LA-UR-16-29614
Approved for public release; distribution is unlimited.

Title: Graph 500 on OpenSHMEM: Using a Practical Survey of Past Work to Motivate Novel Algorithmic Developments

Author(s): Grossman, Max
Pritchard, Howard Porter Jr.
Budimlic, Zoran
Sarkar, Vivek

Intended for: Report

Issued: 2016-12-22 (rev.1)

Graph500 on OpenSHMEM: Using A Practical Survey of Past Work to Motivate Novel Algorithmic Developments

Max Grossman*[1], Howard Pritchard[2], Zoran Budimlić[1], and Vivek Sarkar[1]

[1]Rice University, Houston, TX, USA
[2]Los Alamos National Laboratory, Los Alamos, NM, USA

December 22, 2016

1 Introduction

Graph500 [14] is an effort to offer a standardized benchmark across large-scale distributed platforms which captures the behavior of common communication-bound graph algorithms. Graph500 differs from other large-scale benchmarking efforts (such as HPL [6] or HPGMG [7]) primarily in the irregularity of its computation and data access patterns.

The core computational kernel of Graph500 is a breadth-first search (BFS) implemented on an undirected graph. The output of Graph500 is a spanning tree of the input graph, usually represented by a predecessor mapping for every node in the graph. The Graph500 benchmark defines several pre-defined input sizes for implementers to test against.

This report summarizes investigation into implementing the Graph500 benchmark on OpenSHMEM, and focuses on first building a strong and practical understanding of the strengths and limitations of past work before

*jmg3@rice.edu

proposing and developing novel extensions. While our initial investigation explores implementations in pure OpenSHMEM, we also describe work implementing Graph500 using the AsyncSHMEM framework described in [13]. AsynSHMEM is a hybrid programming model and runtime system that composes OpenSHMEM and asynchronous task parallelism in order to enable novel APIs and functionality. For more details on AsyncSHMEM, please refer to the original publication [13] or the source code available at [1].

The remainder of this report is structured as follows. Section 2 discusses the past work in this area. Section 3 describes the most relevant/successful OpenSHMEM-based implementations we have explored to date. Section 4 discusses how we use the AsyncSHMEM framework [13] to improve the developed reference OpenSHMEM implementations described in Section 3. Section 5 compares the performance of various OpenSHMEM-based and reference implementations.

2 Past Work

Previous work on G500 can be broken into three categories: the available distributed reference implementations (all MPI-based), various research implementations in MPI, and various research implementations in OSHMEM.

2.1 MPI-based Reference Implementations

`graph500.org` offers several reference MPI implementations of the Graph500 benchmark. The provided reference implementations are summarized below:

1. **Simple**: Generally considered the baseline implementation, Simple uses two-sided MPI to implement a straightforward wavefront-based BFS. Nodes in the graph are partitioned across MPI ranks, and at each layer in the wavefront nodes in the next wavefront are transmitted to the MPI ranks which own them. This implementation can be difficult to read, as new receives are detected by inserting periodic polling (`MPI_Test`) in the code. Simple is also not a well-performing or scaling implementation, nor is it intended to be. Mostly, it is an illustration of a classical implementation of BFS in MPI.

2. **One-Sided**: This implementation uses MPI one-sided APIs to implement an algorithm similar to the Simple implementation. In our

experience, it only works on very small datasets and does not perform well even at those scales. There is some literature that compares against it [9] (discussed later), which also reports that in their tests the One-Sided implementation ran out of memory at small scales (datasets smaller than even Graph 500's "Toy" dataset).

3. **Replicated**: The Replicated implementation is the least conventional of the reference implementations in that it isn't message-driven. Instead, it communicates bit masks among all MPI ranks using `MPI_AllGather`, setting bits to indicate which nodes in the graph are to be visited in the next wavefront. There is no overlap of computation or communication in this version. Replicated is unusual among most G500 implementations in that it is not entirely communication-bound, and has enough computational work in each processing element (PE) to benefit from multi-threaded parallelism. However, communication does still take up a significant amount of application time, particularly two calls to `MPI_Allgather`. Replicated is the best performing reference implementation in our experience, and so our evaluation uses it as a baseline.

2.2 Research MPI Implementations

The main research implementation of note in MPI is from [8]. For the purposes of this paper, we will refer to this implementation as the **Tuned** implementation. **Tuned** is available as an open source distribution available from the Graph 500 website. In our investigation, we have found **Tuned** to be a well-scaling implementation of Graph500 in MPI, but also note that it has some stability issues which make it difficult to evaluate against.

