A Study in SHMEM: Parallel Graph Algorithm Acceleration with Distributed Symmetric Memory[⋆]

Michael Ing and Alan D. George

Department of Electrical and Computer Engineering, University of Pittsburgh
NSF Center for Space, High-Performance, and Resilient Computing (SHREC)
{mci10,alan.george}@pitt.edu

Abstract. Over the last few decades, the Message Passing Interface (MPI) has become the parallel-communication standard for distributed algorithms on high-performance CPUs. 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 by reducing synchronization calls between processors and overlapping 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 apps. To provide further context for the comparison of MPI to the OpenSHMEM specification, 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 ∼25%. These findings encourage further study into the use of distributed symmetric-memory architectures and RMA-communication models applied to both additional hardware systems and scalable HPC apps.

Keywords: MPI · RMA · OpenSHMEM · PGAS · HPC · MST.

[⋆] 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

1 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 ubiquity 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 [14].

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 [6].

In the last few decades, an older 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 [9]. 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 [13]. The OpenSHMEM specification is applied to an existing MPI implementation of the algorithm and directly compared. 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 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

To take advantage of the benefits of both shared-memory and distributed-memory architectures, the PGAS model implements a global address space, local and remote data storage, one-sided communication, and distributed data structures [15]. Global addressing allows individual processors to simultaneously access the same spot in symmetric memory. This one-sided communication leads to increased programming flexibility and communication-computation overlap. But not everything can be stored in symmetric memory. Data stored locally (in "private" memory) can be more rapidly accessed, forcing programmers to decide what data needs to be remotely accessible and what can be kept local. This decision point creates an efficient compromise between performance and ease of access at the expense of more vigilant design [15]. Support for distributed data structures allows more data to be stored, opening the door for complex program compatibility.

2.2 SHMEM

In 2010, SHMEM was standardized into the OpenSHMEM specification by the PGAS community, unifying development efforts and expanding its viability for widespread use [3]. 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 SHMEM, OSHMEM, and SHMEM-UCX.

2.3 Minimum Spanning Tree

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 [2]. 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 [11].

3 Related Research

The OpenSHMEM specification has been explored on the API and app levels, including graph processing. This research extends this investigation by analyzing the specification on both levels for an MST graph-processing app, and evaluating the 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), OMPI-SHMEM, and Scalable-SHMEM (Mellanox Scalable) [8]. 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 demonstrates consistently lower latencies compared to other OpenSHMEM libraries, as well as a smaller memory footprint per process. Jose and Zhang also compare the performance of two kernels, Heat Image and DAXBY. They find that MV2X-SHMEM again outperforms other libraries, demonstrating consistent execution time improvement that scales 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 [5] on "SHMEMGraph", a graph processing framework that focuses on the efficiency of one-sided communication and a global memory space. In order to address communication imbalance, computation imbalance, and inefficiency, the SHMEMGraph framework introduces a one-sided communication channel to support more flexible *put* and *get* operations 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 [5].

3.3 Productivity Studies

To evaluate and compare the productivity of the algorithm using different communication paradigms, multiple metrics are needed. Measuring both overall lines of code (LOC) and number of communication-specific API calls strikes a balance between increased complexity and overall workload. Development time has also been used to measure productivity with HPC toolsets as seen in [16], 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

Work done by Yan and Cheng have developed a system to find minimum spanning tree data structures on distributed processors called Pregel [10]. This system is "vertex-centric", focusing on messages sent between vertices to keep communication simple and efficient [17]. 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.

This approach has inconsistency issues due to varying vertex degree in large-scale graphs, leading to unequal communication backlog and bottlenecks. Two improvements were made in the form of vertex mirroring for message combining as well as the introduction of a request-response API, resulting in the aptly named Pregel+ [17]. 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 [17].

