Evaluating One-Sided Communication on Graph500 with MPI-RMA and OpenSHMEM

Jefferson Boothe, Alan D. George
Department of Electrical and Computer Engineering, University of Pittsburgh
NSF Center for Space, High-Performance, and Resilient Computing (SHREC)
Pittsburgh, PA, USA
{j.boothe, alan.george}@pitt.edu

***Abstract*—While traditionally utilized for two-sided and collective communication, the latest MPI standards support remote memory access (RMA) between processes to enable one-sided communication. This paradigm is typically associated with fine-grained communication and irregular memory accesses. Many graph analysis problems feature such irregular memory and communication patterns, making them a good choice for performance evaluation. Graph500 is a popular benchmark built upon breadth-first search (BFS) on an undirected graph, which is well known for its sparse data accesses and fine-grained communication. In this research, we analyze and compare the scalability of multiple implementations of BFS using MPI-RMA against previously developed OpenSHMEM-based implementations optimized to maximize the benefits of one-sided communication. Additionally, we evaluate these implementations against the state-of-the-art MPI reference code using different numbers of processing elements and various problem sizes. Our experimental evaluation shows consistently improved performance with MPI-RMA over the best OpenSHMEM implementation on Graph500's BFS kernel with scales up to 32 nodes on the Pittsburgh Supercomputing Center Bridges-2 Regular Memory partition and University of Pittsburgh Center for Research Computing (Pitt CRC) MPI Cluster. Due to the nature of graph processing having a higher ratio of communication to computation, the communication latency hiding aspects of one-sided communication could not be fully exploited. While we demonstrate MPI-RMA to achieve $\sim$1.8$\times$ better performance over the MPI reference implementation on 32 nodes when only using 4 cores per node, the reference version was more performant in the majority of configurations tested. It is concluded that while one-sided communication has shown promising performance on some large-scale computing tasks, it remains difficult from a development standpoint to leverage the one-sided benefits on more complex kernels.**

***Index Terms*—Distributed processing, high performance computing, message passing, synchronization**

I. Introduction

Distributed-memory computing clusters continue to grow larger in pursuit of increased performance, but this does not come without challenges. One such challenge is the complexity of communication at scale, which, when performed inefficiently, can become a significant performance bottleneck. As such, developing and exploring efficient parallel communication libraries remains a key interest of the high-performance computing (HPC) community. Two-sided communication is a popular approach to sharing data in distributed computing, in which the sending and receiving processes engage in a synchronized handshake. Conversely, one-sided communication permits the sharing of data with minimal to no synchronization between processes, allowing the freedom for more fine-grained communication.

First introduced in [1], the Graph500 benchmark aimed to direct the attention of the HPC community towards improving performance on emerging large-data informatics problems. These problems differed greatly from existing benchmarks due to the larger data requirements and low spatial and temporal locality of the data. The first kernel of Graph500 and the main interest of this research is BFS. The Graph500 standard provides several predefined graph sizes to test against and requires the number of edges to be 16$\times$ the number of vertices. This research investigates the performance of state-of-the-art one-sided communication libraries when applied to graph analytics.

II. Background

Two commonly used paradigms for communicating between processing elements (PEs) at scale are one-sided and two-sided communication patterns. Two-sided communication requires synchronization between the two processes, as an explicit handshake procedure must be completed. When one process is not available to communicate, the other must typically stall. This process is known as a *blocking* operation. Some libraries offer asynchronous two-sided communication, which eliminates the deadlock possibility of blocking calls. With asynchronous two-sided communication, some handshake synchronization is still required between PEs to share the data, only there is more flexibility in the timings of the procedure. One-sided communication, however, is non-blocking in nature while also not requiring any formal synchronization or handshake between processes. It can be beneficial to analyze the computation-communication overlap of parallel programs, as it is an important technique for latency hiding and overhead reduction [2]. By allowing data transfers to occur without interrupting the receiving PE's computations, one-sided communication grants the programmer new opportunities to overlap computation and communication and improve performance.

This research was supported by SHREC industry and agency members and by the IUCRC Program of the National Science Foundation under Grant No. CNS-1738783.

The MPI standard continues to be the most popular and widely used parallel-programming library across the HPC community [3], [4]. Traditional MPI programs utilize blocking two-sided communication in conjunction with collective communication calls. MPI also offers support for asynchronous, non-blocking two-sided communication calls via probing and waiting [5]. Newer MPI standards introduced Remote Memory Access (RMA) functionality, allowing complete one-sided communication between processes.