For example, when compiling the Tuned version you build several executables from the same source. Each executable is specialized to a certain number of MPI ranks. The open source distribution comes with a list of supported numbers of ranks, but when trying to compile with any other number of ranks the build recurses infinitely on both Intel and GNU compilers.

The Tuned implementation also comes with built-in self-tuning, in that it performs many training runs while tweaking internal parameters until it finds what it thinks is the best for the current platform, and then performs the actual BFS kernel. This makes comparisons against other, more general-purpose implementations unfair. However, in the experiments we were able to perform with the Tuned implementation we see it performing similarly to

the Replicated implementation, even when allowing Tuned to tune itself.

2.3 Research OSHMEM Implementations

There are two primary past works describing OpenSHMEM-based G500 implementations. Neither are open source. [9] describes a straight port of the MPI One-Sided implementation to OpenSHMEM, and much of the paper focuses on challenges around resolving differences between the support offered by MPI APIs and OpenSHMEM APIs. While this paper offers motivation for extending certain OpenSHMEM APIs, there are no novel G500 algorithmic contributions.

[12] describes an OpenSHMEM-based implementation that is inspired by the Simple implementation but with various improvements using RDMA. Section 4 of [12] describes their implementation. While this paper inspired some of the investigation described here, there are also a couple of open questions with regards to this paper. For example, in it the authors describe a packet format for transmitting chunks of vertices between PEs for setting up the next wavefront. On the receiver side, the arrival of a packet is detected by polling for a special header token, then polling for a special footer token, and then assuming the entire packet is there. This assumes ordered delivery of bytes and is inconsistent with both the OpenSHMEM specification and the guarantees made by most modern network hardware.

3 OpenSHMEM Graph500 Implementations

Based on the related works described in Section 2, we developed three from-scratch implementations of Graph 500 in pure, flat OpenSHMEM. We refer to these implementations as **Bitvector**, **Concurrent-Fence**, and **Concurrent-Hash**.

3.1 Bitvector

The Bitvector implementation was inspired by the Replicated reference implementation, and uses bitvectors to communicate traversed vertices among PEs at each wavefront level. Recall that Replicated was the best performing MPI reference implementation, and while this algorithmic approach does not take advantage of one-sided RDMA in OSHMEM, a direct port of the

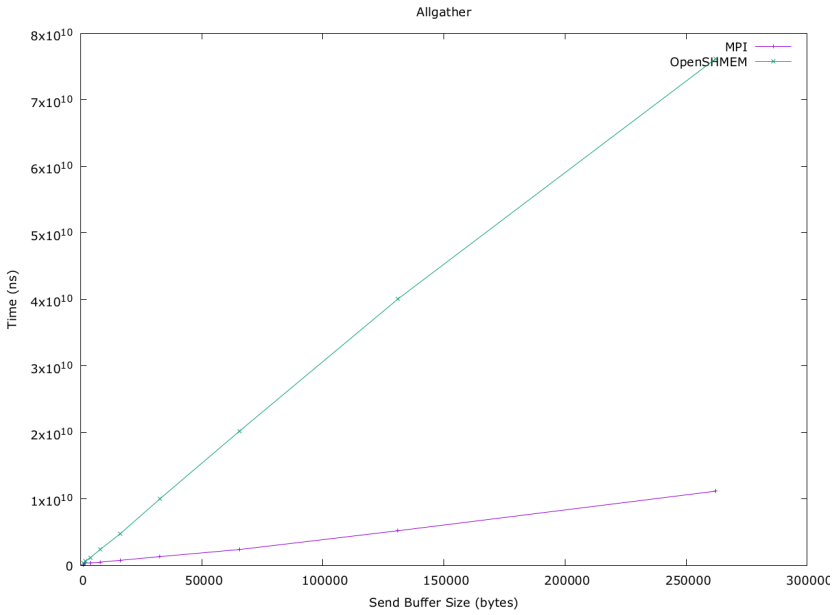

Figure 1: Time to complete 10 all-gather operations using Cray MPICH and Cray SHMEM on Titan with various send buffer sizes.

best performing MPI implementation is an important point of comparison. However, we found that the OSHMEM version performed poorly because of scalability differences between `MPI_Allgather` and `shmem_fcollect` performance on the Titan supercomputer at ORNL [5]. To quantify this performance difference, we wrote a microbenchmark comparing `MPI_Allgather` and `shmem_fcollect` on Titan (source code available at [10] and [11]). Figure 1 plots the time to do 10 all-gathers at varying send buffer sizes. Figure 2 plots the same data, but on a log scale. All experiments were run with Cray MPICH 7.4.0 and Cray SHMEM 7.4.0.

