An Examination of a Symmetric Memory Model's Impact on Performance in a Distributed Graph Algorithm

by

Michael Ing

B.S. in Computer Engineering, University of Pittsburgh, 2019

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Master of Science

University of Pittsburgh

2021

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This thesis was presented

by

Michael Ing

It was defended on

March 31, 2021

and approved by

Samuel Dickerson, Ph.D., Director and Assistant Professor
Department of Electrical and Computer Engineering

Wei Gao, Ph.D., Associate Professor
Department of Electrical and Computer Engineering

Thesis Advisor: Alan D. George, Ph.D., Professor, RH Mickle Endowed Chair
Department of Electrical and Computer Engineering

An Examination of a Symmetric Memory Model's Impact on Performance in a Distributed Graph Algorithm

Michael Ing, M.S.

University of Pittsburgh, 2021

Over the last few decades, Message Passing Interface (MPI) has become the parallel-communication standard for distributed algorithms on high-performance platforms. MPI's minimal setup overhead and simple API calls give it a low barrier of entry, while still providing support for more complex communication patterns. Communication schemes that use physically or logically shared memory provide a number of improvements to HPC-algorithm parallelization. These models prioritize the reduction of synchronization calls between processors and the overlapping of communication and computation via strategic programming techniques. The OpenSHMEM specification developed in the last decade applies these benefits to distributed-memory computing systems by leveraging a Partitioned Global Address Space (PGAS) model and remote memory access (RMA) operations. Paired with non-blocking communication patterns, these technologies enable increased parallelization of existing apps. This research studies the impact of these techniques on the Multi-Node Parallel Boruvka's Minimum Spanning Tree Algorithm (MND-MST), which uses distributed programming for inter-processor communication. This research also provides a foundation for applying complex communication libraries like OpenSHMEM to large-scale parallel apps. To provide further context for the comparison of MPI to OpenSHMEM, this work presents a baseline comparison of relevant API calls as well as a productivity analysis for both implementations of the MST algorithm. Through experiments performed on the National Energy Research Scientific Computing Center (NERSC), it is found that the OpenSHMEM-based app has an average of 33.9% improvement in overall app execution time scaled up to 16 nodes and 64 processes. The program complexity, measured as a combination of lines of code and API calls, increases from MPI to OpenSHMEM implementations by $\sim$25%. These findings encourage further study into the use of distributed symmetric-memory architectures and RMA-communication models applied to scalable HPC apps.

Table of Contents

List of Tables

List of Figures

Preface

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 used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

This research also used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562. Specifically, it used the Bridges-2 system, which is supported by NSF award number ACI-1445606, at the Pittsburgh Supercomputing Center (PSC).

This author also wishes to extend thanks to Alex Johnson and Luke Kljucaric for their support and direction in this research, as well as the NSF Center for Space, High-Performance, and Resilient Computing for providing guidance and resources during the development of this research.

In addition, the author thanks Rintu Parja and Sathish Vadhiyar for their work on the MND-MST algorithm and for providing original source code for use.

Finally, this author would like to express deepest gratitude to his family, friends, and teammates for their continued and constant support in his academic pursuits, without which this author would not have made it this far. Their encouragement and love have played a big part in making this research possible.

1.0 Introduction

To maximize parallel processing and acceleration, programmers must minimize overhead and synchronization bottlenecks. For distributed-memory systems the current standard is the Message Passing Interface (MPI) due to its simplicity and support of many communication methods. Using handshake-based point-to-point *send* and *receive* calls and primitive collectives like *broadcast* and *gather*, MPI supports parallelization of numerous kernels and algorithms [3].

The remote memory access (RMA) model introduces new possibilities for further acceleration of distributed parallel apps. Its support for non-blocking and one-sided communication patterns can reduce synchronization bottlenecks in MPI that stem from multiple sequential handshake communications. The increased flexibility afforded by RMA comes with added complexity, requiring the programmer to manually synchronize parallel processes independently to avoid race conditions and invalid memory accesses. Nevertheless, RMA models can lead to increased acceleration by minimizing communication bottlenecks and maximizing the amount of uninterrupted parallel computation for the target of the communication call [4].

Newer versions of the MPI specification support the RMA model by introducing MPI "windows". These windows must be allocated and locked manually to support one-sided communication [4], which limits flexibility. Any performance improvement gained from the model's one-sided and non-blocking communication comes with added program complexity, which may increase development time and detract from the library's utility.

In the last few decades, the concept of distributed symmetric memory, or "SHMEM", has been revisited as an alternative to MPI, resulting in a new specification called OpenSHMEM. Utilizing a partitioned global address space (PGAS) and adhering to the RMA communication model, this specification attempts to support one-sided, non-blocking communication without adding extensive setup overhead or complex API calls. Many OpenSHMEM API calls are modeled after MPI methods, allowing for a low barrier of entry for parallel programmers while still affording increased parallelization [5]. This research contrasts the two-sided

MPI specification to the one-sided OpenSHMEM variant, evaluating RMA acceleration benefits and quantifying any increased complexity or loss in productivity.

This comparison starts at the API level and then extends to the app level using a parallelized graph-processing algorithm based on Boruvka's algorithm [2]. The OpenSHMEM specification is directly compared to MPI by evaluating two different implementations of the algorithm. A focus on overall execution time and productivity provides a basic framework for the continued study and development of the OpenSHMEM specification at multiple levels of complexity.

In summary, this research contributes:

- An evaluation of OpenSHMEM API calls based on existing distributed-communication standards
- A discussion of OpenSHMEM programming techniques that lead to parallel acceleration and corresponding levels of increased complexity
- Analysis of OpenSHMEM optimizations on a Parallel MST app

2.0 Background

The core of this research focuses on evaluating productivity and performance of parallel communication libraries with distributed apps. The concepts presented in this section illustrate the scope of the app with respect to that goal.

2.1 PGAS

The PGAS model imitates the synchronization benefits of a shared main memory using symmetric memory, while maintaining the performance and locality of the distributed-memory model used by message-passing libraries like MPI. A shared main memory allows for simultaneous and overlapping computation, which can be reproduced from symmetric addressing across processors. The distributed-memory model allows for increased scalability, where separate processors can communicate while storing data in smaller local memories [1]. This distributed partitioning of memory allows a larger total space to be represented over multiple cores or processors, rather than requiring a large-scale memory to be used on a single node or multiprocessor.

To simulate and combine the benefits of both of these memory architectures, the PGAS model implements a global address space, local and remote data storage, one-sided communication, and distributed data structures [1]. Global addressing allows individual processors to simultaneously access the same location in their respective symmetric memories. This one-sided communication leads to increased programming flexibility and communication-computation overlap. However, symmetric memory is limited in size. This forces programmers to decide what data needs to be remotely accessible and what can be stored locally (in "private" memory). This creates an efficient compromise between performance and ease of access at the expense of more vigilant design [1]. Support for distributed data structures allows more data to be stored, opening the door for complex program compatibility.

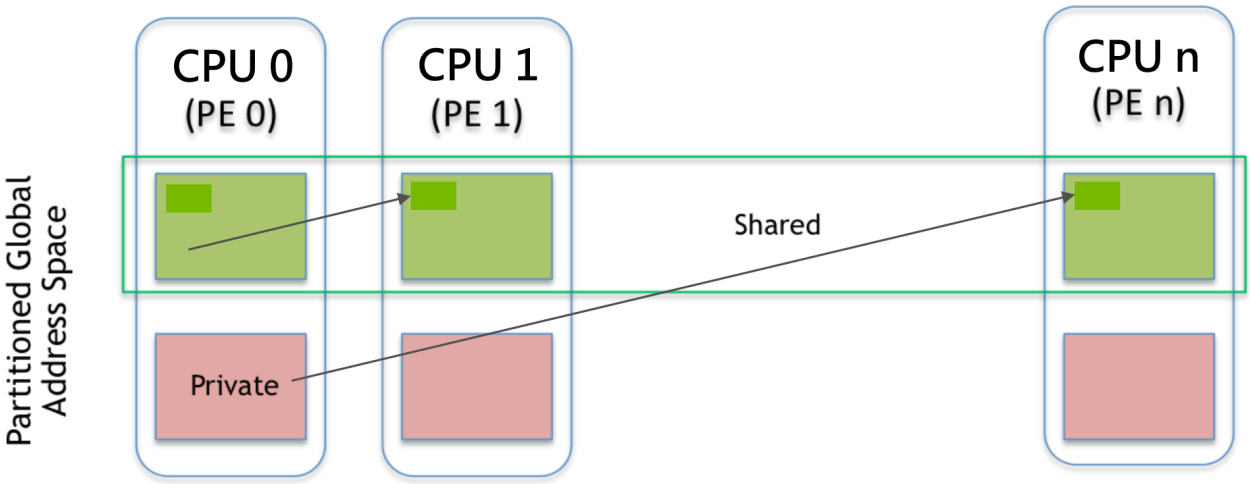

Figure 1: PGAS diagram. [1]

2.2 SHMEM

Originally developed by Cray as a proprietary app interface in the 1990s, SHMEM has been developed into a communication specification used for PGAS programming [6]. By using a PGAS model, SHMEM allows distributed systems to reap the benefits of a "shared" main memory. It also allows for efficient remote data transfer through the use of one-sided point-to-point communication calls like *shmem_put* and *shmem_get* as well as collective calls like *shmem_scatter* and *shmem_gather* [7]. Non-blocking calls like *shmem_get_nbi* and *shmem_put_nbi* provide further means for asynchronous acceleration. Promoted for its ubiquity on existing PGAS systems and structural similarity to well-known communication standards like MPI, SHMEM supports communication-computation overlap via one-sided API-calls [6]. By defining a separate "symmetric heap" in which to store "symmetric variables" remotely accessible by all processors, SHMEM provides the means for simultaneous remote data access, which can lead to uninterrupted computation and scalable parallel acceleration.

In 2010, SHMEM was standardized into the OpenSHMEM specification by the PGAS community, unifying development efforts and expanding its viability for widespread use

[6]. Analogous to the popular MPI specification, OpenSHMEM universalized functions and standardized important aspects of the model including types, collectives, API-call structure and communication protocols. OpenSHMEM has been supported across numerous platforms by multiple libraries, including Cray OpenSHMEMX, OSHMEM, and OSSS-UCX.

The OpenSHMEM specification is under ongoing development and the most current at the time of writing is version 1.5 released in June 2020. This version abstracts away some data structure complexity for collective communications and introduces a teams-based organization style to streamline more complex communication patterns [8]. However, due to its recent release it has yet to be fully supported on existing testbeds and platforms, including the testbeds used for this research. Therefore the app has been developed with OpenSHMEM 1.4, which is supported by OSHMEM and Cray OpenSHMEMX, the specific libraries used to develop and test this work on all three testbeds.

2.3 Minimum Spanning Tree

A weighted graph is a common data structure used in algorithms, composed of a set of vertices (nodes) and a set of weighted edges that connects different vertices. A common analytic of a weighted graph is the minimum spanning tree (MST), and is defined as the set of edges with the least combined weight that connects every vertex in the graph. A fully connected, weighted graph with N vertices would thus have an MST that consists of N-1 edges, the minimum number of edges required to include every vertex. The MST problem has a number of direct applications including network design (telephone, road, electrical circuit) as well as approximation for NP-hard problems in computer science such as the traveling salesperson problem. It also has indirect applications including feature learning, image registration, data storage reduction and cluster analysis [9].

The three most popular solutions to the MST problem are Prim's, Kruskal's, and Boruvka's Algorithms. Prim's algorithm starts with two sets of vertices, those added and not yet added to the MST. It then searches for and adds the edge with the lowest weight to the MST, and adds the new vertices to the appropriate set. This process is iterative, adding the

lowest-weight edge until all vertices are present in the set of added vertices and the chosen edges make up the MST. [10].

Kruskal's algorithm instead starts by sorting all edges in increasing order of weight. The lowest weight edge is then examined to see if its addition would form a cycle. If so, that edge is discarded. If not, it is added to the MST. This process is continued until there are the appropriate number of edges, indicating that the MST is fully formed (N-1 edges for N vertices). [11].