The algorithm used in this research is based on and uses source code from Panja and Vadhiyar [13], who describe the operation of the parallelized, distributed minimum spanning tree graph algorithm. The algorithm is explained in detail in Section 4.4. This research validates the algorithm's performance compared to Pregel+, and shows positive performance improvements for overall execution time on a scaling number of parallel processes from 4 to 16. This work was thus deemed acceptable for use as a state-of-the-art scalable distributed parallel algorithm.

4 Experiments

This section details the nature of experiments performed, data collected, and optimizations implemented. API-level experiments, app datasets, supercomputing testbeds, and MND-MST algorithm optimizations are examined in detail.

4.1 API Level

To frame and analyze results for a larger app, it is important to analyze differences of the baseline, API-level performance. This evaluation is done by directly comparing relevant API calls between MPI and OpenSHMEM. Point-to-point and collective communications are averaged over 500 iterations and these tests are scaled up in message size, with some of the collective operations scaling up in number of parallel processes. Microbenchmark tests for both MPI and OpenSHMEM are created by the MVAPICH project from Ohio State University, with some improvements made to scale to appropriate sizes [12]. Point-to-point benchmarks were executed using two processors 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 64. All API-level benchmarks used one PE per node.

4.2 Datasets

The datasets used for the app consist of large web-based graphs formed by web-crawling [1]. Created by the Laboratory for Web Algorithmics, these graphs are undirected, weighted and have significantly more edges than vertices, making 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 better detection of difference in execution time at scale. Execution times for MPI and SHMEM implementations can be directly compared because the use of different communication libraries have no effect on the way the algorithm is executed. 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.

Table 1. Graph 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

4.3 Testbed

All data was produced by utilizing 2.3GHz Haswell nodes on the Cori partition of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility at Lawrence Berkeley National Laboratory. This supercomputer has over 2,300 nodes each with 128GB of DDR4 memory [4]. Each configuration of runtime parameters averaged execution times over 15 runs. OpenMP sections allocated 4 threads per node.

4.4 Algorithm

This research's algorithm is a parallelized version of the classic Boruvka's algorithm for finding minimum spanning trees, based on [13].

The parallelized version of the algorithm is split into four major parts: graph partitioning, independent computation, merging, and post-processing. During graph partitioning the input graph is read in parallel by each PE and divided into equal parts. The independent computation step allows each PE to run Boruvka's algorithm locally, while the merging step is used for clean-up of individual components and internal edges. 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. Please see [13] for a more detailed description on the algorithm steps and basic functionality.

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. Post-processing occurs after computation and merging, and was set to either “single” or “leader” mode. The “single” mode consists of having each node send all leftover components to PE 0 before final computation, while the “leader” mode splits PEs into groups of 4 for more parallel computation. It was found that the ”single” mode led to better execution due to lower overhead, so all final data was collected using the ”single” post-processing method. 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 24% of the total number of MST edges.

Strong scaling was performed by altering the number of nodes and processing elements per job, scaling nodes from 1 to 16 and PEs from 4 to 64. NERSC nodes were limited to 118GB per node, and 64 PEs per node [4]. Data for multiple 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 for optimal execution. Some failures resulted from symmetric memory (heap and the collective buffer) that was too small to handle communication volume, while others were caused by over-allocation that infringed on private memory. Some node-PE configurations were even rendered impossible, as there wasn't enough memory available to support both symmetric memory for communication and private memory for graph data storage. Webgraphs that were larger in size like the uk-2005 graph tended to require larger symmetric heap sizes to execute properly.

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. The first major OpenSHMEM optimization occurs during the exchanging of ghost information after independent computation, including external vertices and their corresponding edges. In the baseline MPI approach, this step consists of a series of handshake *MPI_send* and *MPI_recv* calls, first exchanging the message size before sending the full data structure of vertices and ghost edges to be updated. Each PE then locally updates the corresponding data structure to reflect changes in component sizes.

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. These communications allow each PE to operate independently while sending the message size, which leads to more efficient execution. Second, the ghost information can be communicated via RMA without the need for any synchronization which eliminates handshaking overhead and slowdown from synchronization.

Finally, 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 by using a *shmem_quiet* for synchronization.