OpenSHMEM is a community effort to standardize an API for parallel programming in the PGAS model with portability across machines and environments [6], [7]. OpenSHMEM innately supports one-sided communication as symmetrical memory blocks are created across all PEs with asynchronous access capabilities. Data objects stored in these symmetric memory heaps are *shared*, and any data stored in traditional local memory are *private*. Communication can only occur on shared data.

III. Related Research

This research builds on an array of existing research described in this section. Relevant studies can be categorized into MPI-based research and OpenSHMEM-based research. This section concludes with a discussion of related comparisons and benchmarks previously completed.

A. MPI-Based Research

Graph500 offers a selection of BFS reference implementations using both traditional MPI and MPI-RMA. While most results on the Graph500 utilize custom implementations, these references can still serve as tools for performance evaluation of novel algorithms. A small summary of relevant references follows:

1) Replicated: The *Replicated-MPI* reference implementation features exclusively collective communication calls such as `MPI_AllGather`. It is still a wavefront-based BFS algorithm but is unique due to its lack of traditional, point-to-point MPI messaging between nodes, instead opting for collective communication. As the best-performing reference implementation used in [8], this research utilizes the *Replicated-MPI* version as a baseline for comparison throughout.

2) One-Sided: The *One-Sided MPI* version of BFS utilizes MPI-RMA for one-sided communication calls to perform a wavefront traversal. It is the only reference code to leverage MPI-RMA, however, prior research has shown it to perform poorly [8], [9]. This implementation will be used in this research to compare new MPI-RMA versions to the only one-sided baseline.

B. OpenSHMEM-Based BFS Research

Previous research has been foundational to this research by exploring and developing novel BFS algorithms specifically for the one-sided communication offered by OpenSHMEM. [10] aimed to perform the Graph500 benchmark using OpenSHMEM by directly translating the *One-Sided MPI* reference version from [11] into SHMEM API calls. The authors mainly compared against the *One-Sided MPI* reference implementation, but since it failed to run at larger scales, they could not fully compare the two. There is limited research into developing novel BFS implementations for OpenSHMEM, as opposed to translating existing ones from MPI. In [8], there are five BFS implementations developed from scratch for OpenSHMEM. This research features the *Concurrent-Fence* and *Concurrent-Hash* algorithms, that feature preallocated buffers to facilitate data sharing. The reader is referred to [8] for more details on the algorithms themselves.

C. Previous Comparisons

There have been numerous investigations into the performance of SHMEM on different machines, often compared to traditional MPI or MPI-RMA. In [12], the authors compared basic CraySHMEM communication calls to the corresponding MPI-RMA counterparts using simple kernels with varying message sizes on a Cray XC30. They found the CraySHMEM communication calls to outperform their MPI-RMA counterparts in nearly every case. In [13], various PGAS runtime models were compared using the Parallel Research Kernels [14]. On the Cray XC30 system Edison, they found MPI-RMA and SHMEM performance to consistently lag behind traditional two-sided MPI on the specific kernels tested.

Other research has been presented that demonstrates the potential of MPI-RMA in improving application performance over traditional MPI on graph applications such as graph matching [15]. MPI-RMA and OpenSHMEM are compared on multiple kernels at comparatively low node counts in [16]. This research aims to compare the performance and scalability of each library using a well-established benchmark (Graph500) along with multiple BFS implementations to highlight some advantages and shortcomings of each.

IV. Approach

The main focus of this research is to develop new Graph500 BFS algorithms utilizing MPI-RMA and to benchmark their performance relative to existing standards as well as the original OpenSHMEM implementations found in [8]. For fairness of comparison as well as ease of development, all BFS implementations were built into the Graph500 benchmarking suite developed in [17]. The suite serves as a consistent wrapper for collecting performance metrics and performing validation. This computation is a key component of calculating traversed edges per second (TEPS), the standard comparison metric amongst Graph500 benchmarks discussed further in Section V.

A. Platforms

BFS was evaluated on the the University of Pittsburgh Center for Research Computing (Pitt CRC) as well as the Pittsburgh Supercomputing Center (PSC) Bridges-2 Regular Memory partition [18], [19]. Hardware for both clusters is summarized in Table I. Code on the Pitt CRC was compiled with GCC 8.2.0 and OpenMPI version 4.0.3. Code on PSC was compiled using GCC 10.2.0 and OpenMPI version 4.0.5.

On both Pitt CRC and PSC, OpenSHMEM implementations were run with a symmetric heap size of 3 GB.

TABLE I
PLATFORM SPECIFICATIONS OF [20] [21] ARE PROVIDED.