The baseline algorithm used for this research is Boruvka's algorithm, one of the simplest and oldest MST solutions. It starts with multiple small components composed of individual vertices and their lightest edges. These small components are then merged along their lightest available edges to form larger components. This process continues until only a single component remains, which is the MST [12]. The bottom-up nature of this algorithm makes it amenable to parallelization, since vertices can be separately tracked by different processors, and computation can be distributed. The time complexity of Boruvka's algorithm can be improved through utilization of clever data structures and parallelization [9].

3.0 Related Research

The OpenSHMEM specification has previously been explored on the API and app levels, including apps focused on graph processing. This research extends this investigation by analyzing the specification on both levels for an MST graph-processing app, and evaluating its impact on productivity.

3.1 OpenSHMEM API Calls

Jose and Zhang tested OpenSHMEM API call performance across four different OpenSHMEM libraries, including UH-SHMEM (University of Houston), MV2X-SHMEM (MVAPICH2X), OSHMEM, and Scalable-SHMEM (Mellanox Scalable) [13]. They compared point-to-point, collective, and atomic performance on an Infiniband Xeon cluster, scaling up to 1MB in message size and up to 4K processes for collective operations. This work found that MV2X-SHMEM demonstrated consistently lower latencies compared to other OpenSHMEM libraries, as well as a smaller memory footprint per process. Jose and Zhang also compared the performance of two kernels, Heat Image and DAXPY. They found that MV2X-SHMEM again outperformed other libraries, demonstrating consistent execution time improvement that scaled with number of processes.

3.2 OpenSHMEM Graph Processing

OpenSHMEM has been used for graph processing in other contexts, as seen in the work of Fu *et. al* [14] on "SHMEMGraph", a graph processing framework that focused on the efficiency of one-sided communication and a global memory space. To address communication imbalance, computation imbalance, and inefficiency, the SHMEMGraph framework introduced a one-sided communication channel to support more flexible *put* and *get* opera-

tions as well as a fine-grained data serving mechanism that improves computation overlap. The resulting framework was used to test four large web-based graphs on five representative graph algorithms, finding 35.5% improvement in execution time over the state-of-the-art MPI-based Gemini framework [14].

Grossman and Pritchard studied SHMEM-based graph processing through their work on HOOVER, a scalable distributed C/C++ framework for dynamic graph problems [15]. HOOVER focused on flexibility while still maintaining scalability, leveraging one-sided communication and a PGAS memory model. It divided work by splitting graph vertices evenly among processing elements (PEs), which would execute separately at first and eventually coalesce and execute in lockstep as the algorithm iterated. Used for problems such as infectious disease and intrusion detection modeling, HOOVER demonstrated scalable speedup up to 6000 PEs using communication-avoidance techniques and non-blocking communication patterns to maximize computation [15].

3.3 Productivity Studies

To evaluate and compare the productivity of the algorithm using different communication paradigms, multiple metrics are needed. As seen in the work of [16], measuring both overall lines of code (LOC) and number of communication-specific API calls strike a balance between increased complexity and overall workload. Development time has also been used to measure productivity with HPC toolsets as seen in [17], but this metric is more subjective and difficult to measure and compare. The OpenSHMEM specification's growing similarities to MPI further legitimize these metrics, making a direct comparison of productivity more viable and informative.

3.4 Parallel MST

Olman and Mao implement a parallel MST algorithm as a means for solving a Bioinformatics clustering problem based on Prim's algorithm [18]. Using a Fibonacci heap to find the next smallest edge, this algorithm first divides the entire network into subgraphs of equal size in terms of number of vertices. Bipartite graphs are also generated, which bridge subgraphs via intra-vertex edges. An MST is then constructed for each subgraph and bipartite graph in parallel. Finally, all resulting MSTs are merged in parallel to form a new graph, which in turn constructs the overall MST for the original graph. The entire algorithm is tuned to the optimum number of graph partitions.

Bently tackles the parallel construction of MSTs in the scope of VLSI technology, defining an algorithm with asymptotic execution time of *O(V log V)*. This algorithm uses a "tree machine" data structure, which is a mirrored binary tree with nodes for broadcasting, computing, and combining data inputs. Based on Prim-Dijkstra's MST, this parallel algorithm uses the binary tree machine to search for minimum weight edges across multiple vertices simultaneously. As edges are added to the MST structure, vertices are eliminated from the original pool and added to the MST pool until no vertices remain and the MST is fully constructed [19].

Yan and Cheng have developed a system to find MST data structures on distributed processors called Pregel [20]. This system is "vertex-centric", focusing on messages sent between vertices to keep communication simple and efficient [21]. Based on the bulk synchronous parallel model (BSP), Pregel was theoretically able to achieve performance improvements for graph processing apps by increasing the number of parallel communications that could simultaneously execute. However, this approach has inconsistency issues due to varying vertex degree in large-scale graphs, leading to unequal communication backlog and bottlenecks. This led to the development of Pregel+, which added vertex mirroring for message combining and introduced a request-response API [21]. Running Pregel+ against modern competitive graph processing systems like Giraph and GraphLab demonstrated the effectiveness of these two techniques, resulting in reduced communication cost and reduced overall computation time for the new Pregel+ implementation [21].

The algorithm used in this research is based on and uses source code from Panja and Vadhiyar [2], who describe the operation of the parallelized, distributed MST graph algorithm. The algorithm is explained in detail in Section 4.4. Panja and Vadhiyar validate the algorithm's performance compared to Pregel+, and show positive performance improvements for overall execution time on a scaling number of parallel processes from 4 to 16. This work was thus deemed suitable for use as a state-of-the-art scalable distributed parallel algorithm.

4.0 Experiments

This section details the nature of experiments performed, data collected, and optimizations studied. Topics include supercomputing testbeds, API-level experiments, app datasets, and the design and optimization of the app based on the MND-MST algorithm.

4.1 Testbeds

Microbenchmark data was collected on three testbeds. These testbeds are the University of Pittsburgh's Center for Research and Computing (CRC), the Pittsburgh Supercomputing Center (PSC), and the National Energy Research Scientific Computing Center (NERSC). CRC has 2.6GHz dual 10-core Haswell nodes with FDR Infiniband interconnects [22]. The Bridges 2 system on PSC has 3.4GHz 256GB EPYC AMD Nodes with HDR Mellanox Infiniband interconnects [23]. NERSC is a U.S. Department of Energy Office of Science User Facility at Lawrence Berkeley National Laboratory, and uses over 2,300 2.3GHz Haswell nodes each with 128GB of DDR4 memory [24].

All MST app data was collected exclusively on the Cori partition of NERSC, with execution times averaged over 15 executions for each configuration of MST runtime parameters. Correspondingly, the microbenchmark results shown below represent only the data collected on the NERSC supercomputing system. Microbenchmark data collected on all three testbeds are compiled in Appendices A and B. Microbenchmark latencies are averaged over 500 executions. For OpenMP sections, 4 threads were allocated per node.

4.2 API Calls

To frame and analyze results for a larger app, it is important to analyze differences of the baseline, API-level performance. This is done by directly comparing relevant API calls

between MPI and OpenSHMEM. Point-to-point and collective tests are scaled up in message size, and the collective operations are scaled up in number of parallel processes. Microbenchmark tests for both MPI and OpenSHMEM are created by the MVAPICH project from Ohio State University, with minor adjustments made to scale all benchmarks to appropriate sizes [25]. Point-to-point benchmarks were executed using two nodes and scaling from 1 byte up to 4 MB in message size. Collective benchmarks were similarly scaled up to 4 MB, and the number of nodes was scaled from 2 to 32. All API-level benchmarks used one PE per node.

4.3 Datasets

The datasets used for the app consist of large web-based graphs formed by web-crawling [26] and created by the Laboratory for Web Algorithmics. These graphs are undirected, weighted and have significantly more edges than vertices, which supports straightforward vertex partitioning and makes them ideal for large-scale parallel processing and MST calculations. Although not all fully connected, consistent MSTs can still be calculated effectively for execution time comparison. These graphs range in size from 1.8 million vertices to over 100 million vertices, with edge counts reaching nearly 2 billion. These large graphs have execution times on the order of tens of seconds, allowing for greater disparity in execution time at scale. Execution times for MPI and SHMEM implementations can be directly compared because the underlying nature of the algorithm remains unchanged despite which communication paradigm is used. Edges are still processed, removed, and exchanged in the same way, and various implementations differ only in the order and method of communication of edges and components.

4.4 Algorithm

Based on [2], the baseline algorithm of this research is a parallelized version of the classic Boruvka's algorithm for finding MSTs, as described in Section 2.3.

Table 1: Webgraph Dataset Details

Webgraph Dataset (E/V = Edge-to-vertex ratio)					
Name	**Size (GB)**	**Vertices**	**Edges**	**Max Deg**	**E/V**
uk-2014	0.15	1.77e6	3.65e7	6.59e4	20.66
gsh-2015	4.70	3.08e7	1.20e9	2.18e6	39.09
ara-2005	4.90	2.27e7	1.28e9	5.76e5	56.28
uk-2005	7.25	3.95e7	1.87e9	1.78e6	47.46
it-2004	8.80	4.13e7	2.30e9	1.33e6	55.74
sk-2005	15.00	5.06e7	3.90e9	8.56e6	77.00

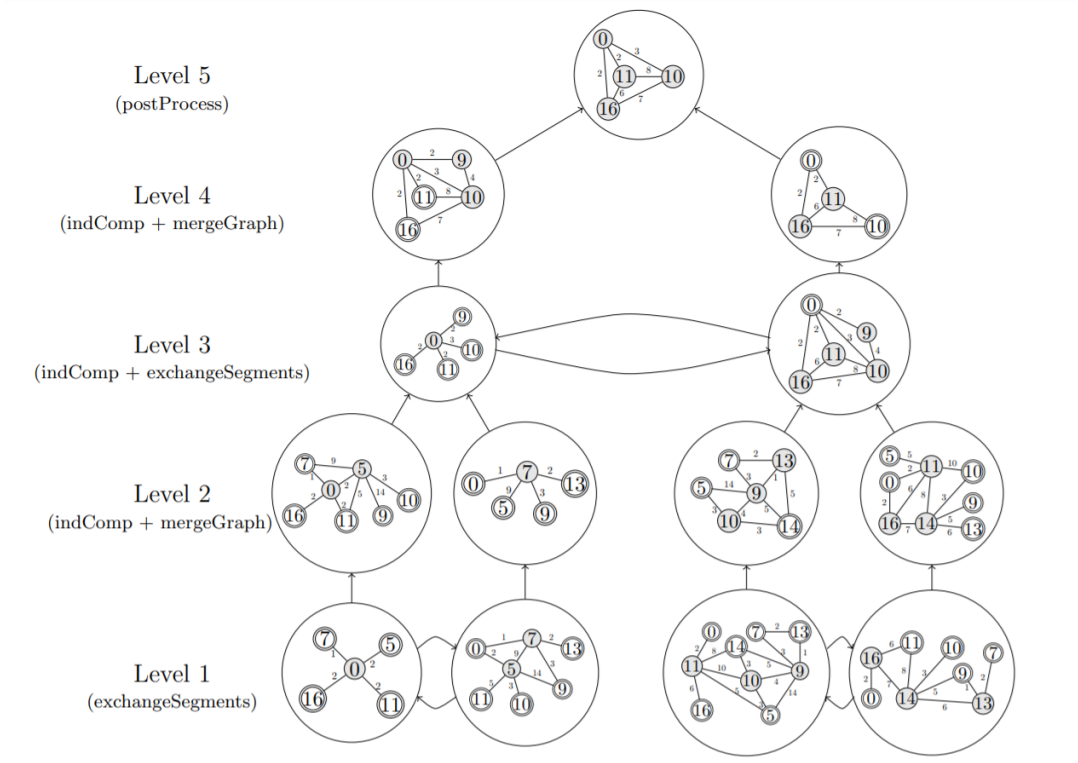

Figure 2: MND-MST algorithm structure. [2]

The parallelized version of the algorithm is split into four major parts: graph partitioning, independent computation, merging, and post-processing, as shown in Fig. 2. During graph partitioning the input graph is read in parallel by each PE and divided into equal parts. All vertices and edges are split evenly among the PEs, with a focus on edge balancing. This partitioning method preserves graph locality while also maximizing parallel computation efficiency. To keep track of edges that span multiple PEs, a list of "ghost information" is maintained by each PE, which consists of the list of edges that are connected to external PEs. Keeping track of this extra information adds memory and communication overhead, but allows independent computation and merging to operate efficiently in parallel [2].