Note that in Figure 1 and 2 there is a large discrepancy between MPI and OpenSHMEM. At the buffer sizes that we are interested in for Graph500 (KBs to MBs) Cray's OpenSHMEM all-gather implementation is 5-10$\times$ slower. At the end of the day, this prevents an efficient port of the Replicated MPI version to OpenSHMEM. As a result, the Bitvector OpenSHMEM implementation is a dead end for the time being.

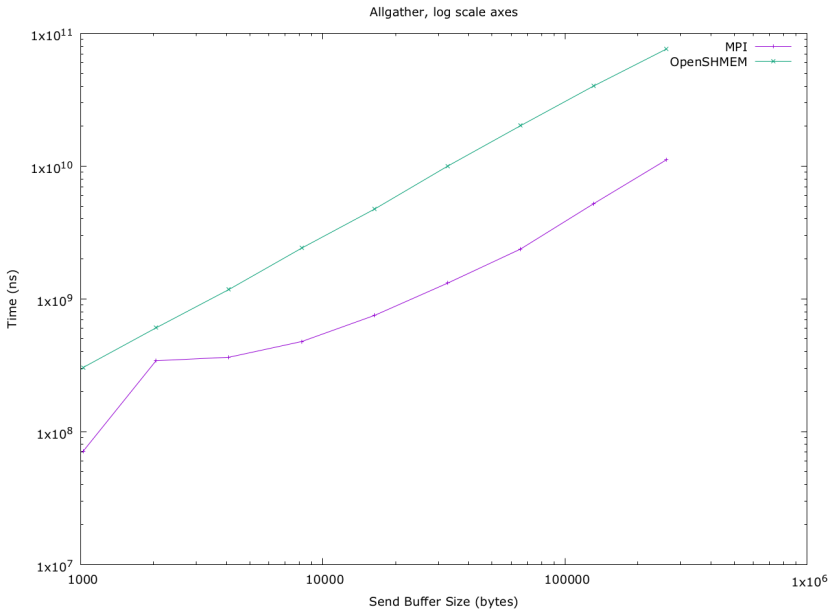

Figure 2: Time to complete 10 all-gather operations using Cray MPICH and Cray SHMEM on Titan with various send buffer sizes. Axes are in log scale.

3.2 Concurrent-Fence

The Concurrent-Fence implementation was inspired by the work described in [12].

Concurrent-Fence is wavefront-based, and distributes nodes in the graph across PEs. Every PE has four primary data structures: an active queue, an inactive queue, a receive buffer, and a linked list of send buffers.

At the start of each BFS wavefront, the following properties hold:

1. The active queue contains a list of vertices in the graph that are owned by this PE and which should be traversed in the current wavefront.

2. The inactive queue is completely empty.

3. The receive buffer is completely empty.

4. The list of send buffers is pre-populated with many send buffers, each sized to hold on the order of hundreds of (vertex, parent) tuples for transmission to remote PEs. Send buffers are used to notify remote PEs of vertices that they should traverse in the next wavefront, as the local PE encounters them as neighbors of vertices in the current wavefront.

The processing of the vertices stored in the active queue involves traversing the queue and iterating over the neighbors of each vertex in it. If the local PE knows a given neighbor vertex has already been visited, it is skipped. Otherwise, unvisited neighbors are added to the send buffer being filled for the PE that owns the unvisited vertex. Send buffers are allocated from the pre-allocated send buffer list when necessary. When a send buffer is filled during traversal of the active list, it is asynchronously sent to the target PE using an OSHMEM non-blocking put.

At the same time as we are traversing the active queue on each PE, each PE also periodically checks for incoming send buffers from remote PEs. Incoming send buffers are placed in a PE's receive buffer by the transmitting PE. A remote atomic add is used on a receive buffer index to reserve space in the receive buffer for each inbound send buffer. The inbound send buffer is then placed in that reserved space. The format of a send buffer is depicted in Figure 3, which is similar to the format described in [12].