Platform	Processor	Nodes	Cores per Node	Frequency (GHz)
Pitt CRC	Dual Intel Xeon Gold 6342	136	48	2.80-3.50
PSC RM	Dual AMD EPYC 7742	488	128	2.25-3.40
	Memory (GB)	**L1 Cache (KB)**	**L2 Cache (KB)**	**L3 Cache (MB)**
Pitt CRC	512 DDR4	64	1024	36
PSC RM	256 DDR4	96	512	256
	Interconnect	**Topology**	**Speed (Gb/s)**	
Pitt CRC	InfiniBand HDR	-	200	
PSC RM	Mellanox InfiniBand HDR	Fat Tree	200	

B. OpenSHMEM Implementations

The OpenSHMEM implementations tested in this research are directly derived from those developed in [8] using the code available at [22]. The *Concurrent-Fence* and *Concurrent-Hash* versions were re-implemented inside of the OSB wrapper [17] to ensure fairness of comparison to all other versions as the OSB handles timing and performance metric calculations. These versions had to be altered slightly to run inside the OSB wrapper. Specifically, much of the code was simply removed as it was made obsolete by the OSB wrapper, such as timing and graph generation. Other portions, such as the buffer preallocation and memory cleanup, were rearranged or extended to ensure proper functionality. The fundamental BFS algorithms remain identical to those in [8].

C. MPI and MPI-RMA Implementations

The baseline MPI implementations of *Replicated* and *One-Sided* were ran as-is from the OSB as references. The novel MPI-RMA implementations were developed based on the algorithms used for *Concurrent-Fence* and *Concurrent-Hash* in [8]. The *Concurrent-Fence* OpenSHMEM implementation begins with preallocating as many buffers as possible inside the symmetric memory heap. Since MPI-RMA does not have an explicitly defined region for shared data to exist, instead opting for windows, a custom limit had to be placed on the maximum buffer count to prevent the application from expending *all* of the processor's available memory on buffers. The *Concurrent-Fence* MPI-RMA implementation utilizes `MPI_Get_accumulate` as a replacement for `shmem_int_fadd` and `MPI_Put` for `shmem_char_put_nbi`. The MPI-RMA windows ensured enough ordering between puts that no translation for `shmem_fence` was required to synchronize further. The *Concurrent-Hash* MPI-RMA implementation is similar to the *Concurrent-Fence* version, differing mainly in the buffer structure and use of CRC32 to reduce the required number of `MPI_Put` calls to send a buffer packet from two to one. For more information on the algorithms themselves, the reader is referred to [8].

V. RESULTS

The main metric of the BFS study is traversed edges per second (TEPS), as defined in the Graph500 standard [1]. TEPS can be obtained by dividing the total number of graph edges traversed by the execution time of that traversal. Execution time itself is not emphasized, as BFS execution can vary due to the randomness of graph generation in addition to the randomness of starting point selection in each trial. TEPS serves to normalize the performance results across all trials, problem sizes, and hardware architectures, representing a computing rate similar to those in other common benchmarks (e.g. floating-point operations per second). This section is divided into three different tests of scalability: varying the number of nodes, varying the number of cores per node, and varying the graph size. All graphs feature $16\times$ as many edges as nodes, per the Graph500 standard [11].

A. Variable Node Count

The results in this section consist of data from both Pitt CRC and PSC platforms. The Pitt CRC results were obtained with all 48 cores per node running each application, at various node counts. Data was collected with 2, 4, 8, 16, and 32 nodes or 96, 192, 384, 768, and 1536 PEs, respectively. Fig. 1 plots the median TEPS versus total PEs when tested with a graph of 2^{26} vertices. *Replicated-MPI* achieves the best performance at all node counts on this graph size, with the best performance of 2.67 GTEPS achieved using 768 PEs (16 nodes). The *Concurrent-Fence* MPI-RMA implementation performs nearly as well as the *Replicated-MPI* implementation at lower node counts but is quickly eclipsed with a peak performance of 0.86 GTEPS with 8 nodes. The One-Sided MPI reference implementation was also tested at this scale but performed poorly and the results were omitted from the graph. For example, with 8 nodes, the *Concurrent-Fence* MPI-RMA performed over $800\times$ faster than the peak performance of the One-Sided MPI reference.