The bulk of the work is performed during independent computation and merging. During independent computation each PE executes Boruvka's algorithm locally, combining as many components as possible to minimize the number of distinct internal partitions. If a lightest edge for a given internal component connects to a "ghost vertex," computation is halted until data is exchanged in the merging step. To improve efficiency, a tolerance threshold is introduced, Θ, which measures the number of MST edges added during each loop iteration. Once the number of new MST edges dips below this threshold, independent computation ceases and merging begins. Θ was optimized to 1e-3 * E [2], where E is the number of total MST edges. This threshold is used to avoid slowdown caused by having large numbers of unresolved components with "ghost edges".

In the merging portion of the algorithm, each PE cleans up local components by removing internal or "self" edges (connecting two vertices within a component) and "multi" edges (heavier edges that were not chosen for the MST). After cleaning up components, PEs recalculate lists of ghost edges to account for any restructuring. Finally, component data is exchanged between PEs in a ring pattern. A small percentage of vertices and edges from each PE are sent to the PE of the next higher rank, updating ghost information and vertex data in turn. This consideration allows independent parallel computation to continue, as new sets of components and edges can be evaluated.

After independent computation and merging, the total MST size is calculated by summing over each PE with a reduction communication. The number of remaining MST edges

(based on the number of vertices) is then calculated and compared against the "MST threshold". Once the number of remaining edges dips below this threshold, post-processing can begin.

The post-processing step combines all remaining components and edges into a smaller number of PEs, where a final round of computation can be done to construct the full MST. This process can be done in one of two ways. With the "single" mode of the algorithm, all data is sent to PE 0, which then does one final round of computation with a Θ of 0 (i.e., all MST edges must be found). The "leader" mode splits PEs into groups of a specified size, and each group combines all data into a "leader" PE, which then runs independent computation again. The group size was set to 4 PEs as consistent with [2] for best performance.

The algorithm is visually presented in Fig. 2. As shown, the initial group of vertices is divided evenly among the processors in the partitioning step. Then independent computation and merging takes place, wherein each processor eliminates edges, combines components, and exchanges some combined vertices and edges with a neighboring processor to continue the process. These steps are repeated until the number of newly added MST edges goes below a new threshold, called the *MST threshold*. Once this threshold is surpassed, all components are combined into a single processor and post-processing occurs. The remaining components are calculated and the full MST is returned.

4.5 Algorithm Variables

Runtime parameters including post-processing mode, MST Threshold, number of nodes, and PE count were tuned during data collection for optimal performance. The post-processing mode was either single or leader. The single mode consists of having each node send all left-over components to PE 0 before performing MST computation on this final PE. In contrast, the leader mode splits PEs into groups of 4, sending leftover components to the leader PE, and then re-running the algorithm to merge and compute the MST. This alternative mode was intended to reduce execution time by parallelizing long post-processing times.

The *MST threshold* determined the point at which component consolidation and post-processing was performed, based on the number of new MST edges. This threshold was optimized to be 0.24, or 24% of the total number of MST edges.

Table 2: ara-2005 Node-PE Configurations

ara-2005 Node-PE Configurations (S: Success, -: Failure)													
		Configuration (Nodes, PEs)											
Nodes		1				2				4			
SH	**CB**	4	8	16	32	4	8	16	32	4	8	16	32
1GB	1GB	S	S	-	-	-	-	-	-	-	-	-	-
2GB	1GB	S	S	-	-	S	S	S	S	S	S	S	S
2GB	2GB	S	S	-	-	S	S	S	S	S	S	S	S
4GB	2GB	S	-	-	-	S	S	S	S	S	S	S	S
4GB	4GB	S	-	-	-	S	S	S	-	S	S	S	S
6GB	6GB	S	-	-	-	S	S	-	-	S	S	S	S
8GB	8GB	-	-	-	-	-	-	-	-	S	S	S	-

Strong scaling was performed by altering the number of nodes and processing elements per job. Nodes were scaled from 1 to 16, and PEs were scaled from 4 up to 64, as the number of PEs is restricted to a multiple of 4. NERSC nodes were limited to 118GB per node, and 64 PEs per node [24]. It was noted in [2] that the algorithm scaled effectively on 4 nodes up to 16 PEs, so data for additional node-PE configurations was collected to further evaluate the scalability of both implementations. Node-PE configurations were also influenced by memory limits and allocations, including that of the private heap, the symmetric heap (SH), and a separate "collective symmetric buffer" (CB) used for SHMEM collective communications. The two symmetric buffers were set before running jobs and were allocated per PE. NERSC memory limitations for individual nodes coupled with large graph sizes required fine-tuning of these parameters to fully execute the algorithm. Multiple webgraphs of different sizes and characteristics were tested to diversify results and draw more robust conclusions about the algorithm and the communication schemes. Graph data can be referenced in Table 1.

Table 3: uk-2005 Node-PE Configurations

uk-2005 Node-PE Configurations (S: Success, -: Failure)													
		Configuration (Nodes, PEs)											
Nodes		1				2				4			
SH	**CB**	4	8	16	32	4	8	16	32	4	8	16	32
1GB	1GB	-	-	-	-	-	-	-	-	-	-	-	-
2GB	1GB	S	S	S	-	S	S	-	-	S	S	-	-
2GB	2GB	S	S	-	-	S	S	-	-	S	S	S	-
4GB	2GB	S	S	-	-	S	S	S	-	S	S	S	S
4GB	4GB	S	S	-	-	S	S	-	-	S	S	S	S
6GB	6GB	S	-	-	-	S	S	-	-	S	S	S	S
8GB	8GB	-	-	-	-	S	-	-	-	S	S	S	-

Configurations for two webgraphs are presented in Tables 2 and 3. Note that the uk-2005 graph is larger in size than the ara-2005 graph, which tends to require larger symmetric heap sizes to execute. Some failures resulted from symmetric memory (heap and the collective buffer) that was too small to handle communications, while others were caused by over-allocation that infringed on private memory. As shown, some node-PE configurations were rendered impossible, as there wasn't enough memory available to support both symmetric memory for communication and private memory for graph data storage. Systems with more available memory per node could allow more extensive configuration testing.

4.6 SHMEM Optimizations

A number of techniques are used to optimize the OpenSHMEM-based app beyond simple one-to-one API call replacement. By leveraging partitioning, non-blocking communication and RMA, SHMEM enables programmers to reduce communication overhead and accelerate

parallel execution without introducing overwhelming complexity. This section details some of the specific SHMEM optimizations used for the baseline parallel MST algorithm to provide a framework for large-scale optimizations for other apps in the future. The first major source of OpenSHMEM optimization occurs during the exchanging of ghost information, which consists of external vertices and their corresponding edges, after independent computation. As each PE could contain ghost information for any other PE, all pairs must be examined and information exchanged. In the baseline MPI approach, this consists of a series of handshake *MPI_send* and *MPI_recv* calls, first exchanging the message size (i.e. the number of ghost edges to be exchanged and updated) before sending the full data structure of vertices and edges to be updated. This process repeats, one pair of PEs at a time, until all information is exchanged. Each PE then locally updates the corresponding data structure to reflect changes in component sizes as well as edges that have been newly removed.

This relatively straightforward communication can be improved with the use of OpenSHMEM. First, the message size can be sent using one-sided *put* and *get* operations followed by a *shmem_wait_until* synchronization API call. While this does not completely remove the handshake from the MPI-based app, each PE can operate independently while sending the message size, which allows for more efficient execution. Second, the ghost information can be communicated via RMA without the need for any synchronization. Since each PE can unilaterally *get* all necessary data, handshaking overhead and slowdown from synchronization are eliminated.

Third, the OpenSHMEM implementation takes advantage of partitioning, which is essentially overlapping communication and computation. Although the message size communication is relatively small (only a single int or long data value), the ghost information itself can consist of thousands or even tens of thousands of edges. Such a large message can be divided and sent between PEs in chunks, each overlapped with the updating of the local PE data structure. Rather than using a single *get* operation to send the entire message, a non-blocking *get* operation of a smaller chunk size is executed. While the smaller non-blocking RMA operation executes, the PE updates the local data structure for the previous data chunk. In this way, communication and computation are overlapped, using a simple

shmem_quiet synchronization call to ensure that the previous chunk of data is transferred fully before being used to update the data structure.

The other prime target for OpenSHMEM optimization is the exchanging of component data during the merging step. In the baseline MPI implementation, sizes of exchanged vertices and edges are communicated for each pair of processors. These sizes are then used to exchange portions of several different data structures between the pair of processors using a series of synchronous send-receive communications. Some local clean-up computation is then performed, copying exchanged information and ensuring that the data structures are properly formatted for continued execution.

The OpenSHMEM implementation avoids the handshake overhead entirely by using non-blocking communication calls as well as RMA. As before, the use of RMA allows each PE to operate independently, retrieving the required information simultaneously. The use of non-blocking API calls allows some local computation to overlap, leading to acceleration. Partitioning is also used to overlap this communication with some of the ending data structure updating and copying. Used together, these techniques take advantage of the thousands of edges that must be communicated between PEs and overlaps that communication with data structure update overhead to maximize the amount of uninterrupted, pure computation. The independent computation step is done locally by each process, so no OpenSHMEM optimizations can be performed. The original MPI algorithm uses blocking communication with no overlap, so both PEs must communicate all data before running computation. The optimized OpenSHMEM implementation uses non-blocking communication-computation overlap, with a pre-defined number of partitions. The data is divided into equal chunks and communicated chunk-by-chunk asynchronously, and each communication is overlapped with computation and later confirmed by a synchronization call (*shmem_quiet*). Although MPI and OpenSHMEM both have the capability for non-blocking communication and computation overlap, the OpenSHMEM implementation benefits from RMA communication calls and fewer lines of code. Non-blocking two-sided MPI still necessitates handshake-based communication, and also requires the use of additional *MPI_Request* and *MPI_Status* objects for synchronization, which adds overhead.

These same techniques are applied to the post-processing step of the algorithm. Data structures are gathered and combined in a similar manner to the merge step, except that they are gathered into a smaller number of PEs for further computation. However, the OpenSHMEM implementation provides further benefits during this step. For the baseline MPI implementation, all communications require handshakes between a pair of processors. For the single mode PE 0 must execute a series of send-receives with every other PE, resulting in a handshake bottleneck. The RMA nature of the OpenSHMEM specification allows each PE to simultaneously *get* data from PE 0 via a series of one-sided communication operations. To support these communications, the OpenSHMEM implementation adds an additional *all-reduce* collective call to first calculate address offsets. At the cost of an extra API call and an extra data structure, this technique removes the handshake bottleneck with PE 0 and allows this entire series of communications to execute asynchronously.

5.0 Results

All data collected are presented in this section, including microbenchmark performance for various API calls and an app-level comparison of OpenSHMEM and MPI. Additional algorithm tuning data and productivity comparisons are also examined. All results shown in this section were collected on the Cori partition of the NERSC system. Additional point-to-point and collective microbenchmark data collected on CRC and PSC are shown in Appendices A and B.

5.1 API Calls

The results of the API-level OSU microbenchmarks executed on NERSC are shown in Figures 3 and 4 and Table 4. All latencies are measured in µs. To provide proper context for the distributed MST algorithm, communication calls that are most often used in the algorithm are presented in these tables, including *get*, *put*, *all-reduce*, and *barrier-all* operations. To compare one-sided and two-sided point-to-point operations, the MPI benchmarks measure two two-sided handshake communications and then divide the round trip time by two. The barrier operation measures the latency for the indicated number of processes to call *barrier*.