The other prime target for OpenSHMEM optimization is the exchanging of component data during the merging step. In the 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. The OpenSHMEM implementation avoids the handshake overhead entirely by using non-blocking communication calls as well as RMA, which allows each PE to operate independently and retrieve the required information simultaneously. 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 large amount of data that must be communicated between PEs and overlaps it with data structure update overhead to maximize uninterrupted computation. 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 to be communicated is divided into equal chunks and communicated chunk-by-chunk asynchronously, with each communication 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 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 final computation. 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 Results

All data collected are presented in this section, including microbenchmark performance and an app-level comparison of OpenSHMEM and MPI. Additional algorithm tuning data and productivity comparisons are also examined.

5.1 API Level

The results of the API-level OSU microbenchmarks executed on NERSC are shown in Tables 2 and 3. 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 run 2 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*.

Table 2. MPI and OpenSHMEM microbenchmark data. Latencies in µs.

Point-to-point Microbenchmarks			
Size	**MPI 2-sided**	**SHMEM put**	**SHMEM get**
64 bytes	1.18	**1.13**	1.71
1 KB	1.46	**1.28**	2.09
32 KB	8.27	5.46	**5.13**
256 KB	31.47	30.73	**28.24**
2 MB	217.49	230.23	**212.70**
4 MB	430.84	461.23	**424.56**

Barrier Microbenchmark		
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**

Table 3. MPI and OpenSHMEM all-reduce. Latencies in µs

OpenSHMEM							MPI						
Size	**N=2**	**N=4**	**N=8**	**N=16**	**N=32**	**N=64**	**Size**	**N=2**	**N=4**	**N=8**	**N=16**	**N=32**	**N=64**
64 bytes	5.20	10.70	13.99	29.41	26.27	29.30	64 bytes	1.36	5.52	5.60	9.06	13.59	24.06
1 KB	6.11	13.62	21.67	28.92	32.22	37.79	1 KB	1.98	13.02	10.16	18.11	23.27	19.12
32 KB	18.59	47.79	72.73	95.59	93.05	101.88	32 KB	20.69	171.23	82.76	185.84	280.24	432.05
256 KB	127.90	214.39	270.90	358.52	274.18	288.69	256 KB	89.35	410.93	403.74	888.88	1113.13	745.60
2 MB	974.60	1167.45	1456.71	1547.68	1454.02	1488.44	2 MB	618.35	2994.57	3344.70	3774.10	2609.00	3459.31
4 MB	1966.46	2342.73	2712.33	2876.35	3127.96	3070.97	4 MB	1217.65	5026.94	4760.20	4891.37	5215.68	4848.71

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 for app execution time.

Collective operations shown in Tables 2 and 3 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$4KB) the OpenSHMEM latencies are on average 74.18% slower than MPI, but at larger message sizes are 28.7% faster on average than MPI. As the number of processes increases, the difference in latency between the MPI and OpenSHMEM calls decreases. There is an average of 111.8% absolute difference in latency from MPI to OpenSHMEM for 2 processes, but only 63.9%, 75.8%, and 71.5% average absolute difference for 4, 8, and 16 processes, respectively. In addition, OpenSHMEM latencies are higher than MPI counterparts for large message sizes ($\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

The scaled execution time data for both implementations of the MND-MST algorithm are presented with raw execution times in Figs. 1, 2, 3, 4, 5, and 6. 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.