Fig. 1 also shows the Pitt CRC results from the same test on a $4\times$ larger graph with 2^{28} vertices. At this problem size, some configurations fail for various reasons, most typically through running out of traditional or symmetrical memory. The cause of this breakdown can generally be explained by fully exhausting available system resources due to the storage demands of the graph, traversal, and/or message buffers. The *One-Sided MPI* reference was unable to complete any trials at this scale. Despite the gaps, data from larger graph tests can still present interesting results and are included. The same trends are present at this scale that can be seen on the smaller graphs. Notably, *Replicated-MPI* continued to be the top performer at higher node counts. However, the *Concurrent-Fence* MPI-RMA implementation was even more competitive at lower node counts and succeeded in running with 4 and 8 nodes, which *Replicated-MPI* failed to do.

Fig. 2 shows the PSC results across varying node counts on the same problem sizes. Notably, these tests were performed using only 64 cores per node of the 128 available. All implementations other than *Replicated-MPI* exhausted

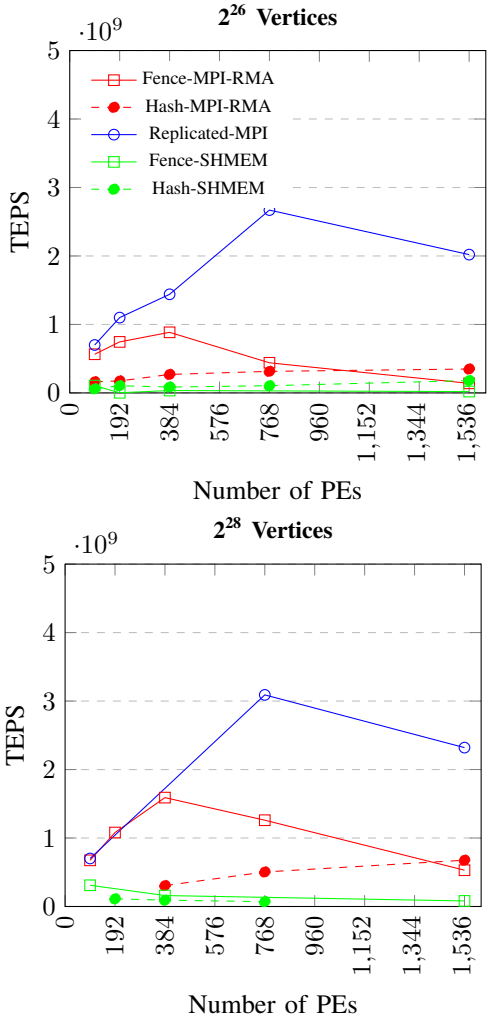

Fig. 1. Median TEPS performance on Pitt CRC with varying node count and all 48 cores utilized on different graph sizes.

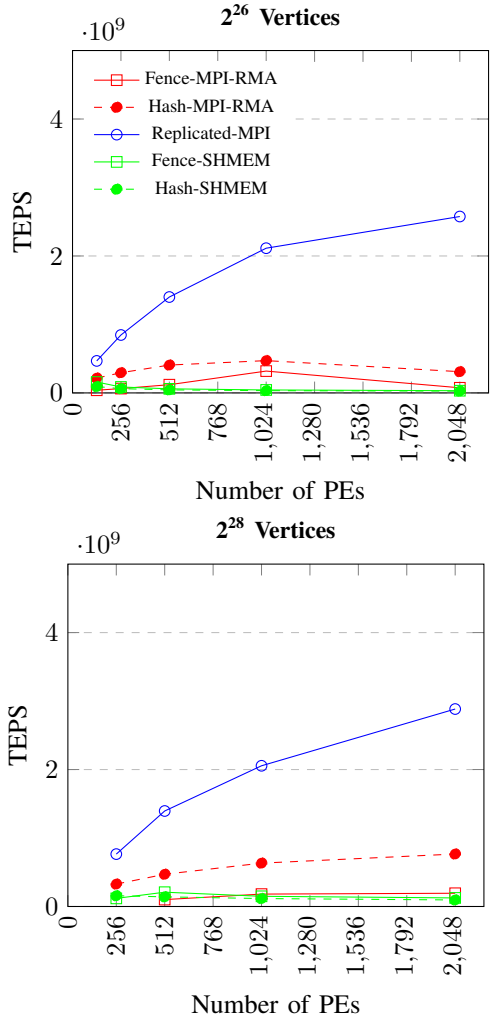

Fig. 2. Median TEPS performance on PSC with varying node count and 64 cores utilized on different graph sizes.

available memory when using all available cores. *Replicated-MPI* achieves the best performance similar to the Pitt CRC results, reaching 2.88 GTEPS with 2048 PEs (32 nodes). The *Concurrent-Hash* MPI-RMA implementation is the best performer utilizing one-sided communication, but still falls behind the *Replicated-MPI* baseline, achieving 0.77 GTEPS under the same conditions. Both OpenSHMEM and the *Concurrent-Fence* MPI-RMA implementations remain significantly slower at all scales tested.