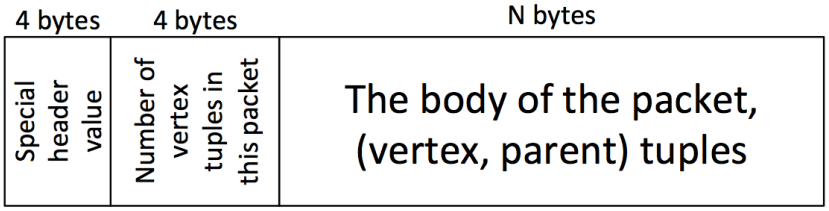

Figure 3: Format of a send buffer in the Concurrent-Fence implementation.

On the sending side, a single send buffer is transmitted as soon as it is filled. First, the size and body of the buffer are transmitted in a single non-blocking PUT. Then, a `shmem_fence` is issued to ensure ordering. Finally, the special header value is sent, also with a non-blocking PUT. On the receiving side, the PE periodically polls for the special header value in its receive buffer. If it is found, the `shmem_fence` on the sending side ensures that the entire body of the buffer has been received. If a PE detects a buffer has arrived in its receive buffer, it processes this buffer by adding any unvisited vertices to its inactive queue.

For termination detection, each PE sends a special "empty" buffer to every other PE. This empty buffer signals to the receiving PE that a PE has completed its processing of its active queue, and includes a count of the number of new vertices transmitted to other PEs by that PE in the current wavefront. Hence, each PE will wait to receive `shmem_n_pes() - 1` "empty buffers" and then terminate the BFS when no PE sent any new vertices on the current wavefront. This operation is similar to a fuzzy barrier, though it has the unfortunate property that the number of bytes transmitted scales exponentially with the number of PEs used (i.e. every PE sends to every other PE). Concurrent-Fence has also been implemented with a simpler scheme that uses a `shmem_int_sum_reduce` followed by a `shmem_barrier_all` for termination detection. However, in our experience these two versions perform comparably at the scales tested.

If termination has not been reached, the active and inactive queues are swapped at the end of each wavefront, all send buffers are restored to the send buffer list, and the receive buffer index is reset to zero.

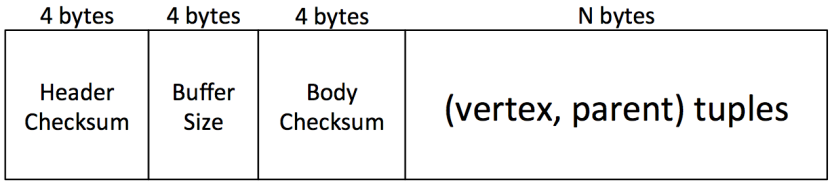

Figure 4: Format of a send buffer in the Concurrent-Hash implementation.

3.3 Concurrent-Hash

The Concurrent-Hash implementation was motivated by the observation that the Concurrent-Fence implementation performs a large number of `shmem_fence` calls, which impedes scalability. It primarily differs from the Concurrent-Fence version in its buffer format, and in how it detects buffer arrival.

The new buffer format is depicted in Figure 4.

Essentially, the goal of this new format is to allow a single send buffer to be sent in a single non-blocking PUT without requiring a fence, while still giving the receiving PE a way to detect receipt of an entire send buffer. The key is the Body Checksum field in Figure 4 which checksums the contents of the body, and the Header Checksum field which checksums the Buffer Size and Body Checksum fields. Before transmitting a send buffer, the sender PE must compute these checksums on the complete send buffer. On the receiving end, the receiving PE no longer polls on a special header value (as it did in the Concurrent-Fence implementation), but rather does the following:

1. Reads the next four bytes in the receive buffer, and for now assumes that is a checksum of a buffer header.

2. Reads the following eight bytes in the receive buffer and computes their checksum. If this does not match the previous four bytes, we can assume no send buffer has arrived and abort processing the receive buffer.

3. Otherwise, we then use the Buffer Size stored in the header to calculate the size of the body of this buffer and compute the checksum of those bytes in the receive buffer. If the calculated body checksum matches the Body Checksum field in the header, the whole buffer has arrived. Otherwise, it has not and we abort.

In this approach, the checksum computation on the sending and receiving ends could become a computational bottleneck. As a result, experiments were run with several high-performance hashing algorithms, including CRC32, MurmurHash [3], and CityHash [2]. In the end, CityHash was found to perform the best for this type of data, and performance profiling shows that each PE generally only spends ∼1% of execution time on hashing during a Concurrent-Hash run of Graph500.