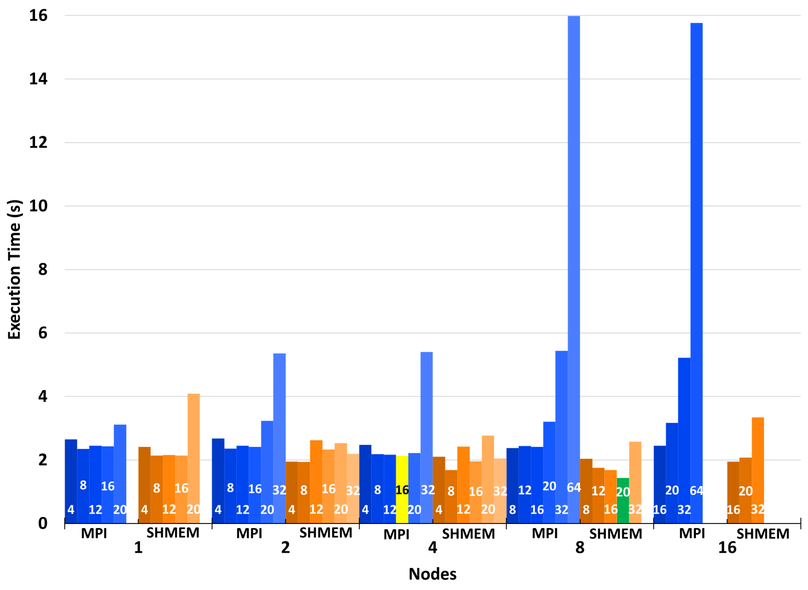

Fig. 1. MPI vs. OpenSHMEM performance for the uk-2014 webgraph. Bar labels denote PEs. Blue=MPI, Yellow=Best MPI, Orange=SHMEM, Green=Best SHMEM.

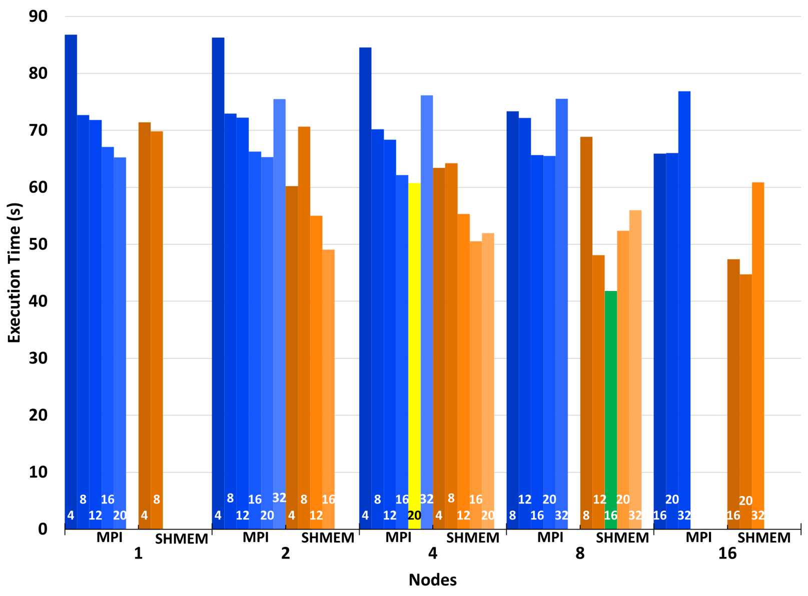

Fig. 2. MPI vs. OpenSHMEM performance for the gsh-2015 webgraph. Bar labels denote PEs. Blue=MPI, Yellow=Best MPI, Orange=SHMEM, Green=Best SHMEM.

