SAND2016-2383C

RMA-MT: A Benchmark Suite for Assessing MPI Multi-threaded RMA Performance

Matthew G. F. Dosanjh, Taylor Groves, Ryan E. Grant, Ron Brightwell
Center for Computing Research
Sandia National Laboratories*
Albuquerque, USA
{mdosanj,tgroves,regrant,rbbrigh}@sandia.gov

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

***Abstract*—Reaching Exascale will require leveraging massive parallelism while potentially leveraging asynchronous communication to help achieve scalability at such large levels of concurrency. MPI is a good candidate for providing the mechanisms to support communication at such large scales. Two existing MPI mechanisms are particularly relevant to Exascale: multi-threading, to support massive concurrency, and Remote Memory Access (RMA), to support asynchronous communication. Unfortunately, multi-threaded MPI RMA code has not been extensively studied. Part of the reason for this is that no public benchmarks or proxy applications exist to assess its performance.**

The contributions of this paper are the design and demonstration of the first available proxy applications and micro-benchmark suite for multi-threaded RMA in MPI, a study of multi-threaded RMA performance of different MPI implementations, and an evaluation of how these benchmarks can be used to test development for both performance and correctness.

I. Introduction

One-sided communication is a promising communication mechanism for extreme scale systems due to its ability to decouple data movement from synchronization. Similarly, multi-threaded communication is a promising mechanism for dealing with the increasing parallelism expected at Exascale. However, the combination of these two mechanisms has not received much attention, despite the fact that they may be deployed together in future systems.

The combination of one-sided communication and multi-threading is particularly attractive because of synchronization costs. In particular, one-sided communication decouples the transport of data from synchronization, while multi-threaded program performance can be heavily impacted by unnecessary synchronization. By coupling the two, a high-performance program can potentially synchronize only when necessary, using the shared memory space exposed through an MPI RMA window, as opposed to incurring unnecessary synchronizations resulting from the use of two-sided communications.

This paper described RMA-MT, a suite of micro-benchmarks and mini-applications that are the first to enable the study of MPI RMA with multi-threading enabled (`MPI_THREAD_MULTIPLE`). It then uses the benchmarks and miniapps to demonstrate the multi-threaded RMA performance of a production version of MVAPICH. In addition, it demonstrates how this benchmark suite can and already has been used for development of future MPI releases in the development trunk of Open MPI.

To the best of the authors' knowledge, code utilizing multi-threaded RMA MPI has not been available to the public until now. The RMA-MT benchmark suite offers four micro-benchmarks and three miniapplications to assess the performance of multi-threaded RMA. These benchmarks can be used to both test MPI implementations, as well as provide a basis for the beginnings of performance optimization for MPI multi-threaded RMA. RMA-MT offers bandwidth, latency, and two message rate micro-benchmarks.

The message rate micro-benchmarks are built from the Sandia Microbenchmarks (SMB) [5] and offer two modes of operation to assess performance. First, they provide a peer-to-peer message rate assessment that supports multiple pairs of peers communicating in one direction from a set of origin peers to a set of target peers. Second, they provide a halo-exchange mode in which message rate in a typical halo exchange is measured. The modifications to these benchmarks involved re-architecting the benchmark to be multithreaded and replacing the two sided communication with one-sided calls. The bandwidth and latency tests are heavily modified versions of MPI multi-threaded tests by Thakur and Gropp [17], which add in RMA capabilities. The modifications to these microbechmarks included re-architecting them to use RMA and data verification to explore the correctness of the implementation.

In addition, the suite provides three multi-threaded RMA mini-apps: HPCCG, MiniFE and MiniMD. These mini-apps represent key portions of real applications used in production today. The miniapplications required overhauling their communication phases to use RMA-MT. They are the first multi-threaded RMA mini-apps of which we are aware, and their design demonstrates how such codes could be adapted to take advantage of multi-threaded MPI RMA. As no significant publicly available codes operate in such a manner, this provides the community an important new foundation for testing and reference purposes.

*Sandia National Laboratories s a multiprogram laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

This paper is structured as follows: Section II reviews MPI RMA and multi-threading modes available in MPI, providing background for this work. Section III discusses the miniapps and micro-benchmarks in depth, detailing their design and the considerations taken into account during the design process. Section IV expounds on our experimental design and environment for running the evaluation section of this paper. Sections V and VI show the results of testing with the micro-benchmarks on a large production system. Section VII concludes the results portion of the paper with an assessment of three mini-apps; HPCCG, miniFE and miniMD. Section VIII provides details of the related work. Finally, Section IX draws conclusions and discusses plans for future work.

II. Background

A. MPI RMA

MPI first provided a one-sided communication interface in the MPI 2.0 specification [10]. It provided three synchronization methods for RMA: `MPI_Win_fence`, `MPI_Win_lock`, and "Post/Start/Complete/Wait" (PSCW). MPI RMA works by allowing one-sided put, get, and accumulate operations on shared windows of memory during an exposed time period, or epoch. Synchronization must occur between different epochs to ensure that the memory window is in a determinant state. This synchronization is key to enable reasoning about the current content of memory and its use in an application. The three provided synchronization methods can be divided into two basic types, active target synchronization and passive target synchronization. `MPI_Win_fence` and PSCW both require the target of the RMA operations to participate in the operations through a call to fence or Post/Wait calls, and are therefore active target synchronization methods. `MPI_Win_lock` does not require that the target call lock, only the origin, and therefore is a passive target synchronization method.

The 2.0 RMA approach was somewhat lacking in flexibility. Because of this, the one-sided (RMA) interface was updated in the MPI-3 specification releases [13]. `MPI_Win_lock_all` was introduced to solve two problems. First, MPI-2 required that when using locks only one target could be locked at any given time. This was resolved with `MPI_Win_lock_all` as it allows an origin to obtain a *shared* lock on multiple targets at the same time. Second, finer grained data movement synchronization was provided through the `MPI_Win_flush` call that allows for assurance that remote operations on the window were complete. Also added in MPI 3.0 were request generating RMA operations (`MPI_RPut` etc.), new memory models (unified vs. separate), and new window types. The new window types enabled windows that can attached to memory after creation, windows with memory allocated by MPI, and shared memory windows. This paper focuses on the basic operation of RMA in MPI with multi-threading and as such does not explore memory models or window types in great depth. The interested reader is referred to [3] for more information on these RMA features and a history of their development.