4 AsyncSHMEM Implementations

All work described in this section was done using the Offload implementation of AsyncSHMEM, described in 4.

As mentioned in previous sections, G500 is generally a communication-bound algorithm. Only in special cases (Replicated) is there enough computational work that G500 can benefit from multi-threaded parallelism, and unfortunately the current performance of `shmem_fcollect` prevents us from building a well-performing OpenSHMEM implementation based on those techniques (see Section 3.1).

Hence, while previous application studies with AsyncSHMEM have focused on the use of asynchronous task parallelism for hybrid applications [13], in G500 we focus more on concurrency and programmability. That is, using a single runtime thread per PE with computation and communication multiplexed cooperatively on it by the AsyncSHMEM runtime.

As part of this work, we built AsyncSHMEM implementations of both the Concurrent-Fence and Concurrent-Hash OpenSHMEM versions.

The changes made to create the AsyncSHMEM versions of Concurrent-Fence and Concurrent-Hash are similar. In particular, we eliminate the need to periodically poll for new incoming messages by using the novel `shmem_async_when` family of APIs described in previous work [13]:

```
void shmem_int_async_when(volatile int *ivar, int cmp,
        int cmp_value, void (*callback)(void));
```

This API is similar to the `shmem_wait` family of APIs, in that it triggers local operations by comparing the value at the symmetric heap location `ivar` to `cmp_value` using the comparison operator specified by `cmp`. However, rather than blocking the current thread like `shmem_wait`, `shmem_async_when` triggers an asynchronous task when the specified condition is satisfied.

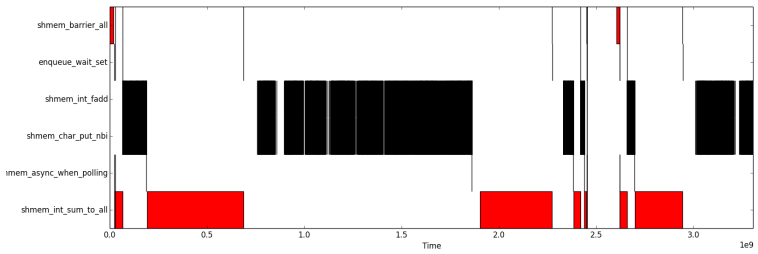

Figure 5: A trace of OpenSHMEM calls and their elapsed time inside a single PE from an execution of Concurrent-Hash on 128 nodes.

In Concurrent-Fence and Concurrent-Hash, we use the `shmem_async_when` API to trigger tasks when new incoming messages are detected by either triggering them on the special header value in Concurrent-Fence, or on a change in the value stored at the location where we expect a header checksum to appear in Concurrent-Hash.

This change reduces clutter in the code, and hands over the problem of polling intervals and other scheduling issues to the global AsyncSHMEM scheduler. It also improves the ability of the AsyncSHMEM system to visualize the application workload by exposing more semantic information to the runtime. For example, Figure 5 plots a trace from a single PE running the Concurrent-Hash implementation. This trace was collected using AsyncSHMEM-specific, low-overhead tracing capabilities. Note that the y-axis corresponds with specific OpenSHMEM APIs, offering more high-level, semantic information to the programmer about where their application is spending time.

5 Performance Evaluation

For the evaluation reported in this section, all runs use a graph with 2^{29} nodes. All experiments are performed on the Titan supercomputer at ORNL [5]. Cray MPICH 7.4.0 and Cray SHMEM 7.4.0 are used in all experiments.

We compare the reference Simple and Replicated MPI implementations to our own Concurrent-Fence and Concurrent-Hash OpenSHMEM implementations. We compare against Simple and Replicated with both 1 core per MPI rank (i.e. flat MPI) and with 16 cores per MPI rank (i.e. hybrid MPI + OpenMP).

The comparison to Simple with 1 core per rank is the most apples-to-apples, as it is algorithmically the most similar MPI-based implementation to our OpenSHMEM versions (and hence has the most similar communication patterns). Using 1 core per rank also makes it a more fair comparison, as the number of PEs will scale equally for both implementations, whereas with 16 cores per rank there are 16× fewer ranks communicating.

The comparison to Replicated with 1 core per rank is done as a one-to-one comparison of the scalability of the algorithm implemented by Algorithm compared to the algorithm implemented by Concurrent-Fence/Hash. While the two implementations are significantly different algorithmically and in terms of the communication performed, it is important to compare to the highest performing alternative.