For point-to-point calls, the OpenSHMEM *put* and *get* operations show comparable latencies at all sizes, with *get* operations slower at low message sizes and faster at high message sizes. This crossover occurs around a message size of 4KB. The MPI basic communication calls show execution latencies that are similarly comparable to both *put* and *get* communication latencies. At smaller message sizes ($\leq$4KB), *put* latencies are lower by an average of 0.091µs, and *get* latencies are higher by an average of 0.531µs. This latency gap widens at larger message sizes to 3.56µs higher for *put* and 3.75µs lower for *get* per operation, but is still a relatively insignificant difference in comparison to app execution time.

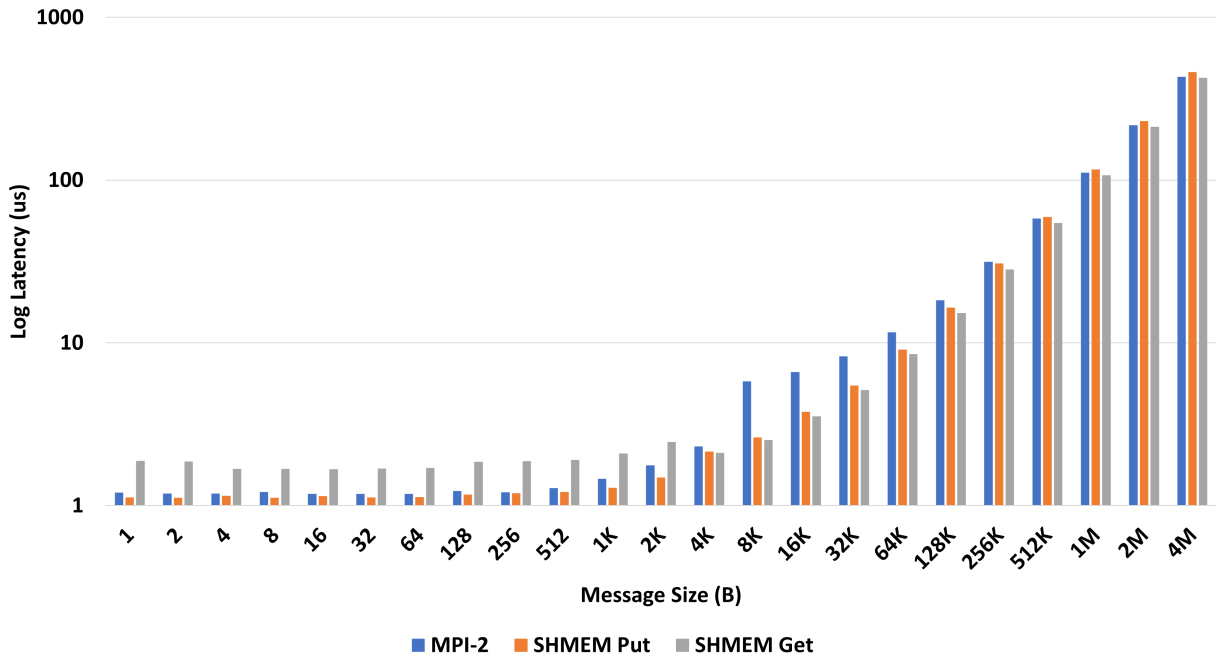

Figure 3: NERSC Point-to-point Log Latencies

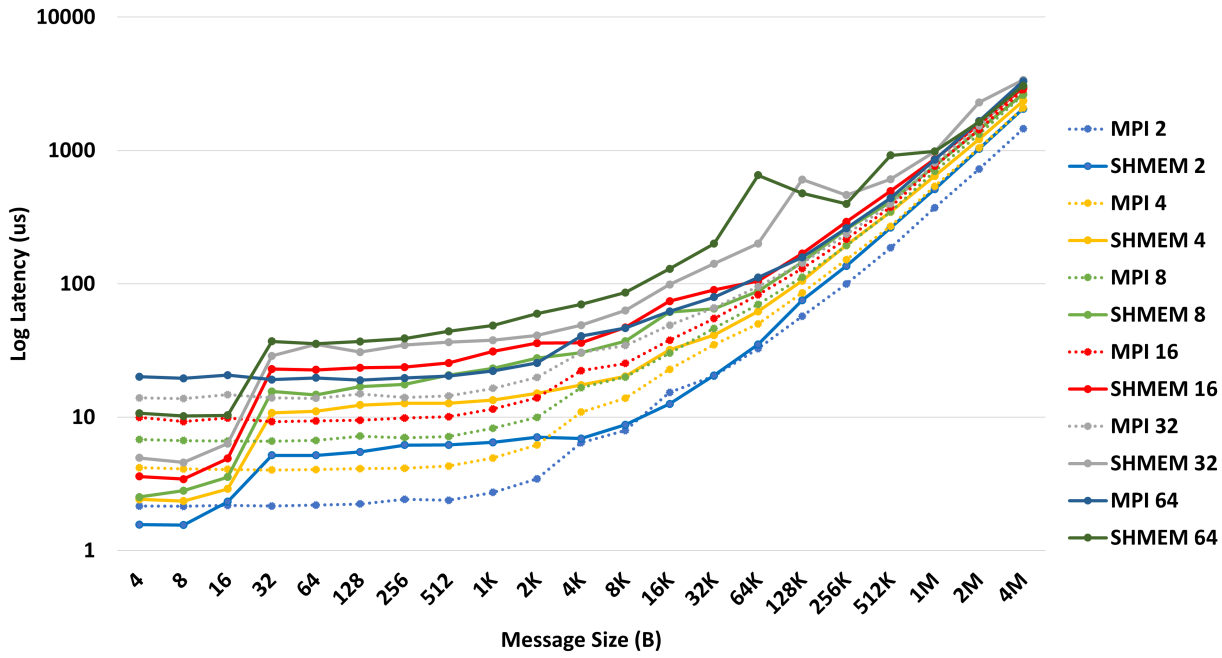

Figure 4: NERSC AllReduce Log Latencies (Legend specifies communication library and node count)

Table 4: Barrier Latencies (μs)

Barrier Latencies (μs)		
N	**MPI 2-sided**	**OpenSHMEM**
2	**1.24**	1.48
4	5.16	**2.15**
8	7.12	**2.62**
16	12.72	**6.41**
32	13.10	**4.62**
64	14.48	**6.64**

Collective operations shown in Figure 4 and Table 4 are scaled in message size and number of processes. The OpenSHMEM *barrier-all* latencies increase at a slower rate than the MPI counterparts, scaling by a factor of 4.47 from 2 to 64 nodes, while MPI scales by a factor of 11.66. The *all-reduce* latencies display more variation. At lower message sizes ($\leq$4KB) the OpenSHMEM latencies are on average 74.18% slower than MPI, but at larger message sizes are 28.7% faster on average than MPI. As the number of processes increases, the difference in latency between the MPI and OpenSHMEM calls shrinks. There is an average of 111.8% absolute difference in latency from MPI to OpenSHMEM for 2 processes, but only 63.9%, 75.8%, and 71.5% average absolute difference for 4, 8, and 16 processes, respectively. In addition, OpenSHMEM latencies are higher than MPI counterparts for large message sizes ($\geq$8KB) with 2 processes, but are on average lower when running with more processes. There is also a range of message sizes (32 bytes to $\sim$2KB) where OpenSHMEM latencies are significantly larger than MPI, with an average percent increase of 118.3%.

5.2 MST Algorithm

Figure 5 displays the differences between the single and leader modes of post-processing measured on 4 nodes, scaled from 4 to 16 PEs on the dataset of six webgraphs for the MPI implementation. The total execution time at 4 processes was similar between the two post-processing methods, with an average of only 1.1% increase in execution time from leader to single modes for MPI. At a higher number of nodes the two versions diverged further, with average percent increase widening to 18.9%, 21.7%, and 22.2% for 8, 12, and 16 PEs, respectively. The peak observed for multiple webgraphs at 12 PEs is a result of under-utilization of resources, since each node gets data from 4 PEs and one node is unused. These differences at a higher number of PEs were in favor of the single post-processing method. As a result, final data was collected using the single post-processing method for both MPI and OpenSHMEM implementations.

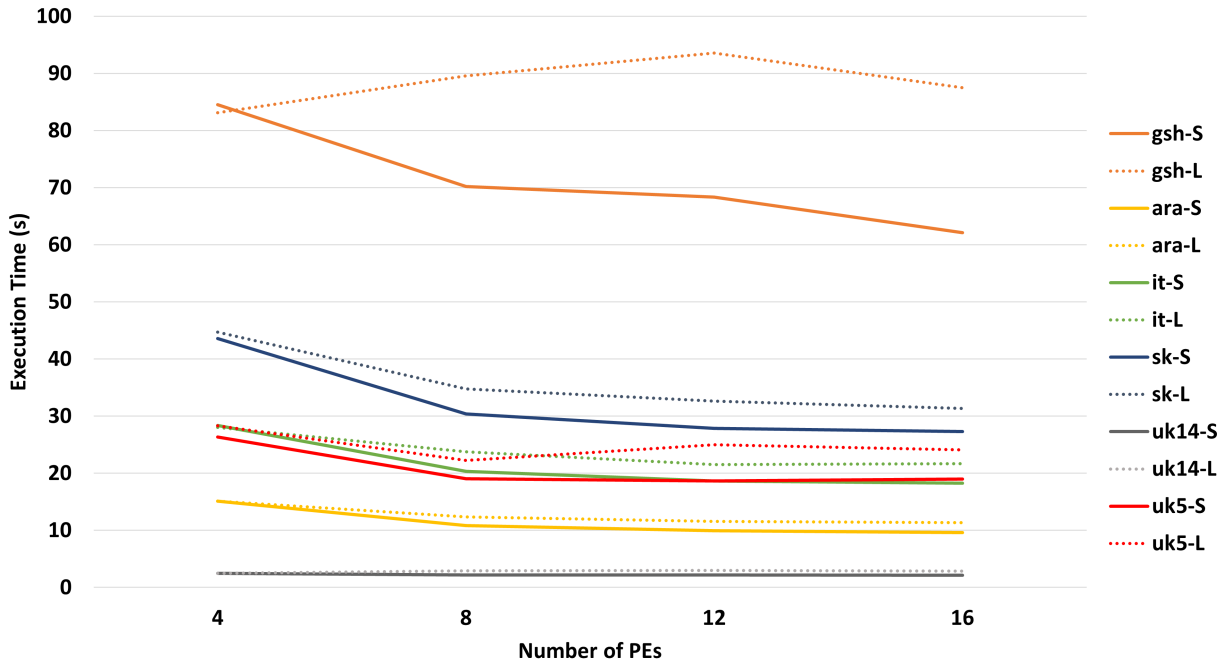

Figure 5: MPI implementation single (-S) vs. leader (-L) post-processing methods.

The scaled execution time data for both implementations of the MND-MST algorithm are presented below with raw execution times in Figs. 6, 7, 8, 9, 10, and 11. Data for these experiments was collected for all 6 webgraphs using NERSC Haswell nodes on the Cori

partition, and was scaled up to 16 nodes and up to 64 PEs. MPI results are denoted by the blue bars, and SHMEM results are denoted by the orange bars. The yellow bar displays the best overall MPI performance, and the green bar displays the best overall SHMEM performance. As mentioned previously, not all node-PE configurations were executable on NERSC due to memory limitations. These are represented by blank bars. Bar labels denote the total number of PEs.

Tables 5 and 6 show averaged performance improvement results for MPI and OpenSHMEM implementations. PE scaling and node scaling are displayed separately. These results are averaged over all 6 webgraphs, and represent performance improvement compared to the 1 node, 4 PE configuration for each implementation. Best percent increase in performance is in bold. Table 7 shows the best configuration by webgraph for each implementation. Configurations are represented as (Nodes, PEs).

Table 5: PE Scaling Performance Improvement

PEs	**MPI**	**SHMEM**
4	2.63%	10.04%
8	25.16%	23.37%
12	28.18%	29.96%
16	**28.46**%	**32.94**%
20	20.94%	17.28%
32	14.93%	19.10%
64	-67.60%	2.43%

Comparative performance data by webgraph is presented in Table 8. This table shows the average percent decrease in total execution time from MPI to OpenSHMEM across all Node-PE configurations. It also shows the correlation coefficients for three metrics (edges, file size, and edge-to-vertex ratio) with respect to average performance improvement.