B. Variable Cores per Node

Fig. 3 plots the performance of each implementation versus total PEs similar to previous figures. However, instead of utilizing all 48 cores per node available on Pitt CRC, a variable number were utilized and the node count is fixed at 32. Results were obtained with 4, 8, 16, 32, and 48 active cores per node while remaining cores idled, or 128, 256, 512, 1024, and 1536 PEs, respectively. The *Concurrent-Fence* MPI-RMA implementation outperforms *Replicated-MPI* at the smallest core-per-node ratios but actually decreases in performance with more than 16 cores per node. Comparatively, *Replicated-MPI* improves significantly in performance up through 32 cores per node, where it achieves the best performance across all tests at 3.92 GTEPS. Fig. 3 also shows the same experiment on a graph with 2^{28} vertices. *Concurrent-Hash* MPI-RMA is shown to improve with increased cores per node for both graph sizes, and most trends match across both tests. *Replicated-MPI* again failed to run at low PE counts, but was the lead performer for all successfully completed trials.

C. Variable Problem Size

The final test performed featured a fixed processing count of 32 nodes utilizing all 48 cores each, evaluated at different graph sizes from 2^{22} to 2^{30} vertices. Fig. 4 shows that performance improves for all implementations as problem size increases. This performance improvement is expected, as larger problems allow higher computation to communication ratios. It is again shown that *Replicated-MPI* significantly

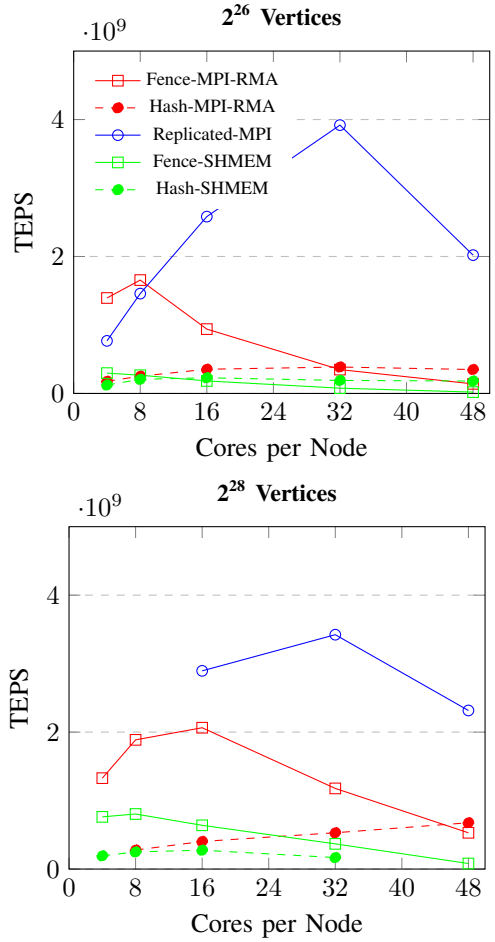

Fig. 3. Median TEPS performance on Pitt CRC with fixed 32 node count and variable cores per node utilized on different graph sizes.

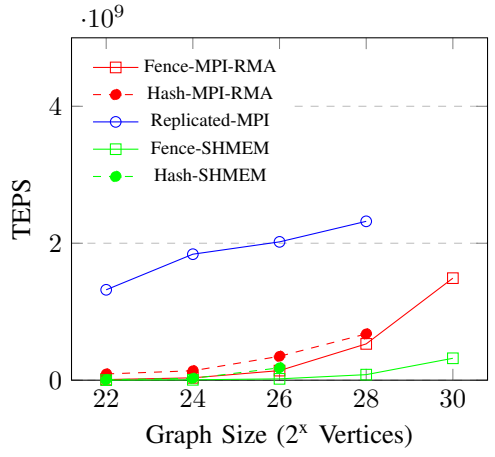

Fig. 4. Median TEPS performance on Pitt CRC across different graph sizes with 32 nodes, 48 cores per node (1536 PEs).

outperformed all one-sided implementations. These results were found to be consistent with all node counts tested. The only implementations to successfully run at scale 2^{30} were the MPI-RMA and OpenSHMEM *Concurrent-Fence* implementations. The successful completion of both *Concurrent-Fence* versions may suggest a smaller memory footprint than those which failed to execute. *Concurrent-Fence* MPI-RMA saw a $2.82\times$ performance increase from size 2^{28} to size 2^{30}, and the SHMEM counterpart saw a $3.95\times$ increase in performance between the same points. On average, the *Concurrent-Fence* MPI-RMA implementation saw a $3.61\times$ increase in TEPS per graph size increase. Comparatively, the *Replicated-MPI* implementation averaged a $1.21\times$ gain in TEPS per graph size increase.