While we have not to-date constructed a hybrid, multi-threaded implementation of either Concurrent-Fence or Concurrent-Hash, we also compare against both Simple and Replicated with hybrid parallelism in order to be certain we are measuring the peak performance possible, given the same hardware resources. However, we note that using hybrid parallelism reduces the number of ranks communicating, and is therefore a weaker test of scalability.

Figure 6 plots strong scaling curves for all six implementations. Simple-1 and Replicated-1 refers to those implementations with 1 core per PE, while Simple-16 and Replicated-16 refers to those implementations with 16 cores per PE. Note that the y axis is log scale in Figure 6, and that none of these implementations use AsyncSHMEM.

First, we note that Concurrent-Fence and Concurrent-Hash are both able to beat flat MPI (i.e. Simple-1 and Replicated-1), and that flat MPI stops scaling after 128 nodes.

Second, we note that while Concurrent-Fence does continue to improve as you add more nodes, Concurrent-Hash's removal of excessive `shmem_fence` calls leads to better scaling.

Third, we note that only Replicated-16 outperforms the flat implementations Concurrent-Fence and Concurrent-Hash as a result of fewer MPI ranks and less inter-rank communication. However, from 128 to 256 nodes our Concurrent-Hash implementation improves by 33% while the Replicated-16 implementation only improves by 13%. Hence, despite the fact that Concurrent-Hash is running with 16× as many PEs, it is still scaling better than Replicated-16.

Figure 7 reports overheads from using the AsyncSHMEM runtime to schedule triggered asynchronous tasks rather than using manual polling. In

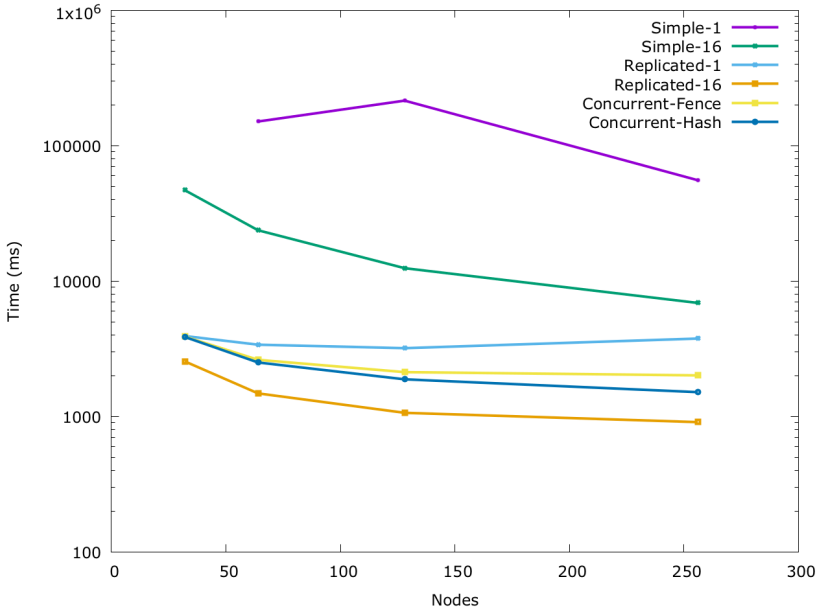

Figure 6: Scaling of several Graph 500 implementations. Note that the y axis is log scale.