Table 6: Node Scaling Performance Improvement

Nodes	**MPI**	**SHMEM**
1	19.53%	3.43%
2	14.94%	21.86%
4	**22.16**%	10.49%
8	17.90%	**37.48**%
16	13.40%	32.90%

Table 7: Best Configurations (Nodes, PEs)

Webgraph	**MPI**	**SHMEM**
uk-2014	(4, 16)	(8, 20)
gsh-2015	(4, 20)	(8, 16)
ara-2005	(4, 16)	(8, 16)
uk-2005	(1, 12)	(4, 8)
it-2004	(4, 20)	(8, 16)
sk-2005	(4, 16)	(16, 32)

5.3 Productivity Studies

In addition to demonstrating scaling and performance results for the MPI and OpenSHMEM-based apps, the development productivity of each implementation of the algorithm is measured and compared. When measuring API calls, OpenSHMEM and MPI share a common setup structure each with corresponding *init* and *finalize* calls. For the sake of simplicity, these along with *shmem_malloc* and *shmem_free* calls are ignored in API counts to avoid dilution. The OpenSHMEM-based app shows an increase in LOC by 18.01%, and an increase in API calls by 34.15% as shown in Table 9.

Table 8: Average Performance Improvement

Webgraph	**Performance Improvement**	**Edges**	**Size (GB)**	**E/V**
uk-2014	21.07%	3.65e7	0.15	20.66
gsh-2015	20.71%	1.20e9	4.70	39.09
ara-2005	37.93%	1.28e9	4.90	56.28
uk-2005	26.75%	1.87e9	7.25	47.46
it-2004	39.04%	2.30e9	8.80	55.74
sk-2005	39.63%	3.90e9	15.00	77.00
Avg/Correlation	**30.86%**	**0.71**	**0.71**	**0.86**

Table 9: Implementation Productivity

Function	**API Calls**		**Lines of Code**	
	MPI	*SHMEM*	*MPI*	*SHMEM*
Graph Part	3	6	247	273
Ghost Info	7	12	54	91
Merge	14	25	117	185
Post Proc	24	29	128	160
Total	**82**	**110**	**1188**	**1402**

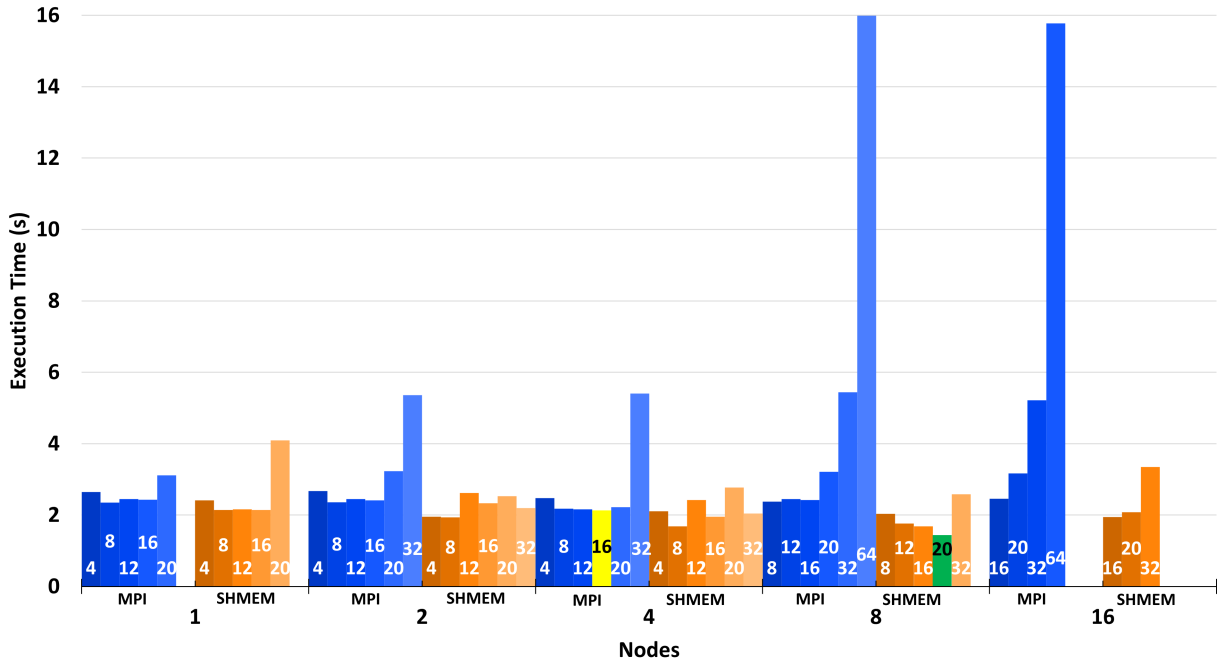

Figure 6: uk-2014 Webgraph Performance Comparison.

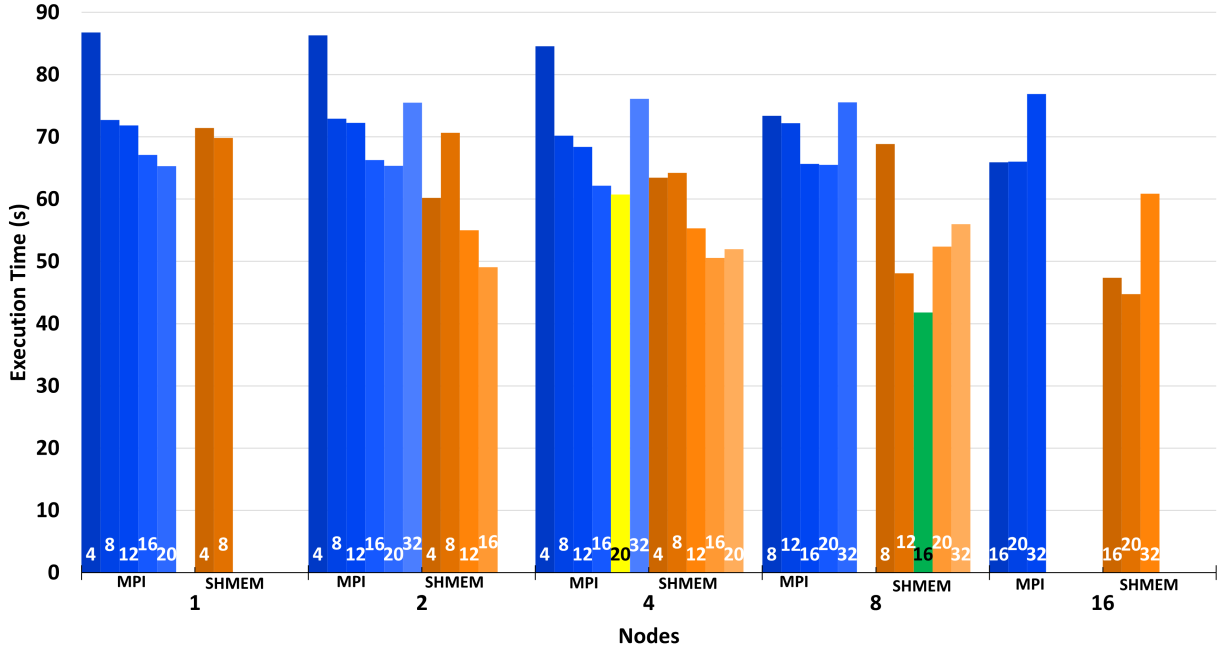

Figure 7: gsh-2015 Webgraph Performance Comparison.

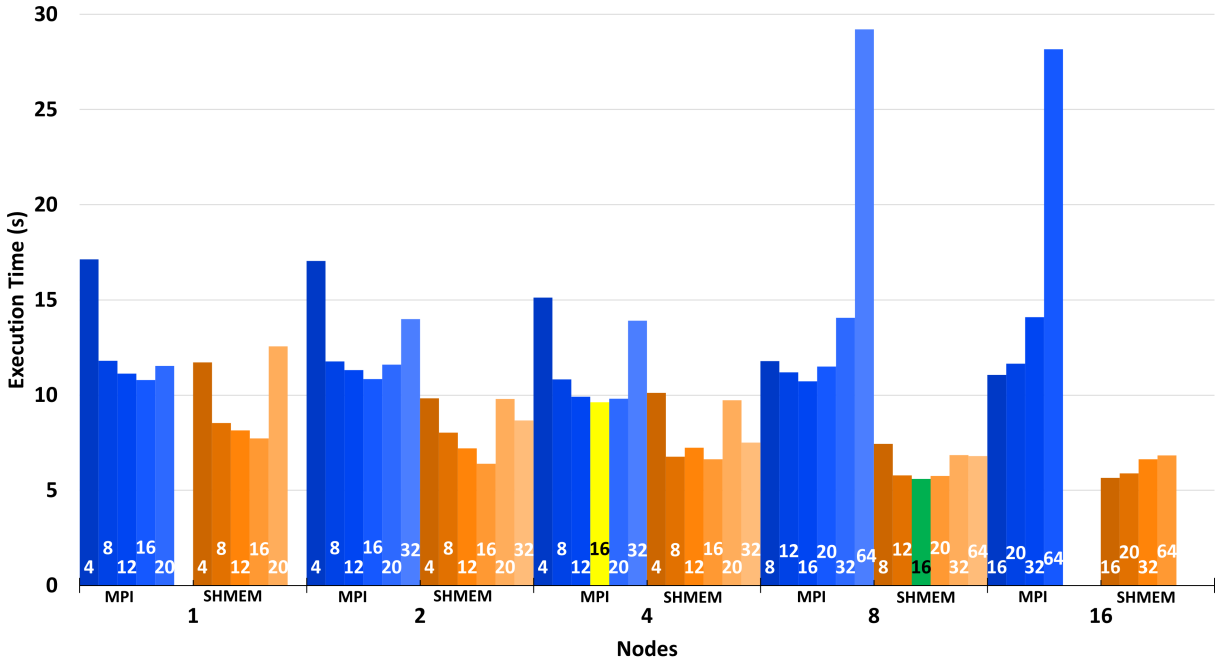

Figure 8: ara-2005 Webgraph Performance Comparison.

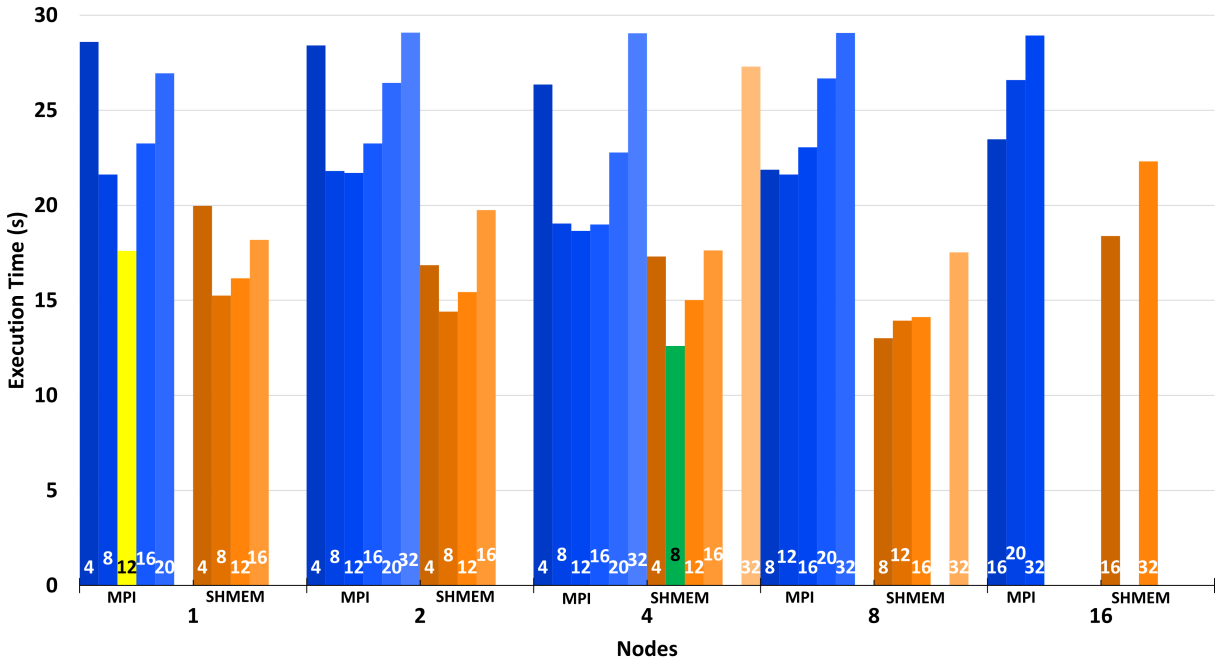

Figure 9: uk-2005 Webgraph Performance Comparison.

