Designing a High Performance OpenSHMEM Implementation Using Universal Common Communication Substrate as a Communication Middleware

Pavel Shamis[1], Manjunath Gorentla Venkata[1], Stephen Poole[1], Aaron Welch[2], and Tony Curtis[2]

[1] Extreme Scale Systems Center (ESSC)
Oak Ridge National Laboratory (ORNL)
{shamisp,manjugv,spoole}@ornl.gov
[2] Computer Science Department
University of Houston (UH)
{dawelch,arcurtis}@uh.edu

Abstract. OpenSHMEM is an effort to standardize the well-known SHMEM parallel programming library. The project aims to produce an open-source and portable SHMEM API and is led by ORNL and UH. In this paper, we optimize the current OpenSHMEM reference implementation, based on GASNet, to achieve higher performance characteristics. To achieve these desired performance characteristics, we have redesigned an important component of the OpenSHMEM implementation, the network layer, to leverage a low-level communication library designed for implementing parallel programming models called UCCS. In particular, UCCS provides an interface and semantics such as native atomic operations and remote memory operations to better support PGAS programming models, including OpenSHMEM. Through the use of microbenchmarks, we evaluate this new OpenSHMEM implementation on various network metrics, including the latency of point-to-point and collective operations. Furthermore, we compare the performance of our OpenSHMEM implementation with the state-of-the-art SGI SHMEM. Our results show that the atomic operations of our OpenSHMEM implementation outperform SGI's SHMEM implementation by 3%. Its RMA operations outperform both SGI's SHMEM and the original OpenSHMEM reference implementation by as much as 18% and 12% for gets, and as much as 83% and 53% for puts.

1 Introduction

OpenSHMEM [1] [2] is an effort towards creating an open standard for the well-known SHMEM library, and is a starting point to accommodate future extensions to the SHMEM API. SHMEM is a Partitioned Global Address Space (PGAS) based parallel programming model. OpenSHMEM 1.0 is a SHMEM specification based on SGI's SHMEM API, which predominantly supports one-sided communication semantics as well as providing collective communication operations, atomic operations, and synchronization operations. Currently, there are many production-grade proprietary implementations of the SHMEM API. OpenSHMEM 1.0 is an effort to create an open, unified standard and a reference implementation [3] of the SHMEM API, led by ORNL's ESSC and UH.

The current reference implementation, which supports various network interfaces in an effort to be portable to spur adoption, is based on the GASNet communication middleware [4]. Though proprietary SHMEM implementations have outstanding performance characteristics on their native hardware, the current reference implementation of OpenSHMEM has several performance drawbacks. For example, atomic operations in the current implementation have a latency that is at least 27% slower than that of the native low-level drivers. In this paper, in order to arrive at the desired performance characteristics, we redesign the network layer of the OpenSHMEM reference implementation to leverage the Universal Common Communication Substrate (UCCS) communication library [5] [6], a low-level network library for implementing parallel programming models. For the rest of the paper, the current reference implementation will be called *OpenSHMEM-GASNet* and the new reference implementation using UCCS will be referred to as *OpenSHMEM-UCCS*.

The rest of the paper is organized as follows: Section 2 provides a brief overview of the UCCS communication middleware and OpenSHMEM specification. Section 3 discusses related works in the area of OpenSHMEM implementations. Section 4.1 details the network layer of OpenSHMEM, and the way it was designed to be independent of underlying network infrastructure. Section 4.2 provides details of UCCS interfaces, data structures, and semantics of operations, and Section 4.3 provides the details of its integration with OpenSHMEM. Section 5 provides an evaluation of the *OpenSHMEM-UCCS* implementation by comparing it to *OpenSHMEM-GASNet* and SGI's SHMEM, and we present concluding remarks in Section 6.

2 Background

This section provides a brief background for the OpenSHMEM specification and UCCS communication middleware.

2.1 OpenSHMEM

Despite the fact that the original SHMEM library was designed by Cray Research, which later was merged with Silicon Graphics (SGI), there are multiple variants of the SHMEM API that have been introduced by different system and hardware vendors. The SGI SHMEM library, which is a part of SGI's Message Passing Toolkit (MPT), provides the original SHMEM interface developed by Cray Research and SGI. Cray provides a SHMEM library implementation for the SeaStar, Aries, and Gemini interconnects. HP supports SHMEM concepts with the HP SHMEM library, which is based on the Quadrics SHMEM library and is available on HP systems. Despite the broad availability of SHMEM implementations, the SHMEM API has not been standardized. As a result, application developers have to handle incompatibilities of different SHMEM implementations at the application level. The OpenSHMEM specification was borne out of the desire to standardize the many similar yet incompatible SHMEM communication libraries into a single API. The OpenSHMEM reference implementation is an open-source implementation of this specification.