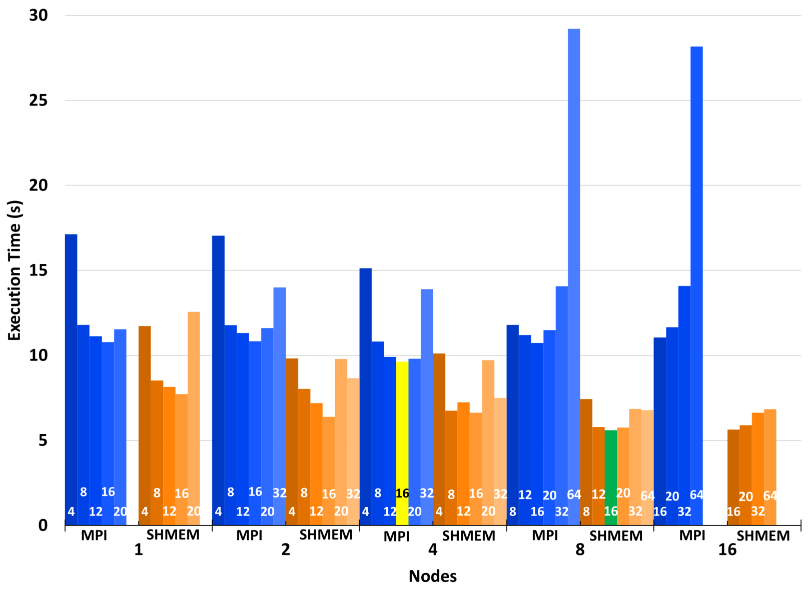

Fig. 3. MPI vs. OpenSHMEM performance for the ara-2005 webgraph. Bar labels denote PEs. Blue=MPI, Yellow=Best MPI, Orange=SHMEM, Green=Best SHMEM.

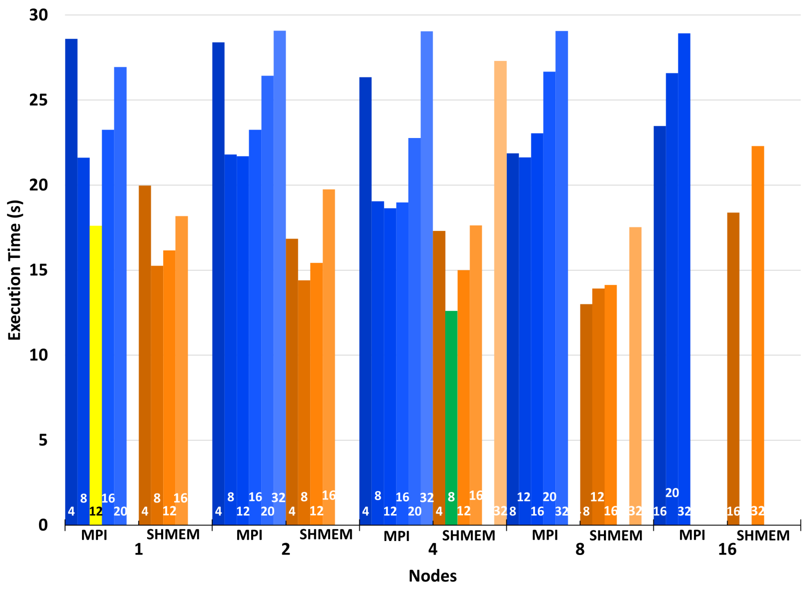

Fig. 4. MPI vs. OpenSHMEM performance for the uk-2005 webgraph. Bar labels denote PEs. Blue=MPI, Yellow=Best MPI, Orange=SHMEM, Green=Best SHMEM.