B. MPI Multi-threading

MPI provides several threading modes. The default threading mode, `MPI_THREAD_SINGLE`, requires the guarantee that each process has only one execution thread at all times. `MPI_THREAD_FUNNELED` relaxes the single thread requirements by allowing multiple execution threads but requires that only one thread, specifically the one that called `MPI_Init_thread`, be the only thread that can make MPI calls. `MPI_THREAD_SERIALIZED` further relaxes guarantees by allowing multi-threaded processes and allowing any thread to call MPI but guaranteeing that only one thread at a time can make MPI calls (serialization is assured outside the MPI library). Finally, MPI provides `MPI_THREAD_MULTIPLE` which allows for multi-threaded processes and any thread may call MPI and they may do so concurrently. Some MPI implementations opt to treat the single, funneled, and serialized threading models similarly, as they all guarantee that only a single thread is in the MPI library at any given time.

MPI has provided `MPI_THREAD_MULTIPLE`, beginning in MPI-2.0 [10]. `MPI_THREAD_MULTIPLE` is not widely adopted, and consequently the multi-threaded mode of MPI is not yet heavily tuned in MPI implementations. MPI RMA with multi-threading is not widely used due to a lack of benchmarks and application code utilizing the combination of methods. However, with new proposals to allow for more exposure of threads to MPI, such as the MPI Endpoints work [4], [15], the use of multi-threaded MPI may increase. RMA is a promising communication mechanism for future extreme scale systems; therefore, it is reasonable to predict that multi-threaded RMA may be used in future MPI programs. This paper seeks to provide, to the best of the authors' knowledge, the first publicly available MPI RMA multi-threaded micro-benchmarks. These micro-benchmarks will provide the foundation to begin optimizing RMA thread multiple as application developers explore alternative MPI communication and threading models in the future.

III. Benchmarks and Mini-applications

This work introduces four micro-benchmarks and three mini-applications to evaluate different aspects of RMA performance and how it affects application performance. In this section, we describe the various elements of the resulting suite.

A. Benchmarks

The RMA-MT test suite includes four micro-benchmarks:

- latency,
- bandwidth,
- single direction message rate, and
- halo exchange message rate.

The goal of these micro-benchmarks is to measure the performance difference between multi-threaded RMA operations using the default locking scheme of MPI and a reduced locking

scheme. For each benchmark, four different synchronization methods (fence, PSCW, lock/unlock, and lock_all/unlock_all) and two RMA operations (Put and Get) are explored. Additionally, these micro-benchmarks allow evaluation of the effectiveness of multi-threaded RMA operations for a varying thread count and message size.

The RMA synchronization methods used in the micro-benchmarks cover all four common methods: fence, lock/unlock, lockall/unlockall, and post/start/complete/wait. It is important to note that RMA synchronization cannot be called on the same window from multiple threads. This is because the RMA synchronization is done at per-rank level. To avoid calling these synchronization methods multiple times per rank, thread-level synchronization is used. When each thread is launched, it updates a simple counter and waits for a broadcast from the parent thread, which signifies that all threads have been created and are ready to begin message transfer. Once each Put/Get thread is waiting, the original thread begins the timer, broadcasts to the Put/Get threads to continue, and runs the RMA synchronization. This method of timing is used to avoid measuring the extra time involved in creating and starting threads. While this is more idealized than would be expected in real applications, the overall thread creation overhead should be relatively small when using a thread pool for performing communication.

1) Latency: RMA operations are not ideal for latency operations due to the overhead of synchronization and that latency tests measure the round trip time of a single message. Despite these shortcomings, a simple multi-threaded latency test is included in the RMA-MT benchmark suite, which provides some insight into the impact multiple threads has on message latency.

For this benchmark, each thread is launched and waits for a broadcast from the parent thread before beginning a single data transfer. This removes any artifacts of initializing the threads. After the call to the non-blocking data transfer, each thread waits for an additional broadcast. Receipt of the second broadcast signifies that all child threads have completed a data transfer and that the initial thread has completed the RMA synchronization.

2) Bandwidth: The bandwidth micro-benchmarks evaluate the potential bandwidth for different RMA synchronization schemes with a varying number of threads. The tests perform a large number of put or get calls between synchronization calls and bandwidth is measured over all iterations. For RMA operations, bandwidth is important to typical use cases because multiple data transfers can utilize a single RMA synchronization, amortizing the cost. The RMA-MT bandwidth micro-benchmarks do not "warm-up" the caches before commencing. Therefore, the resulting average bandwidth reflects this warm-up penalty.

Similar to the latency test, each thread launched by the parent thread waits for an initial broadcast, which signifies that RMA synchronization has occurred. This also signals that all data transfer threads have been launched and are ready to transfer data. Once receiving the broadcast, each data transfer thread performs multiple iterations of put or get to the shared target buffer offset by its thread ID. Following the completion of the data transfer operations, each thread waits for a second broadcast, signifying the completion of a closing RMA synchronization.

3) Message Rate: Message Rate is a subset of the RMA-MT micro-benchmarks, based on the Sandia Micro-benchmarks [5]. Single threaded, two-sided versions of the SMB's have been used in past work [1], [2]. These tests look at different message sizes, peer counts, and two different communication patterns. For this work, applicable communication patterns were extended to evaluate RMA synchronization methods, RMA transfer methods and multiple threads. Communication patterns dealing with two-sided specific communication were not relevant to RMA communication and were not extended. The synchronization methods in these tests are fence, lock, PSCW, and lockall. The lockall implementation calls flush after every transfer operation, to provide multi-threaded progress.

a) Message Rate (Single Direction): The single direction communication pattern looks much like the bandwidth test. It starts up sender ranks that communicate with a paired receiver rank. The primary difference between the two is that single direction uses larger number of ranks. It uses these ranks to test the communication of group of nodes, rather than being limited a single pair.

b) Message Rate (Halo Exchange): The halo exchange test emulates application behavior by implementing a commonly used communication pattern. This test has every rank transfer data to a number of neighbors. In the default case, the benchmark communicates with six neighbors. Due to it's prominence in HPC applications, three of the four two-sided Sandia Micro-benchmarks use this communication pattern, with different variations in the manner and order in which sends and recvs are posted. For the RMA-MT versions of the halo exchange tests, only one version was needed to map to RMA, since RMA doesn't have an unexpected message equivalent.