2.2 UCCS

UCCS is a communication middleware that aims to provide a high performing low-level communication interface for implementing parallel programming models. UCCS aims to deliver a broad range of communication semantics such as active messages, collective operations, puts, gets, and atomic operations. This enables implementation of one-sided and two-sided communication semantics to efficiently support both PGAS and MPI-style programming models. The interface is designed to minimize software overheads, and provide direct access to network hardware capabilities without sacrificing productivity. This was accomplished by forming and adhering to the following goals:

– Provide a universal network abstraction with an API that addresses the needs of parallel programming languages and libraries.
– Provide a high-performance communication middleware by minimizing software overheads and taking full advantage of modern network technologies with communication-offloading capabilities.
– Enable network infrastructure for upcoming parallel programming models and network technologies.

In order to evaluate the OpenSHMEM reference implementation with UCCS instead of GASNet, the reference implementation was extended to integrate with the UCCS network substrate.

3 Related Work

The OpenSHMEM reference implementation is the first open-source implementation of the OpenSHMEM specification, which was developed in conjunction with the specification by the University of Houston and Oak Ridge National Laboratory. Since the OpenSHMEM specification is based on SGI's SHMEM API, SGI SHMEM was the first commercial implementation of the specification.

The HPC community and system vendors embraced the specification and released various implementations of the specification. The University of Florida is developing the GSHMEM [7] project which is an OpenSHMEM implementation based solely on the GASNet runtime. Ohio State University (OSU) distributes the MVAPICH-X runtime environment [8], which is focused on enabling MVAPICH InfiniBand and iWARP support for OpenSHMEM and UPC in addition to MPI. Sandia National Laboratory provides an open source implementation of the specification for the Portals [9] network stack. In addition to the above implementations, there are SHMEM implementations that are tightly coupled to particular network technologies developed by network and system vendors. Mellanox ScalableSHMEM [10] provides support for a family of Mellanox interconnects and is based on proprietary software accelerators [11]. Cray and HP SHMEM provide proprietary SHMEM implementations for platforms developed by these vendors, both of which have been making steps toward supporting the OpenSHMEM specification. Tilera Many-Core processors are supported within TSHMEM [12].

The OpenSHMEM reference implementation differentiates itself as an open-source and explorable community platform for development and extension of the OpenSHMEM Specification. Using high-performance UCCS middleware, the UCCS reference implementation aims to deliver performance that is as good or better than the state-of-the-art commercial implementations.

4 Design

The design section discusses the OpenSHMEM communication layer, the UCCS API, and the integration of UCCS as a communication layer in OpenSHMEM.

4.1 OpenSHMEM Communication Layer

The OpenSHMEM reference implementation consists of the core API and a separate communication layer that can have multiple implementations for handling low-level networking hardware, designed in a way that allows them to be easily swapped out for one another. This separation between layers was done in a way that was as minimalistic as reasonably possible, yet still would be generic enough so as to be able to fully accommodate any potential communication infrastructure cleanly. This allows for the maximum amount of common functionality to be reused across communication layer implementations, but requires careful construction.

Moving in this direction, the first task involved dividing all communication primitives such as puts, gets, and atomics between network-agnostic and network-specific sections. Additionally, any particular functions that multiple network layers may share were also taken out and implemented in the upper layers of the library. These functions include a handful of basic procedures that also do not include any assumptions or code specific to a particular communication infrastructure, but may still be needed during use, such as range checking for processing element (PE) numbers, symmetric addressing, memory allocation and deallocation, and state information querying.

Other communication calls that could be implemented using the previously described primitives were done so in the upper layers of the library as well, removing additional strain on the requirements for generating and maintaining communication layers. Most notably, these include collective calls such as barrier, broadcast, and reduction operations. Finally, any functions that do not require any communication at all were implemented solely in the upper layer, such as `shmem_wait()`.

4.2 UCCS API

The UCCS API is a generic and network hardware agnostic interface that defines abstract concepts, which isolate programmers from hardware specific details. To accomplish this, it first defines a few key concepts: communication contexts, resources, and endpoints. Communication contexts are a method of providing communication scope and isolation to multiple instances of user or system code, and are represented in application code as opaque handles. These handles contain all information about the associated communication context, including resources, endpoints, and memory, and are at the topmost layer of the communication abstraction. Resources represent a particular communication channel available for a network, for which there may be several for a given network in cases such as multi-rail architectures. Similar to communication contexts, resources are also represented by opaque handles, and all the descriptors for the available resources of a particular transport type or list of types in a given communication context are initialized at once.