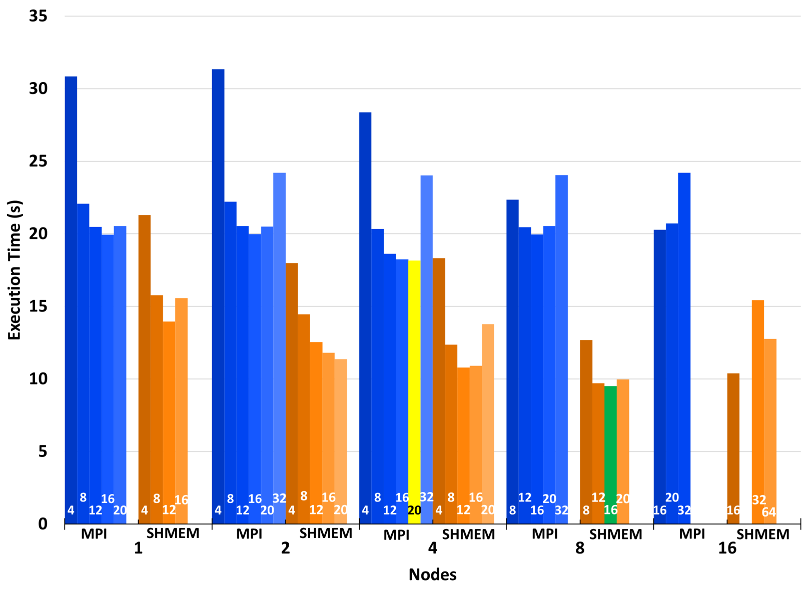

Fig. 5. MPI vs. OpenSHMEM performance for the it-2004 webgraph. Bar labels denote PEs. Blue=MPI, Yellow=Best MPI, Orange=SHMEM, Green=Best SHMEM.

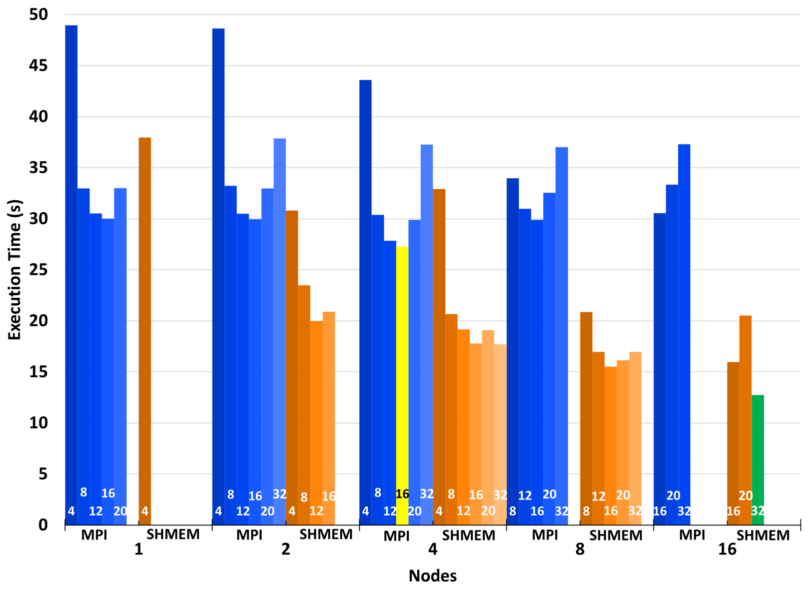

Fig. 6. MPI vs. OpenSHMEM performance for the sk-2005 webgraph. Bar labels denote PEs. Blue=MPI, Yellow=Best MPI, Orange=SHMEM, Green=Best SHMEM.

5.3 Productivity Studies

Table 4. Implementation productivity, measured in LOC and API-calls.

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**

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 4.

6 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 Level

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. 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 process counts. The *all-reduce* operation is more nuanced. The OpenSHMEM implementation outperforms *MPI_allreduce* for message sizes larger than 4KB and processor counts greater than 2. While the discrepancies for collective operations are more significant (an average of 45.1% decrease in latency for *barrier-all* and *all-reduce* compared to only ∼2.5% decrease for *put* and *get*), these decreases 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 minor improvement alone isn't enough to justify an 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 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 [3].

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 addition introduced more overhead in the form of additional API calls as well as lines of code.

One drawback of using OpenSHMEM is that the OpenSHMEM specification version 1.4 only supports "to-all" communication for many collective API-calls, meaning all processes receive data from every communication [9]. This fact 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 sequential point-to-point operations. As a result, all "to-one" communications in the algorithm were 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 utilize strategic programming techniques, non-blocking communication, and RMA to maximize uninterrupted computation 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 optimizations.

At all node counts, both MPI and OpenSHMEM implementations display the best performance improvement at either 16 or 20 PEs with the exception of the uk-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 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 LOC 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 prospective developers. As apps and databases increase in scale, distributed-computing systems will become even more prominent. In turn, the OpenSHMEM specification will continue to grow in viability as a means for parallel performance improvement.