B. Miniapps

This subsection presents the modifications made to a subset the Mantevo Suite [9]. We focused on three Miniapps: HPCCG, MiniFE, and MiniMD. These were selected to stress the diversity of problems that can use RMA and to stress the RMA components of an MPI implementation in different ways. HPCCG was implemented using the Lock_all/Unlock_all to test the most recent synchronization method. This was added in MPI 3.0 to support passive target RMA. MiniFE and MiniMD both use Fence as it fits well with the design of those miniapps and was performant in the microbenchmarks tests, especially for MVAPICH.

1) HPCCG: HPCCG is a conjugate gradient code focusing on the sparse iterative solver. It is designed to be very scalable and is approximately 3100 lines of code. It uses a halo exchange communication on a 27-point stencil. Therefore,

this communication pattern is somewhat similar to the message rate halo-exchange micro-benchmark, however this mini-app performs calculation and only communicates at message sizes that are relevant to the computation. In order to adapt HPCCG to use RMA data transfers with multi-threading, each ranks spawns a communication thread per neighbor in the halo-exchange. Each process creates a thread for each of the neighbors it needs to communicate with and then uses that thread to drive the traffic solely to that neighbor. This means the number of threads created is not an independent variable for the results shown for this mini-app. The message size used in the `MPI_Put` is the same as those used in the `MPI_Isend` in the two-sided version of the mini-app. HPCCG uses lock_all/unlock_all synchronization semantics.

2) MiniFE: MiniFE is a finite elements code and is similar to HPCCG because it focuses on a similar problem, but it has substantially more features. The code is around 8000 lines. Its main loop, much like HPCCG, is a conjugate gradient solver. Again, `MPI_Put` calls replace the `MPI_Isend`. MiniMD uses fence synchronization semantics.

3) MiniMD: MiniMD is a molecular dynamics code focused on recreating the behavior of LAMMPS. The code is under 3000 lines. It's limited to Leonard Jones pair interactions. MiniMD uses two communication phases per iteration. The first being a forward communication, and the second being a reverse communication. In both cases, we have replaced the `MPI_Isend` call with `MPI_Put`. MiniMD like MiniFE uses fence synchronization semantics.

IV. Experimental Methodology

All of the tests were run on a Skybridge production cluster, which consists of 1,848 dual-socket nodes (totaling 29,568 cores). Each node contains 2X Intel E5-2670, 2.60GHz, 8-core processors with 64 GiB (32 per socket) of DDR3-1600, memory. Each node is connected by a Qlogic onload 4X QDR IB interconnect across a Fat Tree topology. The fabric utilizes three 648-port core switches and 108 36-port edge switches (both Qlogic).

There are two versions of MPI used for the tests. For MVAPICH, we used the 2.1 release downloaded from the official website. For Open MPI, we used a copy of the v2.x branch of the ompi-release candidate development repository on GitHub pulled on October 20th 2015. Both were compiled using Intels 15.0.4 compilers, and unless noted otherwise, were compiled with THREAD_MULTIPLE support. MVAPICH was compiled using the ch3:psm netmod, while Open MPI was given the runtime flags to specify the use of the openib Byte Transfer Layer (BTL), due to development issues with the PSM Message Transfer Layer (MTL).

All versions of the micro-benchmarks used many hundreds of iterations within the test. Each test was run 10 times. The results presented are the average of those 10 runs in each figure in Sections V and VI. All figures shown include vertical bars, although some may not be visible due to small standard deviations. For the miniapp runs, 3 runs were performed. All results are the average of those 3 runs, with applicable error bars for the standard deviation. While thread creation could be expected to introduce variance that would result in larger standard deviations than presented in the following sections, the overheads of thread creation and joining are not included in the performance results of the micro-benchmarks. As the overheads due to thread creation/destruction can be highly variable depending on the approach to threading used. For example, thread pools have lower overheads than on-demand creation/destruction of threads.

V. Single Threaded Results

Single threaded comparisons between one-sided and two-sided communication in MPI have been explored before. We primarily include the results of figure 1 to inform the reader of the baseline performance values for the system under test.

In these figures, we reduce the amount of information displayed by only presenting the best performing single-thread synchronization methods for one-sided communication. When reviewing the bandwidth performance of MVAPICH, the best synchronization is Lock_All, which has 2.1% greater throughput on average across all message sizes than the next best performing synchronization method (Lock). When comparing maximum throughput, Lock_All has a 4.0% increase over the next best method (Lock). In the case of Open MPI, the best performing synchronization in terms of bandwidth is PSCW, which is 1.3% greater on average across all message sizes than the next best method (Lock). The maximum throughput of PSCW is 1.4% greater than the next best synchronization method (Lock).

While the differences between one-sided synchronization methods are small for single threaded communication, we see more significant differences when comparing one-sided versus two-sided bandwidth. There are sudden dips in bandwidth for all the series, with the exception of Open MPI PSCW, as different eager/rendezvous thresholds are activated. The Open MPI revision used for these tests implemented RMA using two sided network calls and has since been updated. Of all the single thread techniques evaluated, Open MPI achieves the best peak bandwidth using one-sided PSCW, just surpassing 3 GiB/s.

In the case of the single threaded latency results, again, we only display the best performing synchronization method. For MVAPICH this is PSCW, which has 2.6% decrease to latency on average than the next best performing synchronization method (Fence). When comparing minimum latency, PSCW has a 5.2% decrease over the next best method (Fence). In the case of Open MPI, the best performing synchronization is also PSCW, which saw a decrease to latency of 12.4%, averaging across all message sizes than the next best method (Fence). The minimum latency of PSCW is 23% less than the next best synchronization method (Fence).

While our benchmarks sample message sizes at powers of two, we match the sampling of the original multi-threaded MPI benchmarks [17], which lack samples between 1 and 16 Bytes. In both cases of MVAPICH and Open MPI, the latency of one-sided operations is significantly worse than the latency of two-