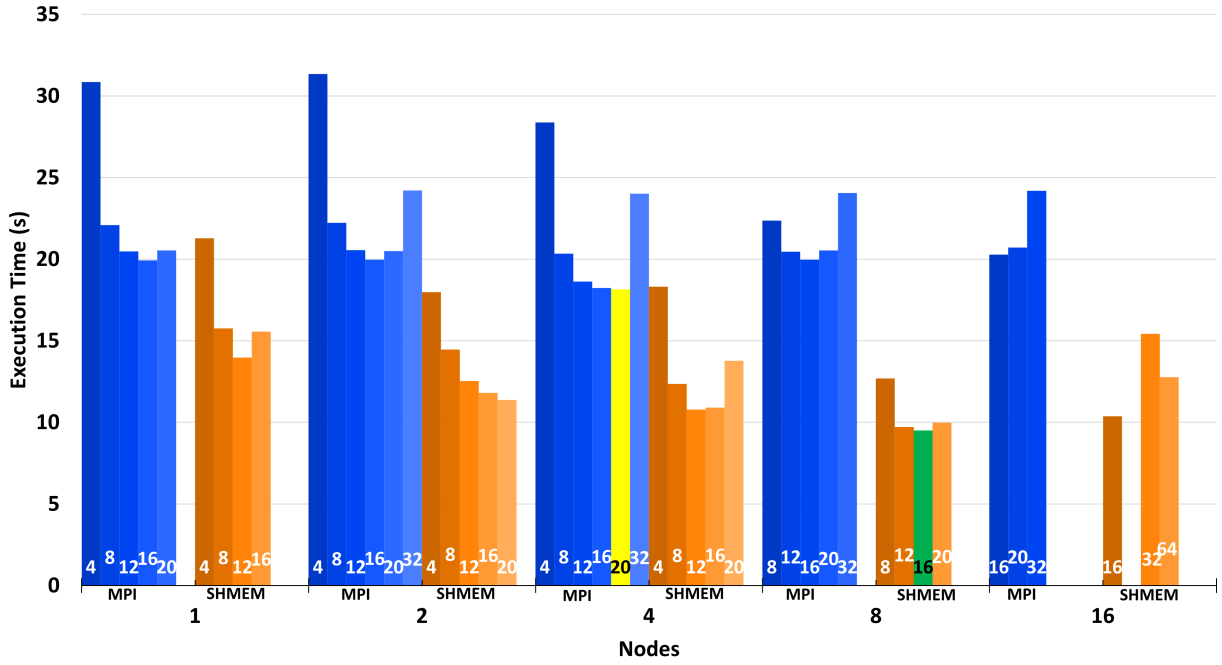

Figure 10: it-2004 Webgraph Performance Comparison.

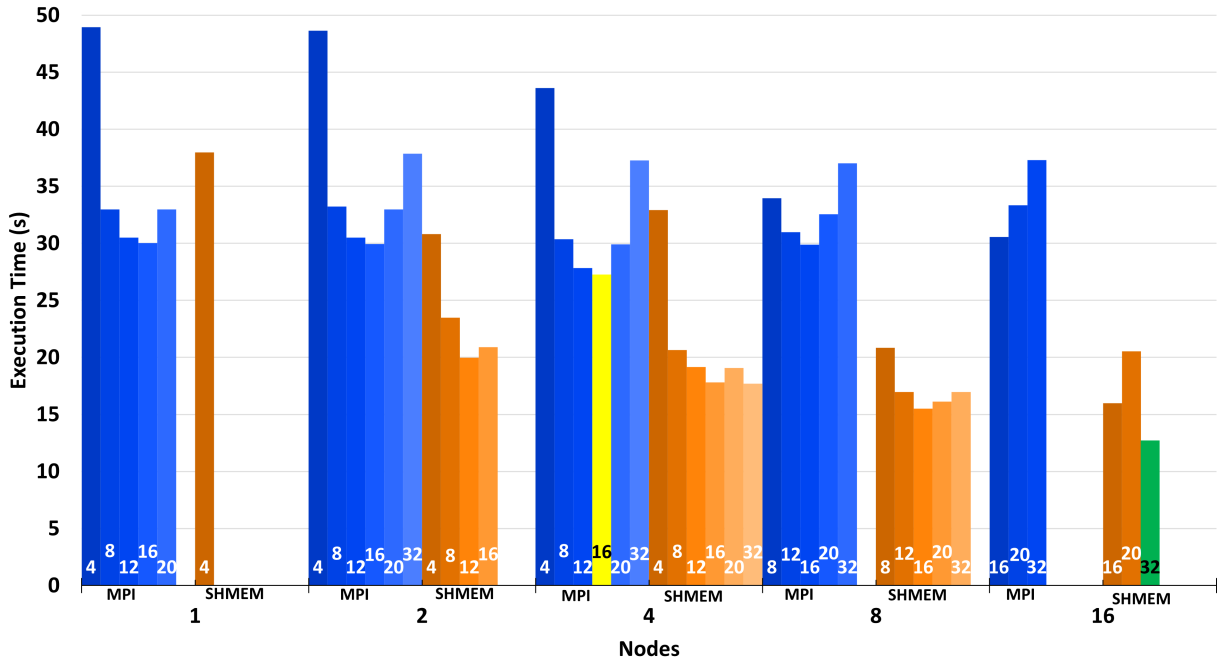

Figure 11: sk-2005 Webgraph Performance Comparison.

6.0 Discussion

This section evaluates differences in performance at the API and app levels, in the context of message size and webgraph composition. It also examines the change in productivity with respect to overall performance.

6.1 API Calls

When compared directly on the API-level, the point-to-point OpenSHMEM operations are on-par with their MPI counterparts, with some variation depending on message size and number of processes. Providing inherent nonblocking behavior at the target PE, the *put* and *get* SHMEM calls have similar latencies to the MPI Send-Recv pair. On the collective side, the OpenSHMEM *barrier-all* operation outperforms that of MPI for all processor counts above two. This is likely due to the lack of a communicator argument in the SHMEM *barrier-all* call that is present in *MPI_Barrier*, which could reduce latency. The *all-reduce* operation is more nuanced. *MPI_allreduce* outperforms the OpenSHMEM implementation for message sizes larger than 16 bytes and processor counts greater than 2. The average percent increase in latency from MPI to OpenSHMEM is 142.09% for message sizes between 32 bytes and 2KB, but this increase falls to an average of only 31.76% for message sizes greater than 2KB. While the discrepancies in latency for collective operations are more significant (45.07% average decrease for *barrier-all* and 57.24% increase for *all-reduce* compared to only $\sim$2.5% combined decrease for *put* and *get*), these differences are still relatively minor in the scope of the entire app runtime. With a difference of at most a few milliseconds per call at the largest message sizes and a few hundred API calls in the entire app at runtime, the performance improvement from SHMEM API calls is on average less than 2% of the total execution time. This result is consistent across all three testbeds (see Appendices A and B), and such a minor improvement alone is not enough to justify the increase in programming complexity that comes with the OpenSHMEM specification. Instead, it is the combination of one-sided

and non-blocking communication patterns with strategic programming techniques explained above that lead to concrete, noticeable speedup over MPI.

6.2 Productivity Studies

The use of communication-computation overlapping techniques and flexible one-sided communication patterns comes with additional program complexity, demonstrated by the ∼34% increase in API-calls and ∼18% increase in LOC for the OpenSHMEM implementation. To combine these metrics into a single result, we averaged both increases to find a combined increased complexity of ∼25%. To produce significant performance improvement and justify this increase in complexity, these programming paradigms must also be thoroughly understood and implemented by the programmer, with the added risk of manual synchronization.

It is important to note that a portion of this increase can be attributed to the use of custom MPI types which are currently not supported by OpenSHMEM. Due to the "shmem_TYPE_OP()" format of SHMEM calls, certain lines were doubled to ensure that the right datatype was being used. Another portion of the increased overhead is caused by the use of "pWrk" and "pSync", two array data structures used to perform certain OpenSHMEM communications including many collective operations [6].

The majority of the differences in productivity can be attributed to the merge and post-processing portions of the algorithm, due to the high number of communication operations present. In addition, the optimized OpenSHMEM-based app uses partitioning and non-blocking communication, which adds additional complexity in the form of synchronization calls (*shmem_barrier* and *shmem_quiet*).

Finally, certain symmetric variables and data structures had to be introduced to keep symmetric memory locations consistent between processors. With MPI, variables of the same name are stored in separate locations across processors and can thus be of different sizes. However, any pointer or variable declared in the symmetric memory must be the same size across every PE to avoid invalid accesses. For this reason, new "maximum value" variables

were introduced to ensure symmetric variables had consistent sizes across PEs, which had to be calculated via collective communication. This introduced more overhead in the form of additional API calls as well as lines of code.

One drawback of using OpenSHMEM is that, as of writing, the OpenSHMEM specification version 1.4 only supports "to-all" communication for many collective API-calls, meaning all processes receive data from each communication [5]. MPI allows for single processes to be the target of a collective communication, as with an *MPI_Reduce* or *MPI_Gather*. This is due to the use of the symmetric heap present across all PEs, and leads to more overhead for corresponding OpenSHMEM calls. In addition, performing any "to-one" collective operation equivalent to an *MPI_Reduce* or *MPI_Gather* must be programmed manually, using a series of sequentially executed point-to-point operations that reduce productivity and increase execution time. An example comparison between an MPI to-one collective communication and an OpenSHMEM to-one communication using seqeuential point-to-point operations can be found in Appendix B. As a result, any such to-one communications in the algorithm were simply replaced with to-all communications, unless noted otherwise.

6.3 MST Algorithm

The changes in API calls alone do not provide a significant amount of performance improvement, and increase the programming complexity of the app. To fully exploit the benefits of the OpenSHMEM specification, the programmer must use strategic programming techniques, non-blocking communication, and RMA to maximize uninterrupted computation and minimize communication time.

The result of the added overhead and nuanced programming strategies is promising, with performance improvements from MPI to OpenSHMEM averaging over 30% for all node-PE configurations. Some graphs seemed to perform better with OpenSHMEM; the it-2004 and sk-2005 webgraphs averaged nearly 40% improvement in execution while gsh-2015 and uk-2014 showed an average improvement of 20%. This variation in performance correlates roughly with file size and number of edges, with the largest two webgraphs (sk-2005 and it-

2004) showing the best improvement and the smallest two webgraphs (gsh-2015 and uk-2014) showing the least improvement. The correlation coefficient between average percent decrease in execution time and both file size and number of edges is 0.71. Performance improvement is even better correlated with edge-to-vertex ratio, with a correlation coefficient of 0.86. This improvement is likely due to the larger number of edges per vertex to analyze, which results in a larger volume of communication and more potential for performance gain from SHMEM optimizations. These strong correlations suggest promising scaling results for OpenSHMEM with larger graphs and other large-scale HPC apps.

In terms of PE scaling, this work demonstrates results that are consistent with [2], insofar as the app scales up to and reaches a minimum around 16 PEs before plateauing or losing performance. At all node counts, both MPI and OpenSHMEM implementations display the best performance improvement between 12 and 20 PEs with the exception of the sk-2005 webgraph. When measuring percent decrease in execution time compared to the 1 node, 4 PE configuration, both implementations show optimum performance with a PE count of 16, with an average percent improvement of 28.46% for MPI and 32.94% for OpenSHMEM. The worst performance for both implementations is at 64 PEs, followed closely by 4 PEs. PE counts of 8 to 20 see more consistent performance improvement.