VI. Discussion

All BFS implementations featuring one-sided communication demonstrated relatively poor performance against the two-sided baseline. Notably, the *Replicated-MPI* implementation built upon collective communication calls consistently performs as well as or better than all other versions tested. Collective communication is common in application development and the relative optimizations of such calls are consistent performance concerns of the OpenMPI designers [23]. While the fine-grain control one-sided communication aligns with the behavioral patterns of BFS, the results indicate that at most scales tested, the overhead of facilitating such communication outweighs potential gains. Despite the transfers themselves benefiting from the one-sided communication paradigm, the larger quantity of communication calls required remains a significant contributor to the worsened computation-communication overlap exhibited by all one-sided implementations. *Replicated-MPI*, on the other hand, does not attempt to overlap the computation and communication at all, instead benefiting solely from the minimal overhead of its collective communication calls.

The performance implications of the increased overhead are clearly shown in the variable core-per-node results in V-B. The performance worsens at higher core-per-node counts due to the increased communication requirements, while the computation is too small to provide ample opportunity to hide the communication latency. The reverse effect can be seen as performance increases for all implementations as the problem size grows. A larger problem directly increases the amount of computation per wave of traversal, granting more opportunity to overlap said computation with the necessary communication calls and yield increased performance. This trend coincides with the current Graph500 rankings, which show the majority of top performers executing on massive graphs with 2^{40} or more vertices [11]. For maximum performance, one should aim for the largest problem size possible. Interestingly, this study found that both the MPI-RMA and OpenSHMEM *Concurrent-Fence* applications were capable of working with a larger problem size than the other versions with the given hardware constraints. The cause of *Replicated-MPI*'s failure to execute at lower node counts on the Pitt CRC is uncertain due to

inconsistent behavior. The performance of *Replicated-MPI* on CRC also shows a significant drop when transitioning from 32 to 48 cores per node for both graph sizes. Since *Replicated-MPI* relies heavily on collective communication calls, these findings suggest that optimal performance may be achieved when communicating with a PE count that is a power of two.

All PSC data was gathered using half the available cores per node as every implementation other than *Replicated-MPI* was found to experience runtime errors when utilizing full resources. A notable explanation is that the available memory remains constant with a fixed node count. As in, each node has a set amount of memory available, and increasing the node count increases the total memory of the system. When the number of nodes is held constant and cores per node increases, the total memory remains the same, and the proportional memory per PE actually decreases. The reverse effect of decreasing the cores per node doubled the available memory per PE on PSC, allowing it to run successfully. Introducing better local memory sharing and less duplicate data storage across cores on each node could improve the hardware limitations and allow more resources to be fully utilized. It was also found that all one-sided versions grew in performance relative to problem size at a much faster rate than *Replicated-MPI*. Since the one-sided implementations seek performance gains through overlapping computation and communication, it is also expected that performance grows faster than the *Replicated-MPI* baseline, which utilizes collective calls with no latency hiding. This difference in latency hiding ultimately suggests that with a sufficiently large problem, it is expected that all one-sided implementations will outperform the *Replicated-MPI* baseline due to their computation-communication overlapping. The remaining issue, then, is at what size would this occur, and is it feasible to execute on the given hardware resources.

Another significant factor in communication library performance is the machine itself. While the results found do not necessarily match those found in [8], all of their findings were conducted at node counts beyond the scope of this research, using different SHMEM and MPI libraries with different hardware and interconnects. Additionally, they do not report TEPS, so any direct comparisons are difficult to make. While this study utilized open standards portable to any machine, [8] specifically targets the CraySHMEM library on the Edison supercomputer, a Cray XC30 machine. This library is likely well optimized for Cray specific devices in ways that a portable standard such as OpenSHMEM cannot match. It can be observed, however, that the MPI-RMA versions of *Concurrent-Fence* and *Concurrent-Hash* consistently outperform their OpenSHMEM counterparts in the previous study. Previous studies have found one-sided communication with the MPI-2 standard to be exceptionally poor [24], but there is limited research comparing SHMEM routines with MPI-RMA calls from the newer MPI-3 standard. The results in [12] demonstrate a significant improvement with MPI-3 over MPI-2, but were still outperformed by CraySHMEM when performing different one-sided communication calls. It is worth noting that such a benchmark study does not exist with the specific libraries or hardware utilized in this research. One difference between the algorithms is that the OpenSHMEM implementations require an additional fence call to guarantee ordering when compared to their MPI-RMA counterparts. Since computation-communication overlap is so important to realizing performance gains with one-sided communication on BFS, this additional synchronization requirement of the OpenSHMEM implementations may be a strong contributor to the poor performance. More likely, however, is that the system architecture and libraries featured are not optimized to the same degree as the full-Cray hardware and software stack featured in other studies.