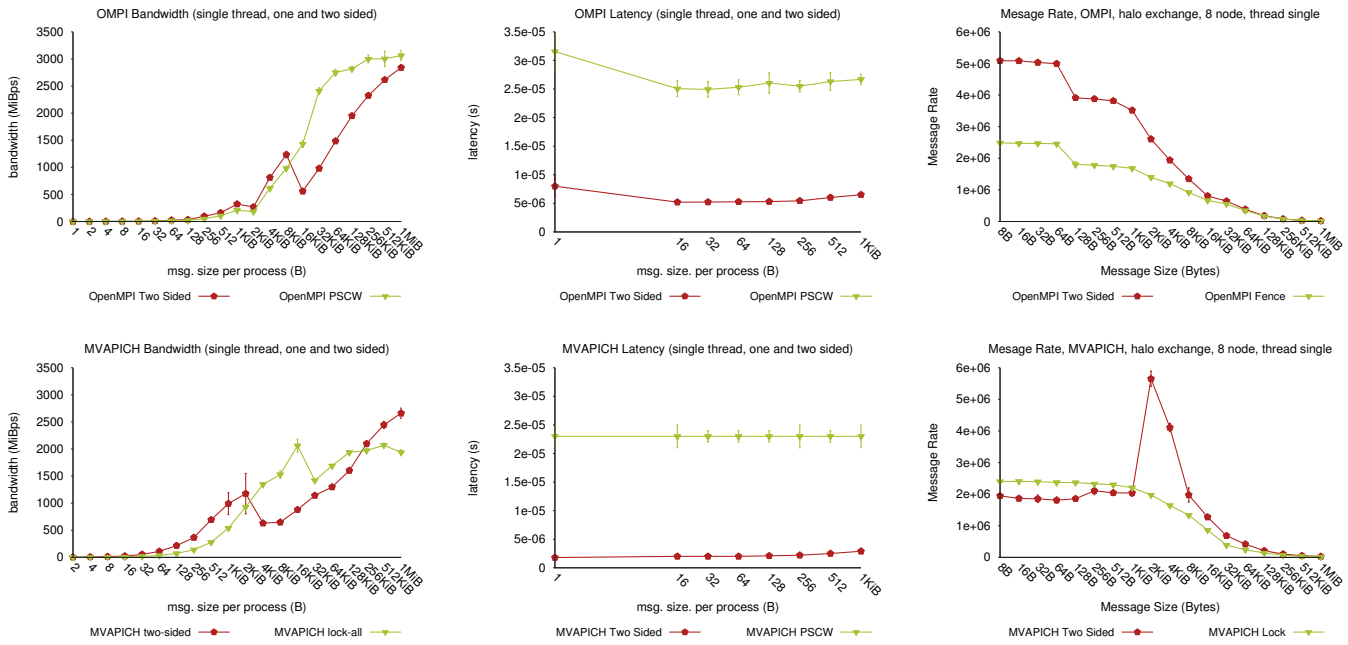

Fig. 1. Single threaded one sided and two sided bandwidth, latency, and message rate of MVAPICH and Open MPI for varying message size.

sided operations, which is somewhat expected as the maturity of the one-sided code in MPI implementations is significantly less than that of the two-sided communication code path. In these experiments the best observed latency was seen in two-sided MVAPICH (1.8μs).

For the message rate halo exchange results, we also only display the best synchronization method. For Open MPI, the best was Fence, which did an average of 1.7% better than PSCW and 12.9% better than Lock. For MVAPICH, Lock showed the best performance, doing 12.1% and 11.8% better than Fence and PSCW respectively. One anomaly in these results is the spike in MVAPICH Performance at 2KiB message size. This is also observed in Section VI and we will discuss it further there. For messages smaller than 2KiB, MVAPICH provided one of the few instances of RMA message rate performance exceeds the two sided baseline. For those message sizes, MVAPICH Lock does an average of 21.4% better than the baseline.

VI. Multi-Threaded Benchmarks

In this section, we illustrate the use of our benchmark suite and how it can provide insights about the underlying system. We present and analyze the results of bandwidth, latency, and message rate benchmarks for varying message sizes, thread counts and MPI distributions. We have split these results into two separate sections by MPI distribution and caution the reader against making any comparison between the MVAPICH and Open MPI distributions for multi-threaded runs. These results shouldn't be compared for two reasons. First, the evaluated version of Open MPI is not a released version and is currently under development. This version is not currently fully tested and on occasion our experiments fail. Of course, these failed runs have not been included in the performance results. We have included a discussion of these failures at the end of this section to illustrate the ability of our benchmarks to evaluate correctness and functionality in addition to performance. Secondly, the system benchmarked (Skybridge) utilizes Qlogic onload network cards, which utilize the PSM interface in MVAPICH. For Open MPI, because we ran into functionality issues the RMA-MT in the PSM MTL, we used the OpenIB BTL instead. These interfaces represent different levels of optimization for the underlying hardware. Because of this, a comparison between the two distributions may be misleading, and as the goal of examining Open MPI was not to assess performance as much as it was to use the benchmarks and mini-apps to demonstrate their utility in debugging and improving MPI implementations in development.

A. MVAPICH Release

1) Bandwidth: Our results from MVAPICH only reported minor differences in throughput when comparing different synchronization methods. For brevity, we only include the plot for the Lock_All synchronization method because there is less than a 2% difference in average throughput across synchronization methods. As a disclaimer, we should note that the code paths for offload cards in MVAPICH have had more effort put into performance optimizations for one-sided communication, such that synchronization methods become a contributing factor to performance. In figure 2 the results require close examination; we see that the dips and peaks in throughput occur as each series reaches the same per-thread message size. To elaborate, as each thread reaches the point where it sends 16 KiB of data, we see a sharp reduction in throughput. This occurs at 16 KiB for single thread results 32 KiB for two threads, and so on. Overall as we

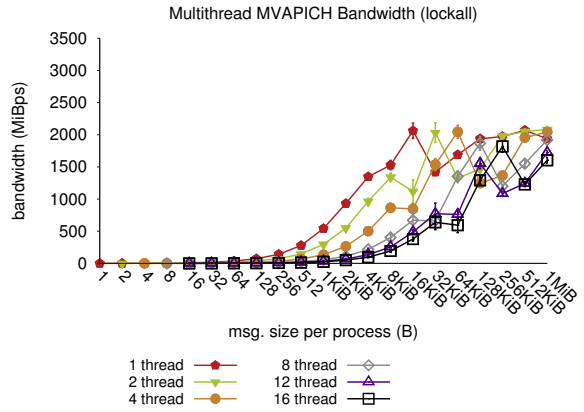

Fig. 2. MVAPICH bandwidth results for one-sided (lockall) communication with varying thread counts.

increase the number of threads to 16, we see a 19% reduction in throughput. This occurs because the overheads of coarse grained locks that become larger as thread counts increase.

2) Latency: The results of figures 3(a) & 3(b) show that for small message sizes, there are significant differences to latency across the four synchronization methods and thread counts. Specifically, we see that PSCW and fence both outperform Lock and Lock_All performance significantly. PSCW achieves 44% and 27% of Lock_All latency at 1 and 16 threads, respectively. For PSCW and Fence, we see a increase to latency of almost 5X or 88 and 90 μs respectively, as we increase the thread count from 1 to 16 threads. Lock and Lock_All see an increase to latency of almost 8X or 303 and 365 μs respectively, as the number of threads increase from 1 to 16. Because there was not a significant difference in bandwidth across the synchronization methods in the previous section, the benchmark suite suggest that either PSCW or fence are preferred for the given system when using MVAPICH.

