Underwood, K. S. Hemmert, A. Rodrigues, R. Murphy, and R. Brightwell, "A Hardware Acceleration Unit for MPI Queue Processing," in *19th IEEE International Parallel and Distributed Processing Symposium*, Apr. 2005, pp. 96b–96b.

[28] T. Hoefler, S. Di Girolamo, K. Taranov, R. E. Grant, and R. Brightwell, "sPIN: High-performance Streaming Processing In the Network," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: ACM, 2017, pp. 59:1–59:16. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126970

[29] W. Schonbein, R. E. Grant, M. G. F. Dosanjh, and D. Arnold, "INCA: in-network compute assistance," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. Denver, Colorado: Association for Computing Machinery, nov 2019, pp. 1–13. [Online]. Available: http://doi.acm.org/10.1145/3295500.3356153

[30] C. Zimmer, S. Atchley, R. Pankajakshan, B. E. Smith, I. Karlin, M. L. Leininger, A. Bertsch, B. S. Ryujin, J. Burmark, A. Walker-Loud *et al.*, "An evaluation of the CORAL interconnects," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2019, p. 39.

[31] M. G. Dosanjh, R. E. Grant, P. G. Bridges, and R. Brightwell, "Re-evaluating network onload vs. offload for the many-core era," in *2015 IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 342–350.

[32] T. Patinyasakdikul, D. Eberius, G. Bosilca, and N. Hjelm, "Give MPI threading a fair chance: A study of multithreaded MPI designs," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2019, pp. 1–11.

[33] M. Si, A. J. Peña, J. Hammond, P. Balaji, M. Takagi, and Y. Ishikawa, "Casper: An asynchronous progress model for MPI RMA on many-core architectures," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 665–676.

[34] S. Sridharan, J. Dinan, and D. D. Kalamkar, "Enabling efficient multithreaded MPI communication through a library-based implementation of MPI endpoints," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 487–498.

[35] 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.

[36] R. E. Grant, M. G. F. Dosanjh, M. J. Levenhagen, R. Brightwell, and A. Skjellum, "Finepoints: Partitioned Multithreaded MPI Communication," in *High Performance Computing*, ser. Lecture Notes in Computer Science, M. Weiland, G. Juckeland, C. Trinitis, and P. Sadayappan, Eds. Springer International Publishing, 2019, pp. 330–350.

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

[38] M. G. Dosanjh, T. Groves, R. E. Grant, R. Brightwell, and P. G. Bridges, "RMA-MT: a benchmark suite for assessing MPI multi-threaded RMA performance," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2016, pp. 550–559.

[39] N. Hjelm, M. G. Dosanjh, R. E. Grant, T. Groves, P. Bridges, and D. Arnold, "Improving MPI multi-threaded RMA communication performance," in *Proceedings of the 47th International Conference on Parallel Processing*, 2018, pp. 1–11.

[40] T. Patinyasakdikul, X. Luo, D. Eberius, and G. Bosilca, "Multirate: A flexible MPI benchmark for fast assessment of multithreaded communication performance," in *2019 IEEE/ACM Workshop on Exascale MPI (ExaMPI)*. IEEE, 2019, pp. 1–11.

[41] R. Thakur and W. Gropp, "Test suite for evaluating performance of MPI implementations that support MPI_THREAD_MULTIPLE," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2007, pp. 46–55.

[42] D. Doefler and B. W. Barrett, "Sandia MPI microbenchmark suite (SMB)," *Technical report, Sandia National Laboratories*, 2009.