VII. Conclusions

This study investigated one-sided communication for the Graph500 benchmark by expanding on existing OpenSHMEM research to compare with novel MPI-RMA implementations. While one-sided communication offers fine-grain control suitable for BFS, the results showed that the added overhead of facilitating such communication outweighed the benefits at most scales tested across two different computing clusters.

The performance of the one-sided implementations was best when there was ample opportunity to overlap computation and communication. We demonstrated that the *Concurrent-Fence* MPI-RMA implementation achieved $\sim 1.8\times$ better performance over the *Replicated-MPI* baseline on 32 nodes utilizing four cores per node on Pitt CRC. As cores per node increased, the performance of all MPI-RMA and OpenSHMEM versions declined with increased overhead and decreased computation time per PE, restricting opportunities for communication overlap. We also demonstrated that the performance of all one-sided implementations scaled better with problem size compared to the *Replicated-MPI* baseline. However, the problem sizes feasible to test were not sufficiently large for the one-sided communication to outperform the *Replicated-MPI* baseline. Expanding this research to even larger scales, such as tens of thousands of PEs, could potentially illuminate differing scalability at extreme node counts. Similarly, evaluating the performance of each implementation with larger problem sizes may demonstrate an interesting breaking point at which one-sided communication consistently outperforms two-sided.

We also demonstrate that the *Concurrent-Fence* MPI-RMA implementation outperformed the best one-sided reference implementation by nearly three orders of magnitude. While other studies show promising results with one-sided communication, we find that the system architecture and library pairing may play a significant role in expected performance. Evaluating further with different architectures and libraries, such as CraySHMEM utilized in [12], [24], could also result in interesting conclusions about the impact of architecture on the performance of one-sided communication, as many have found SHMEM performance to excel on such architectures. Ultimately, as interest in one-sided communication libraries continues to grow, this research will enable informed design decisions based on performance analysis on well-established benchmarks.

ACKNOWLEDGMENT

This research was supported by SHREC industry and agency members and by the IUCRC Program of the National Science Foundation under Grant No. CNS-1738783. This research was supported in part by the University of Pittsburgh Center for Research Computing, RRID:SCR 022735, through the resources provided. Specifically, this work used the H2P cluster, which is supported by NSF award number OAC-2117681. This work used Bridges-2 at Pittsburgh Supercomputing Center through allocation ELE200001 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by National Science Foundation grants #2138259, #2138286, #2138307, #2137603, and #2138296.

REFERENCES

[1] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray Users Group (CUG)*, vol. 19, pp. 45–74, 2010.

[2] V. Kumar and A. Grama, *Introduction to parallel computing*, 2nd ed. Harlow, England: Addison-Wesley, 2003, oCLC: 1156877846. [Online]. Available: https://openlibrary.org/books/OL18188624M

[3] H. Yu, Z. Chen, X. Fu, J. Wang, Z. Su, J. Sun, C. Huang, and W. Dong, "Symbolic verification of message passing interface programs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 1248–1260. [Online]. Available: https://dl.acm.org/doi/10.1145/3377811.3380419

[4] D. E. Bernholdt, J. Nieplocha, P. Sadayappan, A. G. Shet, and V. Tipparaju, "Characterizing Computation-Communication Overlap in Message-Passing Systems," Jan. 2008. [Online]. Available: https://www.osti.gov/biblio/948730

[5] "MPI: A Message-Passing Interface Standard."

[6] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing OpenSHMEM: SHMEM for the PGAS community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. New York New York USA: ACM, Oct. 2010, pp. 1–3. [Online]. Available: https://dl.acm.org/doi/10.1145/2020373.2020375

[7] M. Baker, B. Chapman, T. Curtis, E. D'Azevedo, and J. Dinan, "OpenSHMEM Application Programming Interface."