For node scaling, MPI shows optimum performance with 4 nodes at an average of 22.16% improvement, while OpenSHMEM peaks at 8 nodes, with an average of 37.48% decrease compared to 1 node and 4 PEs. MPI displays worst performance with 16 nodes, while OpenSHMEM displays worst performance when using 1 node. With too few or too many nodes, graph data can either be too distributed or not distributed enough, resulting in extra communication overhead or inadequate parallelization. The variability of scaling results is due to the partitioning of the graphs by the processes, and is highly dependent on the format of the graph itself. While some graphs are amenable to more PEs and increased vertex subdivision, other graphs might not be able to mask the increased communication overhead with independent computation or data partitioning.

7.0 Conclusions

At the app level, PGAS communication models such as OpenSHMEM show promising results in terms of consistent scaled performance improvement, in spite of limited latency difference between API calls. Through the utilization of strategic programming techniques and flexible RMA communications, the OpenSHMEM specification demonstrated significant improvement over MPI on a parallel graph app, with an equal or lower percent increase in programming complexity based on lines of code and number of API calls. The performance improvement from MPI to OpenSHMEM also demonstrates positive correlation with increasing webgraph size and edge-to-vertex ratio, indicating that OpenSHMEM has promising scaling potential on HPC apps. As the specification continues to be developed, more complex communication schemes will be supported, increasing the range of apps and problems that can adopt this growing model.

This research provides a foundation for studying the OpenSHMEM specification at a higher level. The baseline API-call comparison provides context for evaluating the presented RMA programming optimizations, and the examination of productivity quantifies the increased workload for the app developer. As apps and databases increase in scale, distributed-computing systems will become even more prominent, and large-scale distributed app developers may turn to other methods to improve performance. With high scalability and a low barrier of entry, the OpenSHMEM specification should be heavily considered for complex apps in the ever-adapting field of HPC.

8.0 Future Work

The speedup displayed from using OpenSHMEM optimizations is promising, and scales well. It has presently only been applied to the baseline version of the algorithm which focuses on CPUs. Panja and Vadhiyar also describe a hybrid version of the algorithm, leveraging GPUs to achieve higher levels of acceleration, with the added cost of host-device communication overhead and complexity. There is significant potential for further development on this implementation. NVIDIA has recently released its own version of the OpenSHMEM library for GPUs, called NVSHMEM, which uses GPUDirect RDMA (GDR). This allows GPUs to directly communicate with one another, avoiding the CPU communication bottleneck [27]. In addition to the acceleration displayed in this work with one-sided and non-blocking communication, the application of the NVSHMEM library to the MST algorithm could lead to further latency reduction.

While NVSHMEM has not yet been applied to larger apps, it is our hope to continue to expand this work to the hybrid GPU algorithm, potentially combining OpenSHMEM and NVSHMEM libraries. This would more robustly explore the performance improvement potential of the MND-MST algorithm, and would combine two SHMEM libraries at a larger scale.