In order to complete a communication call, a specific endpoint must be selected to communicate with using a specific resource. These endpoints represent the ultimate destination for a communication operation, and are also represented by opaque handles. To obtain a set of valid endpoints, a connectivity map must be generated for a resource and be queried to discover if a particular execution context is reachable via the resource for which it was generated. Endpoints in UCCS are defined in relation to resources, such that any one endpoint descriptor is only associated with one particular resource descriptor. Figure 1 describes the relationship between UCCS context, resources, and endpoints.

The interface for UCCS is divided between the core API and its run-time environment (RTE) (Figure 2). The RTE API is an interface providing run-time services such as process startup, out-of-band communication, and a key storage and retrieval system, as well as other run-time services. UCCS does not implement run-time services, but relies on external libraries such as ORTE [13], SLURM [14], STCI [15], and other third party runtime libraries. Such an approach allows it to decouple the core communication API from the run-time environment, in a way that enables easy porting to different run-time libraries.

The core API features consist of initialization, remote memory access, atomic memory operations, active messages, and collectives. Aside from the functions required for creating descriptors for communication contexts, resources, and endpoints, UCCS also provides methods for querying network capabilities and registering, deregistering, and exporting memory segments for use in future calls. Remote Memory Access (RMA) operations consist of functions for one-sided puts and gets optimized for either short, medium, or large message sizes, as well as functions for non-contiguous data. Atomic Memory Operation (AMO) functions include atomic add, fetch-and-add, increment, fetch-and-increment, swap, and

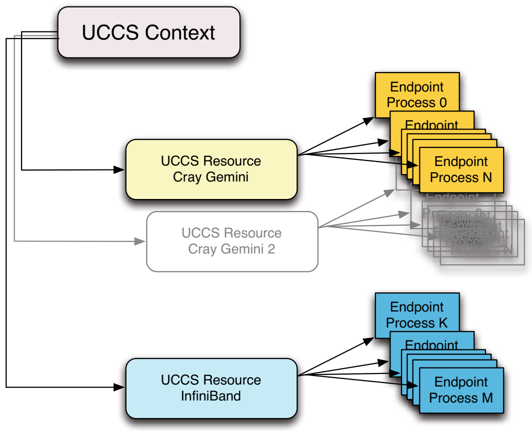


Fig. 1. A relation between the UCCS communication context, resource, and endpoints

conditional swap for both 32 and 64 bit data sizes. The Active Message (AM) interface can be used to support remote execution and two-sided communication operations. This is performed by first registering a callback handler, then sending the data itself in a manner similar to the RMA put operations. Group communication can also be performed using the provided collective operation functions.

All communication calls in UCCS are inherently non-blocking, so their completion must be checked by waiting on a request handle created for tracking an operation's progress. To aid in the management of outstanding operations, UCCS provides functions to test or wait on these handles in whichever way best suits the given situation. The user may test or wait either on a specific handle, all handles in a provided list, or any of the handles in a provided list. These management functions result in a returned status that indicates whether the operation completed successfully, an error occurred, or some other status as appropriate. In addition to the test and wait functions for remote completion, it is also possible to ensure local completion of all outstanding operations by flushing all communication intended for a particular set of endpoints from the local execution context calling the function.

4.3 UCCS and OpenSHMEM Integration

The UCCS and RTE APIs provide an interface that enables simple yet efficient implementation of the OpenSHMEM API (Figure 3). The integration process can be divided into two primary phases: UCCS library initialization and communication semantics implementation.

4.3.1 Initialization

The initialization of the RTE starts up all the PEs in the system, after which the number of participating PEs and each individual caller's index (PE number) can then be queried.

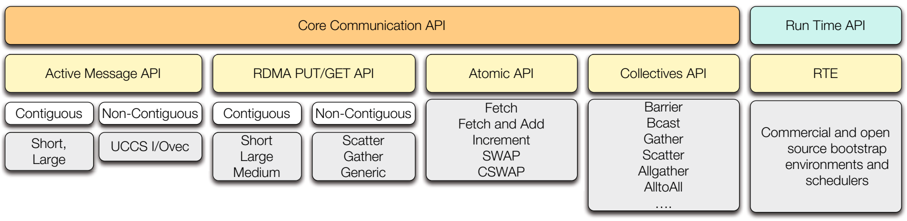


Fig. 2. UCCS API Layout

Once the RTE has been initialized, system data can then be exchanged between PEs without the need for the full communication framework to be online. For a high performance implementation of OpenSHMEM, this can be especially important for two reasons:

1. There is no requirement that the symmetric heaps on the PEs must exist in memory at exactly the same address, so all PEs must be able to broadcast the starting address of their own symmetric heap to all other PEs to allow address translation.
2. Some network technologies such as InfiniBand may require some form of memory registration information before being able to access remote devices, which would also have to be communicated first before any communication operations from client code may be called at all.