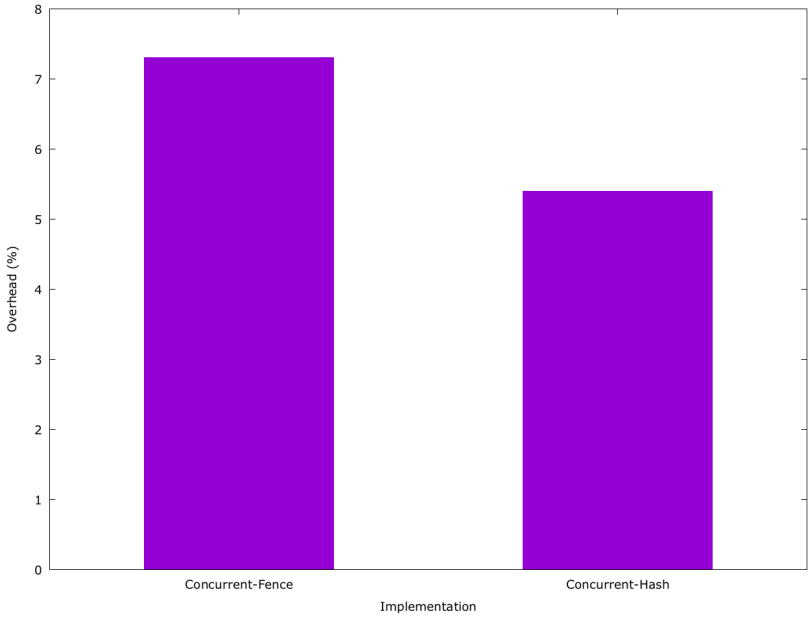

Figure 7: AsyncSHMEM Overheads.

general, we see approximately 5% overhead. While this is larger than desired, it is an upper limit. There has not yet been any exploration of more complex, cooperative, non-greedy scheduling algorithms for computation and communication on the AsyncSHMEM runtime. Most of this overhead likely comes from memory allocation system calls needed for task creation in the runtime, which are unnecessary in a manual polling approach.

6 Conclusions & Future Work

This work summarizes the state of the world today for the Graph500 benchmark and details our own investigation into running it on OpenSHMEM and AsyncSHMEM. While for flat distributed programs OpenSHMEM has demonstrated a significant performance and scaling benefit, when comparing against a hybrid MPI+OpenMP implementation the reduction in communication from the hybrid approach leads to speedup over our flat OpenSHMEM-based implementations. Unfortunately, it seems the current performance of OpenSHMEM all-gather operations limits our ability to implement a similarly structured version of Graph500 which can make use of intra-process

parallelism.

While past work on irregular applications on AsyncSHMEM has focused on the performance improvement possible from added parallelism, this work instead demonstrates benefit from using AsyncSHMEM for programmability and better tooling.

Future work in executing Graph500 on OpenSHMEM must focus on developing hybrid versions of the benchmark that combine multi-threaded and multi-process parallelism. While the current version of the OpenSHMEM specification [4] offers no way to use OpenSHMEM APIs in a thread-safe manner, ongoing discussions in the OpenSHMEM Threading Working Group are actively working to rectify that. That change may enable additional novel work in Graph500 on OpenSHMEM.

Acknowledgments

This research was funded in part by the United States Department of Defense, and was supported by resources at Los Alamos National Laboratory.

References

[1] Asyncshmem source code. `https://github.com/habanero-rice/hclib/tree/resource_workers`.

[2] CityHash. `https://github.com/google/cityhash`.

[3] MurmurHash. `https://github.com/aappleby/smhasher`.

[4] Openshmem application programming interface version 1.3. `http://openshmem.org/site/sites/default/site_files/OpenSHMEM-1.3.pdf`.

[5] Titan. `https://www.olcf.ornl.gov/titan/`.

[6] Top 500. `https://www.top500.org/`.

[7] M. Adams. Hpgmg 1.0: a benchmark for ranking high performance computing systems. 2014.

[8] F. Checconi and F. Petrini. Massive data analytics: the graph 500 on ibm blue gene/q. *IBM Journal of Research and Development*, 57(1/2):10–1, 2013.

[9] E. F. D'Azevedo and N. Imam. Graph 500 in openshmem. In *Workshop on OpenSHMEM and Related Technologies*, pages 154–163. Springer, 2014.

[10] M. Grossman. Mpi allgather microbenchmark. `https://github.com/habanero-rice/hclib/blob/resource_workers/test/performance-regression/cpu-only/graph500-2.1.4/oshmem/mpi_allgather.c`.

[11] M. Grossman. Openshmem allgather microbenchmark. `https://github.com/habanero-rice/hclib/blob/resource_workers/test/performance-regression/cpu-only/graph500-2.1.4/oshmem/oshmem_allgather.c`.

[12] J. Jose, S. Potluri, K. Tomko, and D. K. Panda. Designing scalable graph500 benchmark with hybrid mpi+ openshmem programming models. In *International Supercomputing Conference*, pages 109–124. Springer, 2013.

[13] Max Grossman, Vivek Kumar, Zoran Budimlic, Vivek Sarkar. Integrating Asynchronous Task Parallelism with OpenSHMEM. In *OpenSHMEM Workshop*, 2016.

[14] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the graph 500. *Cray User's Group (CUG)*, 2010.