[8] M. Grossman, H. Pritchard, Z. Budimlić, and V. Sarkar, "Graph500 on OpenSHMEM: Using A Practical Survey of Past Work to Motivate Novel Algorithmic Developments," in *Proceedings of the Second Annual PGAS Applications Workshop*. Denver CO USA: ACM, Nov. 2017, pp. 1–8. [Online]. Available: https://dl.acm.org/doi/10.1145/3144779.3144781

[9] T. Naughton, F. Aderholdt, M. Baker, S. Pophale, M. Gorentla Venkata, and N. Imam, "Oak Ridge OpenSHMEM Benchmark Suite," in *OpenSHMEM and Related Technologies. OpenSHMEM in the Era of Extreme Heterogeneity*, S. Pophale, N. Imam, F. Aderholdt, and M. Gorentla Venkata, Eds. Cham: Springer International Publishing, 2019, vol. 11283, pp. 202–216, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-030-04918-8_13

[10] E. F. D'Azevedo and N. Imam, "Graph 500 in OpenSHMEM," in *OpenSHMEM and Related Technologies. Experiences, Implementations, and Technologies*, M. Gorentla Venkata, P. Shamis, N. Imam, and M. G. Lopez, Eds. Cham: Springer International Publishing, 2015, vol. 9397, pp. 154–163, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-319-26428-8_10

[11] "Graph 500 | large-scale benchmarks." [Online]. Available: https://graph500.org/

[12] G. A. Negoita, G. R. Luecke, M. Kraeva, G. Prabhu, and J. P. Vary, "The Performance and Scalability of the SHMEM and Corresponding MPI-3 Routines on a Cray XC30," in *2017 16th International Symposium on Parallel and Distributed Computing (ISPDC)*, Jul. 2017, pp. 62–69.

[13] R. F. Van Der Wijngaart, S. Sridharan, A. Kayi, G. Jost, J. R. Hammond, T. G. Mattson, and J. E. Nelson, "Using the Parallel Research Kernels to Study PGAS Models," in *2015 9th International Conference on Partitioned Global Address Space Programming Models*, Sep. 2015, pp. 76–81.

[14] R. F. Van Der Wijngaart and T. G. Mattson, "The Parallel Research Kernels," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. Waltham, MA, USA: IEEE, Sep. 2014, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/document/7040972/

[15] S. Ghosh, M. Halappanavar, A. Kalyanaraman, A. Khan, and A. H. Gebremedhin, "Exploring MPI Communication Models for Graph Applications Using Graph Matching as a Case Study," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2019, pp. 761–770, iSSN: 1530-2075. [Online]. Available: https://ieeexplore.ieee.org/document/8820975

[16] C. Abidi, "Maintaining Communication at Scale with OpenSHMEM," Master's thesis, University of Pittsburgh, Sep. 2022, num Pages: 50 Publisher: University of Pittsburgh. [Online]. Available: http://d-scholarship.pitt.edu/43291/

[17] "OSB: Oak Ridge OpenSHMEM Benchmarks," Sep. 2022, original-date: 2018-03-29T18:24:57Z. [Online]. Available: https://github.com/ornl-languages/osb

[18] T. J. Boerner, S. Deems, T. R. Furlani, S. L. Knuth, and J. Towns, "ACCESS: Advancing Innovation: NSF's Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support," in *Practice and Experience in Advanced Research Computing*, ser. PEARC '23. New York, NY, USA: Association for Computing Machinery, Sep. 2023, pp. 173–176. [Online]. Available: https://dl.acm.org/doi/10.1145/3569951.3597559

[19] S. T. Brown, P. Buitrago, E. Hanna, S. Sanielevici, R. Scibek, and N. A. Nystrom, "Bridges-2: A Platform for Rapidly-Evolving and Data Intensive Research," in *Practice and Experience in Advanced Research Computing*, ser. PEARC '21. New York, NY, USA: Association for Computing Machinery, Jul. 2021, pp. 1–4. [Online]. Available: https://doi.org/10.1145/3437359.3465593

[20] "Computing Hardware | crc.pitt.edu | University of Pittsburgh." [Online]. Available: https://crc.pitt.edu/resources/computing-hardware

[21] "Bridges-2 System Configuration | PSC." [Online]. Available: https://www.psc.edu/bridges-2-system-configuration/

[22] "Openshmem graph500 implementations." [Online]. Available: https://github.com/habanero-rice/hclib/tree/resource_workers/test/performance-regression/full-apps/graph500-2.1.4/oshmem

[23] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, vol. 3241, pp. 97–104, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-540-30218-6_19

[24] C. M. Maynard, "Comparing One-Sided Communication With MPI, UPC and SHMEM," *Proceedings of the Cray User Group (CUG)*, vol. 2012, 2012.