3) Message Rate: Figure 4 presents the results of the halo exchange message rate benchmark when run under MVAPICH. The single direction version of this micro-benchmark has been omitted for space concerns. This figure shows the total message throughput of the benchmark for each of the synchronization methods. For space concerns only the RMA Get transfer mechanism is shown. It should be noted that the two sided baseline shown in each graph is a special case, as it is run under a single threaded instance of MPI where the multithreading has been turned off at compile. This was done to compare RMA-MT to a current day implementation.

In figure 4(a) we can see the effect of the extra process level synchronization of running under thread multiple. Fence and PSCW were very similar, on average there was 2.3% difference between the two. For small messages under 2KiB, Fence, Lock, and PSCW didn't show significant differences in message throughput. Lock-All on the other hand, performed significantly worse, averaging 49.5% throughput compared to the baseline, while the others averaged 85.9%. Fence and PSCW handled large messages the best out of all synchronization methods, with fence achieving a message rate that was 47.4% of the single threaded baseline.

Figure 4(b) shows the message rate throughput when run on a thread per core. As shown in the graph, large message rate throughput is roughly the same, which makes sense given that the bottleneck quickly becomes the network, rather than the MPI implementation itself. For small messages, there is a large reduction in performance for Fence, Lock/Unlock, and PSCW. Fence, for instance has average throughput of 68.2% compared to the version with one thread.

The most unexpected result from this series of tests is the spike in message rate for the baseline, Lock, and PSCW at 2KiB. While bandwidth is expected to fluctuate in both directions at the message size increases, message rate (which is normalized for message size should go down. The increase is unexpected, but has been confirmed in other work examining RMA message rate for MVAPICH2 [7].

B. Open MPI development branch

1) Bandwidth: This section presents results about the performance for different synchronization methods, thread counts and message sizes of the chosen distribution. The keen reader may observe the lack of Lock_All data in this section. In our experiments, we found the Lock_All synchronization method of this development branch failed too frequently at high thread counts to confidently display results for, therefore it is excluded.

Examining the results of figure 5(a)-5(b), it is evident that for single threads, the bandwidth at 1MiB is extremely close across the different synchronization methods (3065, 3062, and 3077 MiB/s for Lock, PSCW and Fence, respectively). However, we can see that when using 16 threads, synchronization method plays an important role in the observed bandwidth, with Fence seeing a decrease of 573 MiB/s or 21% compared to Lock. Focusing on Fence, as we scale up the number of threads from 1 to 16, we see a decrease to bandwidth of 849 MiB/s or 28%.

2) Latency: The results of Open MPI latency (shown in figure 6(a)-6(b)) tell a different story than the bandwidth results earlier. For small message sizes (under 1KiB) we see that fence and PSCW provide significantly better latency at high thread counts, with PSCW providing the best latency overall. In the worst case (Lock), we see that as we increase thread count from 1 to 16, we see an increase of 235 μs or 6X. In the best case of PSCW, the increase is only 102 μs or 5X. Importantly, our benchmark suite shows that PSCW is preferred when using Open MPI to achieve the both the best bandwidth and latency.

3) Message Rate: Figure 7 presents the results of the halo exchange message rate benchmark when run under Open MPI. It should be noted that the two sided baseline shown in each graph is a special case, as it is run under a single threaded instance of MPI where the multithreading has been turned off at compile. This was done to compare RMA-MT to a current best practices for running MPI. Again, Fence and PSCW performance was very similar (within 3.2% of each

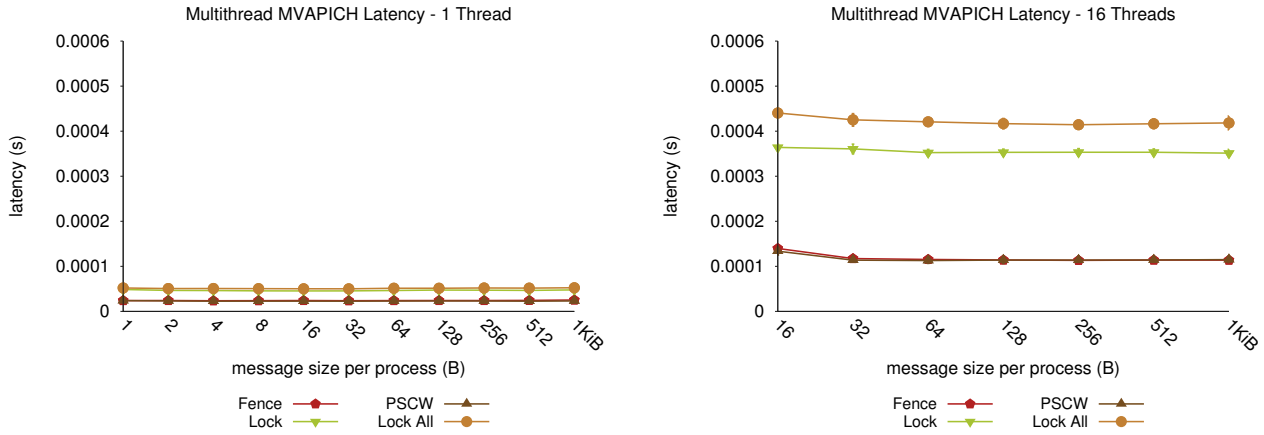

Fig. 3. MVAPICH latency results for various one-sided synchronization methods and thread counts.