8 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 technology allows GPUs to directly communicate with one another, avoiding the CPU communication bottleneck [7]. In addition to the acceleration displayed in this work with non-blocking RMA 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 extension would more robustly explore the performance improvement potential of the MND-MST algorithm, and would combine two SHMEM libraries at a larger scale.

References

1. Boldi, P., Vigna, S.: The WebGraph framework I: Compression techniques. In: Proc. of the Thirteenth International World Wide Web Conference (WWW 2004). pp. 595–601. ACM Press, Manhattan, USA (2004)
2. Borúvka, O.: O jistém problému minimálním [about a certain minimal problem] **5**(3), 37–58 (1926)

3. Chapman, B., Curtis, T., Pophale, S., Poole, S., Kuehn, J., Koelbel, C., Smith, L.: Introducing openshmem: Shmem for the pgas community. In: Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model. PGAS '10, Association for Computing Machinery, New York, NY, USA (2010), https://doi.org/10.1145/2020373.2020375
4. Friesen, B.: Cori system - nersc documentation (2020), https://docs.nersc.gov/systems/cori/
5. Fu, H., Gorentla Venkata, M., Salman, S., Imam, N., Yu, W.: 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). https://doi.org/10.1109/CCGRID.2018.00078
6. Gropp, W.D., Thakur, R.: Revealing the performance of mpi rma implementations. In: Cappello, F., Herault, T., Dongarra, J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. pp. 272–280. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
7. Hsu, C.H., Imam, N.: Assessment of nvshmem for high performance computing. International Journal of Networking and Computing **11**(1), 78–101 (2021). https://doi.org/10.15803/ijnc.11.1$_7$8
8. Jose, J., Zhang, J., Venkatesh, A., Potluri, S.: A comprehensive performance evaluation of openshmem libraries on infiniband clusters. In: OpenSHMEM Workshop (2014)
9. Laboratory, O.R.N., Laboratory, L.A.N.: Openshmem application programming interface version 1.4 (Dec 2017)
10. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. p. 135–146. SIGMOD '10, Association for Computing Machinery, New York, NY, USA (2010), https://doi.org/10.1145/1807167.1807184
11. Nesetril, J.: A few remarks on the history of mst-problem. Archivum Mathematicum **33**(1), 15–22 (1997)
12. Panda, D.K., Subramoni, H., Chu, C.H., Bayatpour, M.: The mvapich project: Transforming research into high-performance mpi library for hpc community. Journal of Computational Science p. 101208 (2020). https://doi.org/https://doi.org/10.1016/j.jocs.2020.101208
13. Panja, R., Vadhiyar, S.: Mnd-mst: A multi-node multi-device parallel boruvka's mst algorithm. In: Proceedings of the 47th International Conference on Parallel Processing. ICPP 2018, Association for Computing Machinery, New York, NY, USA (2018), https://doi.org/10.1145/3225058.3225146
14. Schulz, M., Bland, W., Gropp, W., Rabenseifner, R., Bangalore, P., Blaas-Schenner, C., Bosilca, G., Grant, R., Hermanns, M.A., Holmes, D., Mercier, G., Pritchard, H., Skjellum, A.: Mpi: A message passing interface standard 2019 draft spec (Nov 2019)
15. Stitt, T.: An Introduction to the Partitioned Global Address Space (PGAS) Programming Model. OpenStax CNX (March 2020), http://cnx.org/contents/82d83503-3748-4a69-8d6c-50d34a40c2e7@7
16. Wang, G., Lam, H., George, A., Edwards, G.: Performance and productivity evaluation of hybrid-threading hls versus hdls. In: 2015 IEEE High Performance Extreme Computing Conference (HPEC). pp. 1–7 (2015). https://doi.org/10.1109/HPEC.2015.7322439
17. Yan, D., Cheng, J., Lu, Y., Ng, W.: Effective techniques for message reduction and load balancing in distributed graph computation. In: Proceedings of the 24th International Conference on World Wide Web. p. 1307–1317. WWW '15, International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE (2015), https://doi.org/10.1145/2736277.2741096