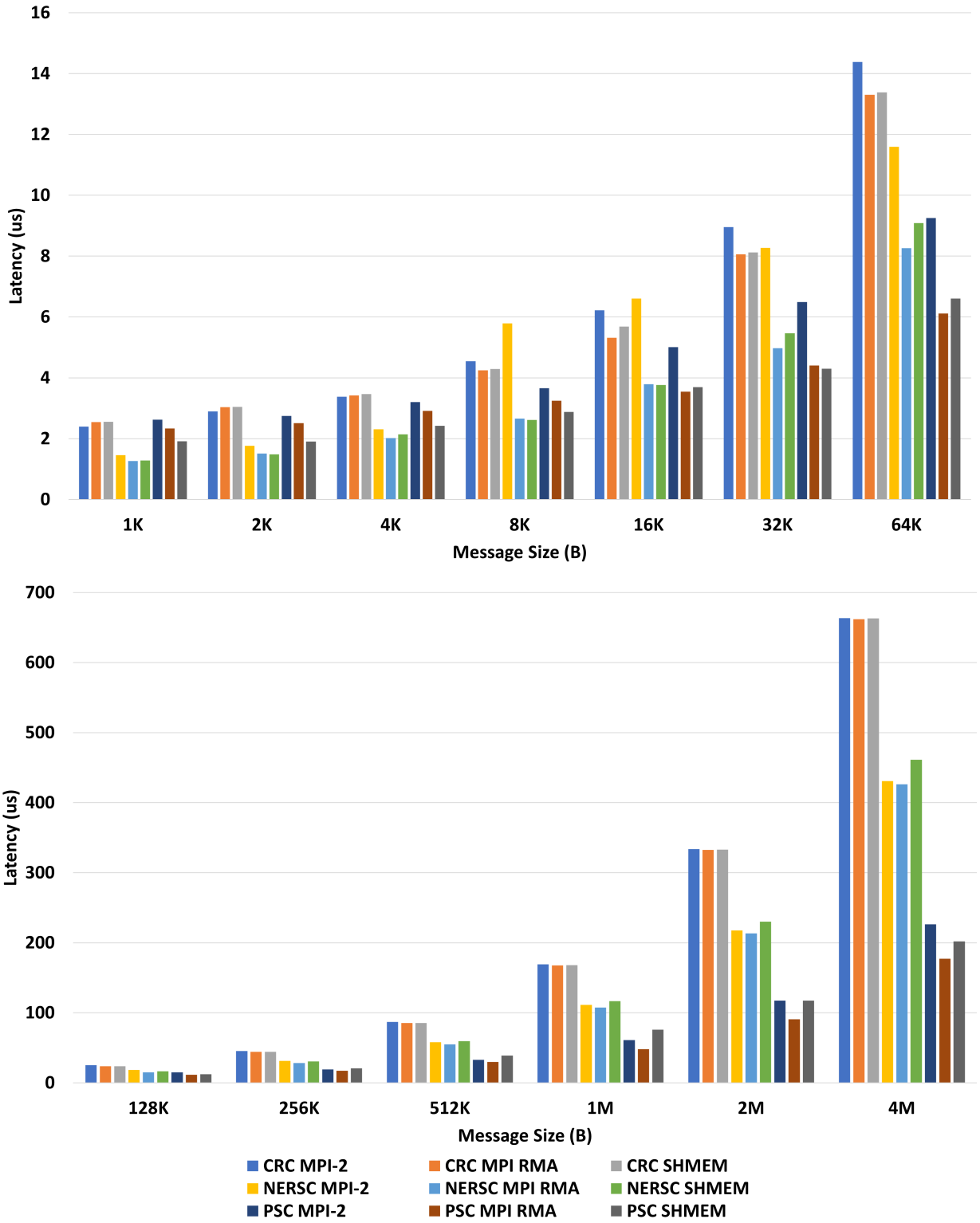

Figure 12: Scaling Put Latencies

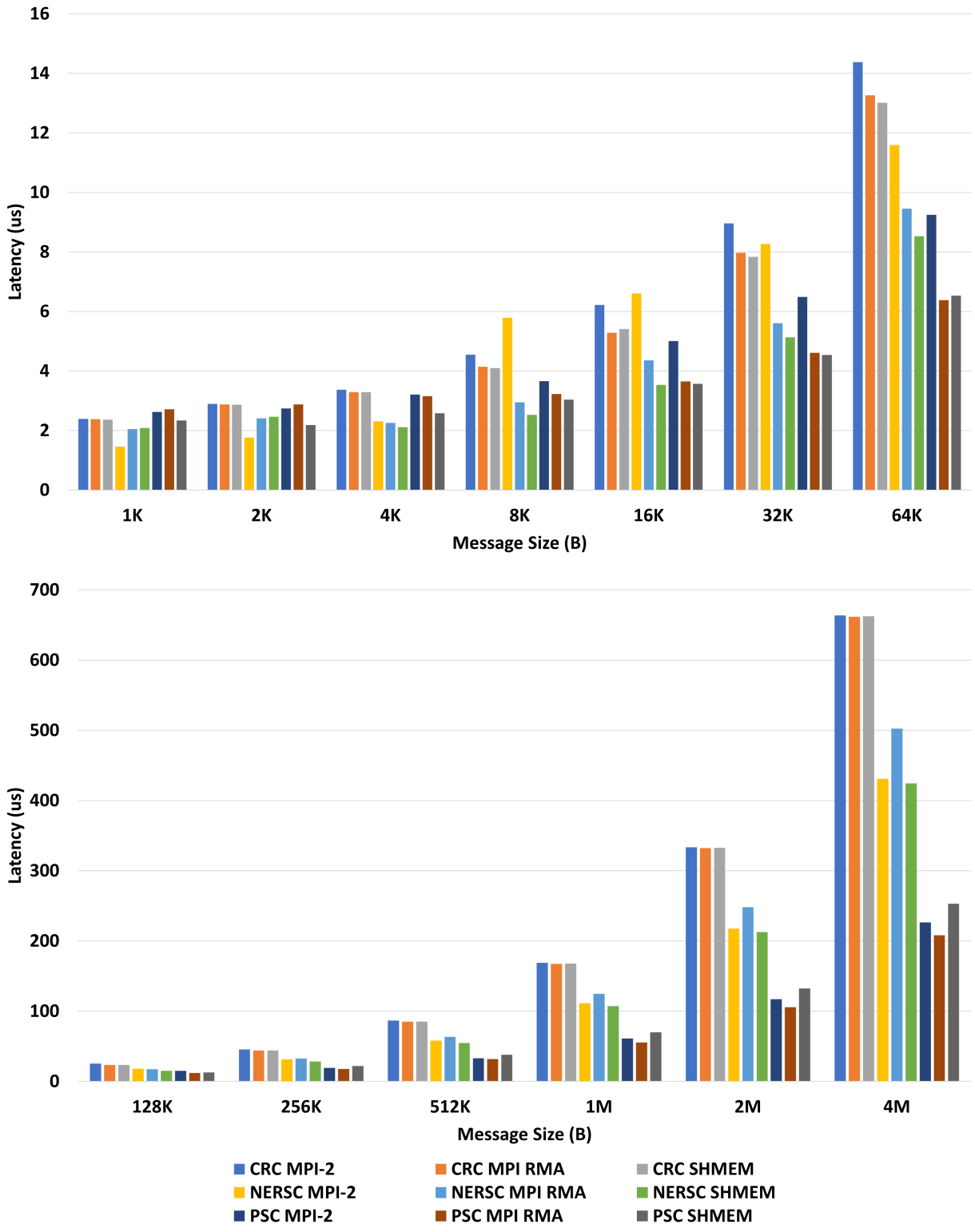

Figure 13: Scaling Get Latencies

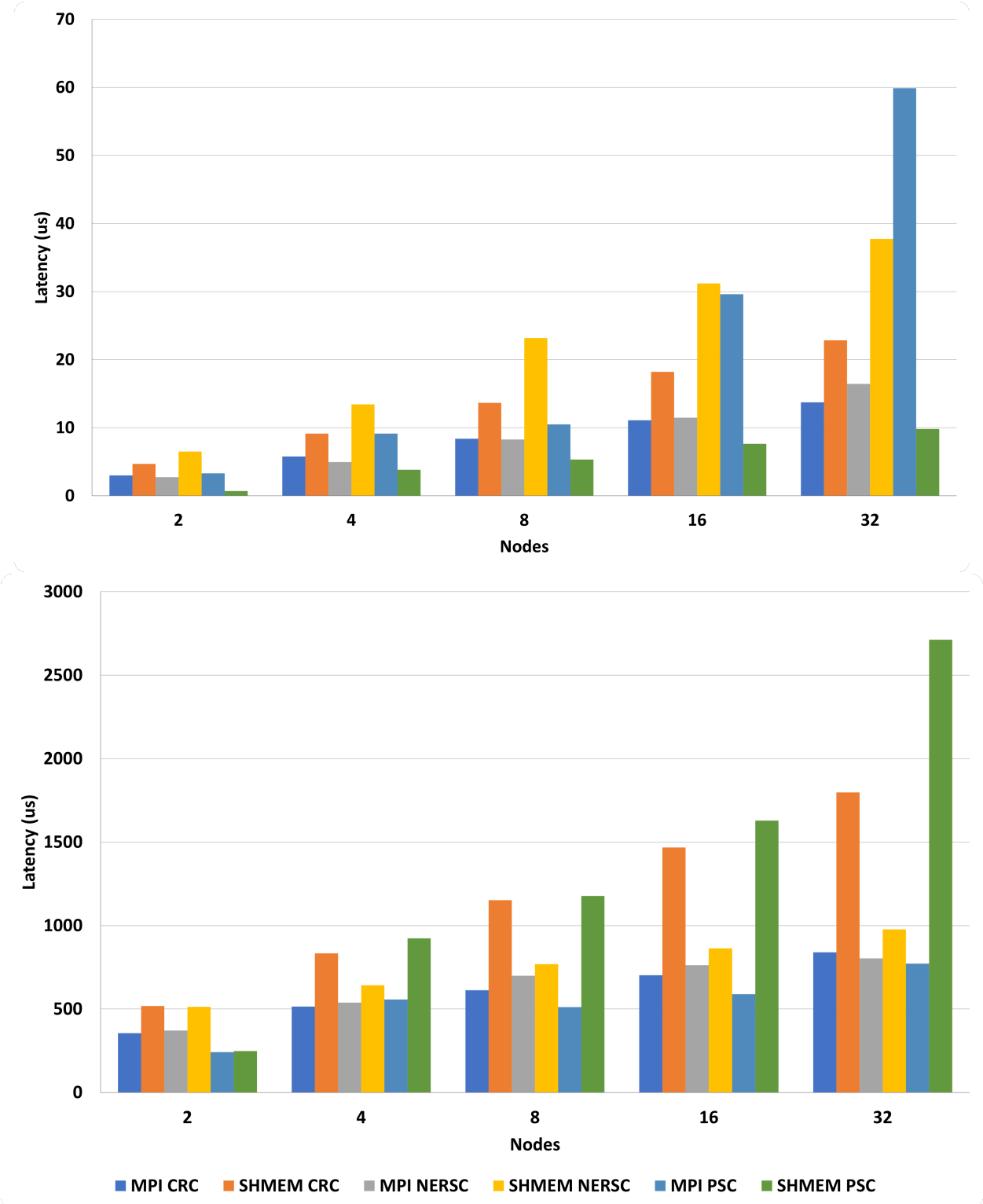

Figure 14: Scaling AllReduce Latencies (Top: 1KB, Bottom: 1MB)

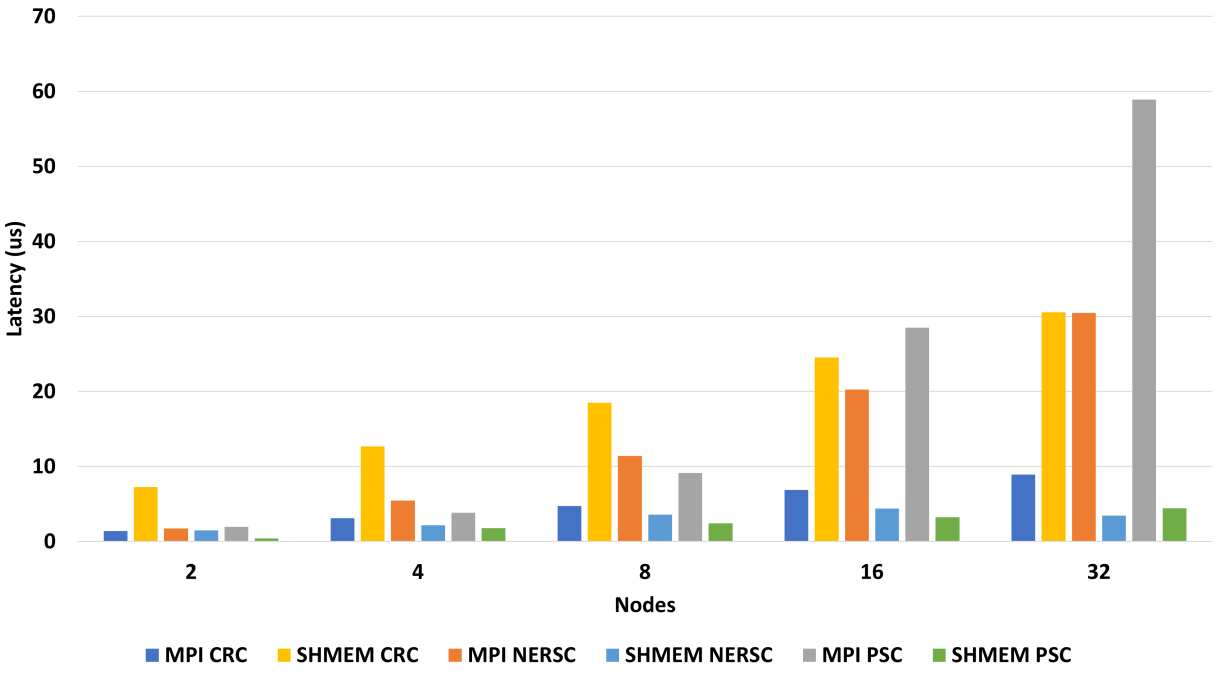

Figure 15: Scaling Barrier Latencies

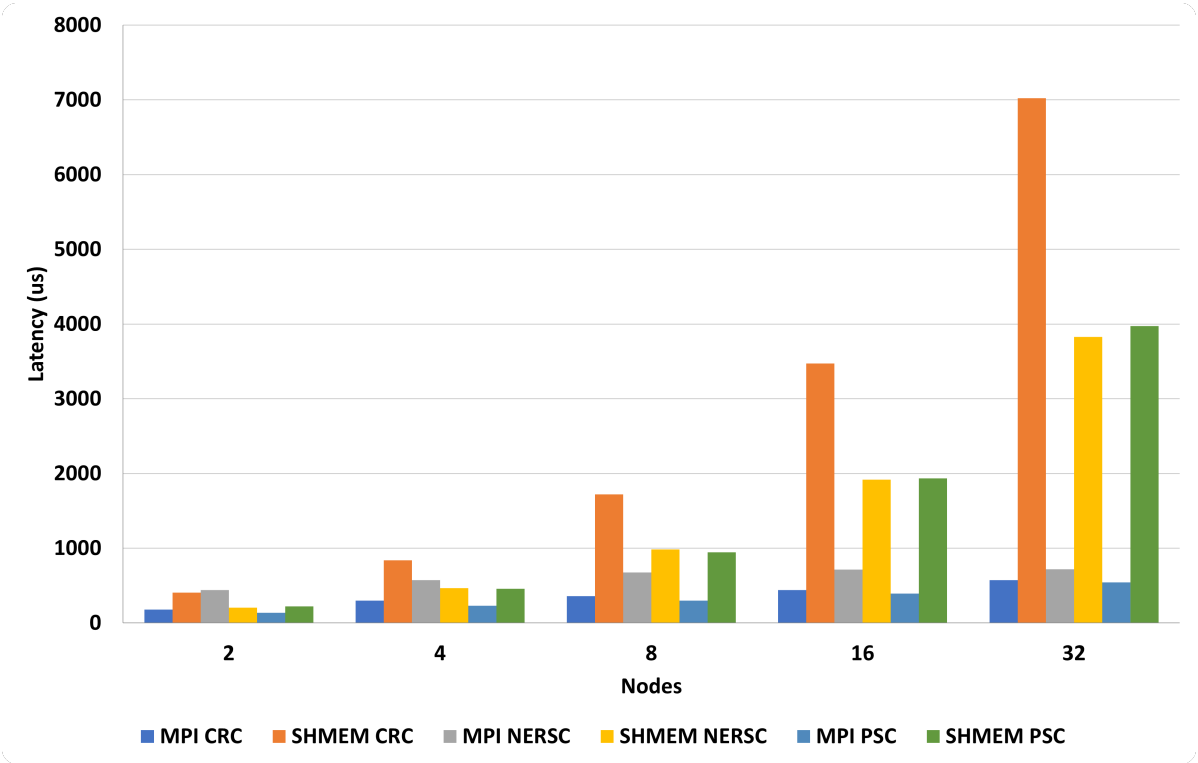

Figure 16: Scaling Reduce Latencies (1MB)

Bibliography

[1] T. Stitt, *An Introduction to the Partitioned Global Address Space (PGAS) Programming Model*. OpenStax CNX, March 2020.

[2] R. Panja and S. Vadhiyar, "Mnd-mst: A multi-node multi-device parallel boruvka's mst algorithm," in *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, (New York, NY, USA), Association for Computing Machinery, 2018.

[3] M. Schulz *et al.*, "Mpi: A message passing interface standard 2019 draft specification," Nov. 2019.

[4] W. D. Gropp and R. Thakur, "Revealing the performance of mpi rma implementations," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (F. Cappello, T. Herault, and J. Dongarra, eds.), (Berlin, Heidelberg), pp. 272–280, Springer Berlin Heidelberg, 2007.

[5] U. D. of Defense, O. R. N. Laboratory, and L. A. N. Laboratory, "Openshmem application programming interface version 1.4," Dec. 2017.

[6] B. Chapman *et al.*, "Introducing openshmem: Shmem for the pgas community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, (New York, NY, USA), Association for Computing Machinery, 2010.

[7] K. Feind, "Shared memory access (shmem) routines," in *Mile High Performance (CUG 1995)*, Cray Research, Inc., 1995.

[8] "Openshmem application programming interface version 1.5," Jun. 2020.

[9] J. Nesetril, "A few remarks on the histroy of mst-problem," *Archivum Mathematicum*, vol. 33, no. 1, pp. 15–22, 1997.

[10] V. Jarník, "O jistém problému minimálním [about a certain minimal problem]," vol. 6, no. 4, pp. 57–63, 1930.

[11] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," vol. 7, pp. 48–50, American Mathematical Society with MSC, 1956.

[12] O. Borúvka, "O jistém problému minimálním [about a certain minimal problem]," vol. 5, no. 3, pp. 37–58, 1926.

[13] J. Jose, J. Zhang, A. Venkatesh, S. Potluri, and D. Panda, "A comprehensive performance evaluation of openshmem libraries on infiniband clusters," in *OpenSHMEM Workshop*, March 2014.

[14] H. Fu, M. Gorentla Venkata, S. Salman, N. Imam, and W. Yu, "Shmemgraph: Efficient and balanced graph processing using one-sided communication," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 513–522, 2018.

[15] M. Grossman *et al.*, "Hoover: Distributed, flexible, and scalable streaming graph processing on openshmem," in *OpenSHMEM and Related Technologies. OpenSHMEM in the Era of Extreme Heterogeneity*, (Cham), pp. 109–124, Springer International Publishing, 2019.

[16] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[17] G. Wang, H. Lam, A. George, and G. Edwards, "Performance and productivity evaluation of hybrid-threading hls versus hdls," in *2015 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, 2015.

[18] V. Olman, F. Mao, H. Wu, and Y. Xu, "Parallel clustering algorithm for large data sets with applications in bioinformatics," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 6, no. 2, pp. 344–352, 2009.

[19] J. L. Bentley, "A parallel algorithm for constructing minimum spanning trees," *Journal of Algorithms*, vol. 1, no. 1, pp. 51–59, 1980.

[20] G. Malewicz *et al.*, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, (New York, NY, USA), p. 135–146, Association for Computing Machinery, 2010.

[21] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Effective techniques for message reduction and load balancing in distributed graph computation," in *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, (Republic and Canton of Geneva, CHE), p. 1307–1317, International World Wide Web Conferences Steering Committee, 2015.

[22] "Pitt research center for research computing," 2018.

[23] J. Towns *et al.*, "Xsede: Accelerating scientific discovery," *Computing in Science Engineering*, vol. 16, pp. 62–74, sep 2014.

[24] B. Friesen, "Cori system - nersc documentation," 2020.

[25] D. K. Panda, H. Subramoni, C.-H. Chu, and M. Bayatpour, "The mvapich project: Transforming research into high-performance mpi library for hpc community," *Journal of Computational Science*, p. 101208, 2020.

[26] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, (Manhattan, USA), pp. 595–601, ACM Press, 2004.

[27] C.-H. Hsu and N. Imam, "Assessment of nvshmem for high performance computing," *International Journal of Networking and Computing*, vol. 11, no. 1, pp. 78–101, 2021.