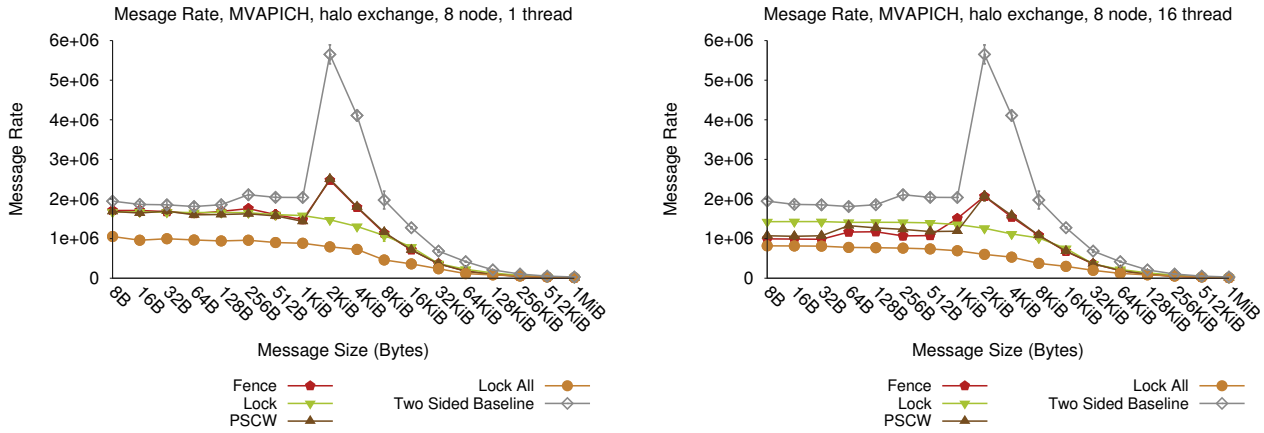

Fig. 4. MVAPICH message rate comparison in a multithreaded context

other). In this graph we can see the effects of using multiple cache lines, when we see the drop in performance from 64 byte to 128 byte message sizes. For small messages, the effects of RMA-MT are clear. Those effects have less impact as we increase message size. For example, Fence performs at 43.4% of the baseline for 8 byte messages. However, once we get up to 1 MiB, it performs at 96.4% the rate of the baseline. Figure 7(b) shows the message rate throughput when run with one thread per core. The trends here are strikingly similar to their single threaded counterparts, averaging a 1.3% difference overall.

4) Development Branch Failures: The Open MPI micro-benchmark results for latency and bandwidth presented here consisted of 840 runs of our MPI benchmark, where each run performs hundreds of one-sided communications across 20 different message sizes. Because we were using a development branch of Open MPI we had a number of runs where errors were detected. These errors were limited to multithreaded runs and are enumerated as follows: three segmentation faults, 22 assertions, and 6 cases where the target or origin buffer did not pass a checksum, representing an error in less than 1% of the runs.

For the message rate micro-benchmarks, we ran roughly 2160 runs across all the combinations of message size, synchronization method, and transfer operation. We observed 81 failures in those runs. It should be noted that the message rate benchmark does more iterations than the other micro-benchmarks and thus has a higher probability of hitting an error. Of the errors we observed for message rate only 12.3% were associated with Get operations, only 16.0% were associated with single threaded runs, and only 19.8% were associated with message sizes less than 64 KiB.

As previously mentioned Lock_All saw a significantly larger number of errors, so was not included in our results. Fortunately, our benchmarks have brought these errors to the attention of Open MPI developers so that they may be fixed before release.

C. Discussion

Many of the results in this section show a degradation in performance when using RMA-MT. This degradation is due to a number of factors, one of the most apparent is thread level synchronization. Ideally, RMA would require little locking within MPI as it doesn't use most of the shared data structures

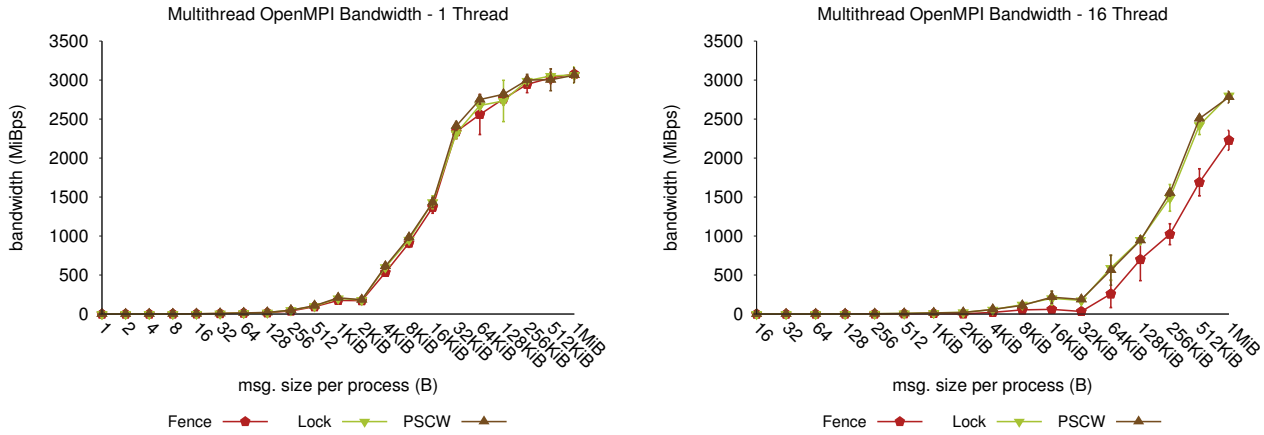

Fig. 5. Open MPI bandwidth results for various one-sided synchronization methods and thread counts.

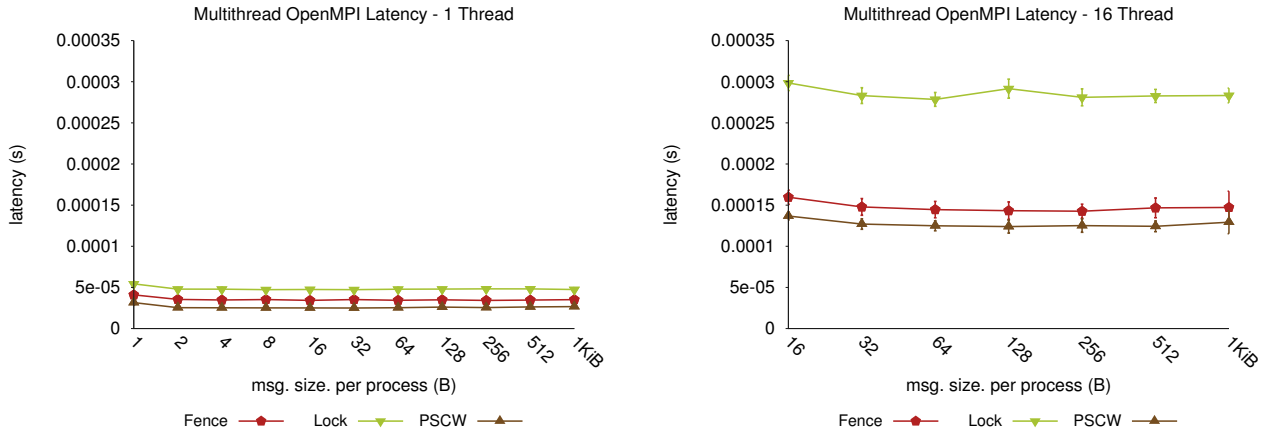

Fig. 6. Open MPI latency results for various one-sided synchronization methods and thread counts.

such as the match list. However, examining the version of MVAPICH used for this study, a lock encapsulates the the entire call into MPI. This is due to multi-threaded RMA in MPI being underutilized and thus unoptimized. The benchmarks in this study provide performance data and RMA-MT capable code that MPI implementations may use to optimize their performance. In addition to synchronization costs, RMA performance has the potential to be degraded by contention for shared memory resources as seen in [6].

VII. MiniApp Results

This section presents the results from running the modified mini-applications. Each test was run with 16 ranks per node, had a weak scaling problem size, and had the problem size adjusted to run for roughly a minute. The tests were run from 16 to 512 ranks using both MVAPICH and Open MPI. from HPCCG. Figure 8 graphs the performance of our tests normalized compared to the performance of the original non-RMA version.

A. HPCCG

For HPCCG, the results in Figure 8 demonstrate the RMA-MT overhead for HPCCG using one thread per communication between communicating rank pairs (neighbors). These runs used a 160^3 per rank problem size resulting in an average runtime of 57.8 seconds for the 16 rank MVAPICH baseline and 56.9 seconds for the 16 rank Open MPI baseline. As shown on the graph, the MVAPICH RMA-MT runs are very close to the baseline; because the standard deviation of these runtimes is often on the order of half a second, this performance difference is not statistically significant.

In contrast, the message rate halo exchange, which has an identical communication pattern, had performance difference was statistically significant performance gap. This is promising as it means that the performance gap left to bridge with application communication patterns may be less than that implied from the micro-benchmark results for future RMA-MT codes. This shows that the RMA-MT approach with Unlock_all/Lock_all is scaling well.

For Open MPI, we see a more significant increase in runtime of up to 2.9% at 256 ranks. It should be noted, that given the

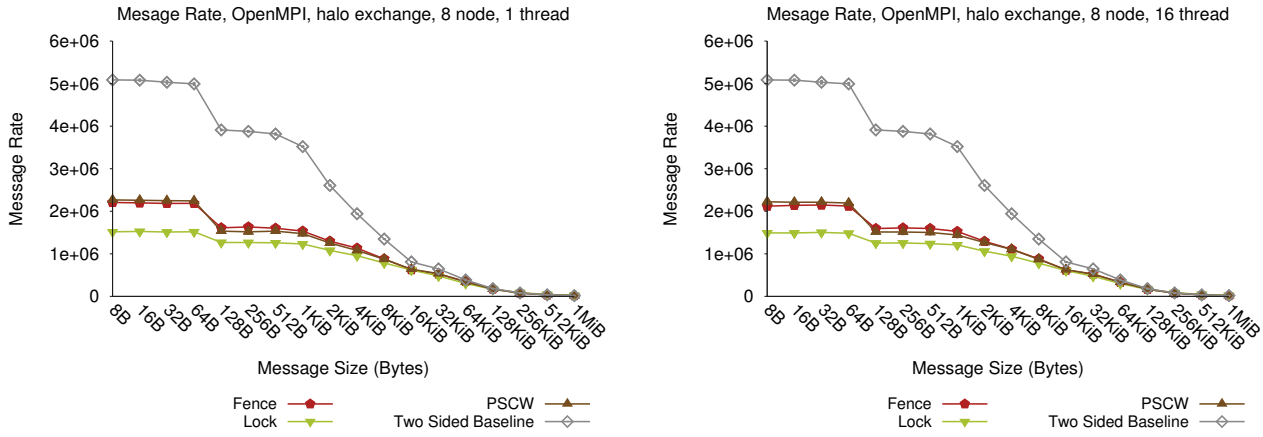

Fig. 7. Single-threaded comparison of the different RMA operations

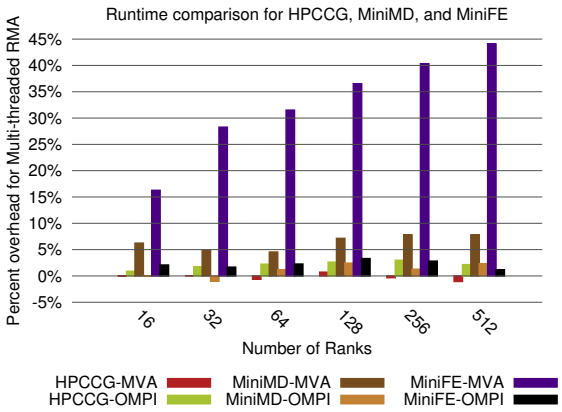

Fig. 8. RMA-MT mini-app run time overhead compared to the regular version

significant amount of errors in lock-all for Open MPI observed in previous sections, this result should be looked at skeptically.

B. MiniMD

For MiniMD, the results in Figure 8 show the RMA-MT overhead for MiniMD using one thread per communication between communicating rank pairs (neighbors). Our MiniMD implementation differs from our HPCCG implementation in that it uses Fence as the mechanic for window synchronization. These runs used a 150^3 per 16 ranks problem size resulting in an average runtime of 54.9 seconds for the 16 rank MVAPICH baseline and 54.5 seconds for the 16 rank Open MPI baseline.

Unlike HPCCG, the MVAPICH RMA-MT test show a large performance degradation from the baseline. This is due to the larger amount of communication calls in MiniMD, and the extra window synchronization required. Given this, we see an overhead of up to 7.8%. For Open MPI, we see a smaller change, of up to 2.4% overhead compared with the baseline.

C. MiniFE

Finally, Figure 8 shows the RMA overhead for MiniFE using one thread for each communication pair. The communication pattern is similar to HPCCG, as they both are proxy apps for conjugate gradient problems; to differentiate them we have used the fence synchronization mechanism for MiniFE rather than lockall. The problem size that used for these tests was a 330^3.

The results for MVAPICH show the highest the RMA-MT overheads of any benchmark, ranging from 16.3% to 44.1%. While this is larger than the other mini-applications, it is not entirely unexpected. Both MiniMD and the message rate micro-benchmarks have significant overhead when using fence as a synchronization method. Because MiniFE uses a substantially larger problem than MiniMD, communication becomes more of a bottle neck. For Open MPI, MiniFE has an overhead of up to 3.0%, much smaller but again larger than the overhead that it had for MiniMD.

VIII. Related Work

Several MPI benchmark suites have been enhanced to support measuring MPI-3 RMA performance. The OSU Benchmark Suite from Ohio State University [14] supports several different measurements associated with MPI-3 RMA operations, including different window creation and synchronization methods. It also supports several benchmarks for the OpenSHMEM one-sided operations. However, it does not currently measure operations in the context of multiple threads. Likewise, the Intel MPI Benchmark suite [12] also has several benchmarks for measuring MPI-3 RMA performance and allows for measuring the impact of the different MPI thread levels, but does not currently measure performance involving multiple threads within an MPI rank.

Understanding the relationship between threads and the performance of communication operations has also been the subject of previous research. A test suite specifically for measuring the performance of MPI communication for multi-threaded processes was presented in [17]. This suite was used to measure the performance of MPI point-to-point and collective communication functions in open source and vendor MPI implementations on three different platforms. More recently, the emergence of many-core processors has motivated

closer examination of the interaction of threading, one-sided operations, and the need for achieving more concurrency from the network. Proposals have been made to better support thread safety and performance optimizations for threaded programs in OpenSHMEM [16], and a proposal for endpoints in MPI [15] seeks to offer enhanced network performance for multi-threaded MPI applications. Similar examinations are occurring for low-level one-sided communication layers as well, including extensions to the GASNet [8] networking programming interface. Several of the issues with extracting more concurrency from the networking hardware and software stack were explored in [11].

IX. Conclusions and Future Work

This paper has presented the design of multi-threaded RMA micro-benchmarks and mini-applications. It has used the micro-benchmarks and mini-app developed to explore the performance of multi-threaded RMA on production systems, providing the first performance numbers available for such MPI usage models. Using the micro-benchmarks it was determined that up to 99% performance degradation can occur when using multiple threads to perform RMA operations for small messages in a current release of MVAPICH. However, there were a limited number of cases where multiple threads aided communication. The mini-apps saw a variety of performance effects; MiniFE and MiniMD both had a performance penalty when using MVAPICH. MiniFE in particular had a sizable penalty of up to roughly 44%. Open MPI saw less of a performance penalty for the miniapps which had a performance penalty of up to 4%. The slowdown using threads was not unexpected when compared to previous thread multiple studies [4] that have found similar multi-threading related slowdowns. However, unlike previous studies, this work has explored MPI RMA in a multi-threading context, which has fewer serialization requirements for ordering guarantees than typical two-sided point to point communications in MPI. This paper also demonstrated the use of this benchmark suite to drive development by testing functionality and correctness, in addition to the performance. This showed that the Open MPI development branch has a number of issues, ranging from triggering asserts to incorrect data transfer. The miniapps also have the ability to test functionality, correctness, and performance as the number of ranks and nodes is scaled up.

Future work in this area concerns using the benchmarks developed to determine if performance enhancements can be made to existing MPI implementations for RMA thread-multiple. Given that RMA does not require the strict ordering requirements of two-sided MPI communication, this approach is expected to be parallelizable in existing implementations with some modifications. Another future work is expanding the mini-app suite to utilize additional synchronization methods such as passive target. Other MPI one sided functions, such as MPI_Accumulate, can be added in future revisions of the benchmark suite. In addition, converting more mini-apps to use RMA thread multiple communication, and utilizing different synchronization, transfer methods, and algorithms, such as those that support asynchronous one sided communication, will provide further insight into the methods of optimizing application code using RMA-MT like communication patterns and help to evaluate where performance improvements could be made in MPI implementations.

The micro-benchmarks used in this paper will be open-sourced for use by the community. When released they will be available from http://www.cs.sandia.gov/smb/.

References

[1] 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*, 28(4):415–424, 2014.

[2] B. W. Barrett, S. D. Hammond, R. Brightwell, and K. S. Hemmert. The impact of hybrid-core processors on MPI message rate. In *Proceedings of the 20th European MPI Users' Group Meeting*, pages 67–71. ACM, 2013.

[3] J. Dinan, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. An implementation and evaluation of the MPI 3.0 one-sided communication interface. *Concurrency and Computation: Practice and Experience*, 2013.

[4] J. Dinan, R. E. Grant, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur. Enabling communication concurrency through flexible MPI endpoints. *International Journal of High Performance Computing Applications*, 28(4):390–405, 2014.

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

[6] T. Groves, R. E. Grant, and D. Arnold. NiMC: Characterizing and eliminating network-induced memory contention. In *30th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2016)*, 2016.

[7] J. R. Hammond, S. Ghosh, and B. M. Chapman. Implementing OpenSHMEM using MPI-3 one-sided communication. In *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools*, pages 44–58. Springer, 2014.

[8] P. Hargrove. GASNet-EX collaboration. https://sites.google.com/a/lbl.gov/gasnet-ex-collaboration, 2015.

[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*, 2009.

[10] S. Huss-Lederman, B. Gropp, A. Skjellum, A. Lumsdaine, B. Saphir, J. Squyres, et al. MPI-2: Extensions to the message passing interface. *University of Tennessee, available online at http://www. mpiforum. org/docs/docs. html*, 1997.

[11] K. Z. Ibrahim, P. H. Hargrove, C. Iancu, and K. Yelick. An evaluation of one-sided and two-sided communication paradigms on relaxed-ordering interconnect. In *IEEE International Parallel and Distributed Processing Symposium*, May 2014.

[12] Intel. Intel MPI benchmarks 4.0. https://software.intel.com/en-us/articles/intel-mpi-benchmarks, 2015.

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

[14] Ohio State University. OSU micro-benchmarks 4.4.1. http://mvapich.cse.ohio-state.edu/benchmarks/, 2015.

[15] S. Sridharan, J. Dinan, and D. D. Kalamkar. Enabling efficient multi-threaded MPI communication through a library-based implementation of MPI endpoints. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2014.

[16] M. ten Bruggencate, D. Roweth, and S. Oyanagi. Thread-safe SHMEM extensions. In S. Poole, O. Hernandez, and P. Shamis, editors, *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools*, volume 8356 of *Lecture Notes in Computer Science*, pages 178–185. Springer International Publishing, 2014.

[17] R. Thakur and W. D. Gropp. Test suite for evaluating performance of MPI implementations that support MPI_THREAD_MULTIPLE. In F. Cappello, T. Herault, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4757 of *Lecture Notes in Computer Science*, pages 46–55. Springer Berlin Heidelberg, 2007.