To communicate this data, the RTE layer's Storage Retrieval System (SRS) was used, which allows for the easy broadcasting of key-value pairs throughout the system. The starting addresses for the symmetric heaps as well as data on memory segments and associated registration information are individually published to the SRS session, which automatically handles distribution of the data to other PEs subscribed to the session in the system. To keep proper track of this, all information collected about a particular PE is wrapped in a container, such that the full view of the system is an array of these containers indexed by PE id. After the initial bootstrapping is complete, the UCCS context may be created, and resources discovered for it. After creating the descriptors for the resources, those resources are then queried to discover what their network capabilities are. These capabilities include the list of supported operations as well threshold values for the maximum size for small and medium messages supported by the network resource. This information is then stored so as to make the best choices for what operations and message sizes to use in future communication. When all this information is obtained and exchanged, the endpoints may then be set up. During the PE initialization process, each PE will query UCCS endpoints to determine the reachability of every other PE. Finally, a barrier is performed based on the exchanged information, which will establish successful completion of UCCS initialization upon return.

4.3.2 Communication Semantics

OpenSHMEM RMA and AMO operations map directly to RMA and AMO interfaces in UCCS. Since UCCS exposes only non-blocking semantics, OpenSHMEM communication calls are implemented as a UCCS communication operation and then a wait on the operation's associated request handle. Once the handle has been completed, the OpenSHMEM operation is marked for completion as well. In all cases, the destination address must first be modified so that the offset of the address with respect to the calling PE's symmetric heap is the same as the offset for the new address with respect to the destination PE's heap. For puts and gets on arbitrary sizes of data, the size of the message is first checked against the threshold values discovered in intialization for the maximum allowed for short or medium messages. The destination endpoint is then looked up using the address translation table built during initialization, and the put or get for the appropriate size is invoked on it for the requested values. Atomics are similar, but don't have the requirement to check for message size, merely needing to have the appropriate UCCS call invoked and waited for to satisfy OpenSHMEM's completion policy. Since all other communication operations are built off of these, all that is left is to ensure that upon exit all associated memory is freed, and call the RTE's own finalize function.

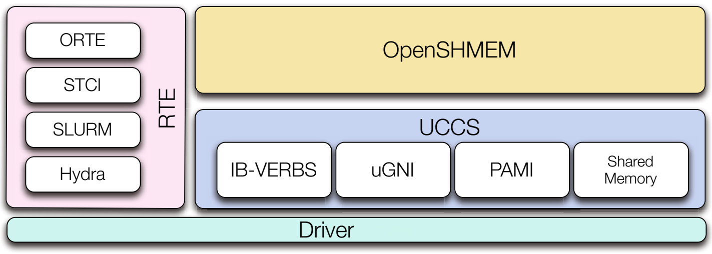


Fig. 3. OpenSHMEM and UCCS software layers

5 Results

The evaluation of this implementation was conducted on an SGI Altix XE1300 system located at the Oak Ridge National Laboratory's Extreme Scale System Center. The system consists of 12 compute nodes, each with two Intel Xeon X5660 CPUs for a total of 12 CPU cores and 24 threads. Compute nodes are interconnected with Mellanox's ConnectX-2 QDR HCA with one port. This particular system was selected due to the availability of SGI MPT version 2.03 that comes with a state-of-the-art SHMEM implementation for InfiniBand interconnects. In addition, we installed the newly updated OpenSHMEM 1.0 implementation, GASNet version 1.20.2, and pre-production UCCS version 0.4. This version of the UCCS library provides high-performance communication services for InfiniBand interconnects only. Evaluation of intra-node communication support and other interconnects is out of the scope of this paper. All tests were run with both the OpenSHMEM implementation's GASNet and UCCS communication layers, as well as SGI's SHMEM and, when appropriate, are compared to results obtained using InfiniBand verbs (IB-verbs) library. We were not able to evalute the Mellanox ScalableSHMEM implementation, since the pre-production version of the library did not run on our platform.

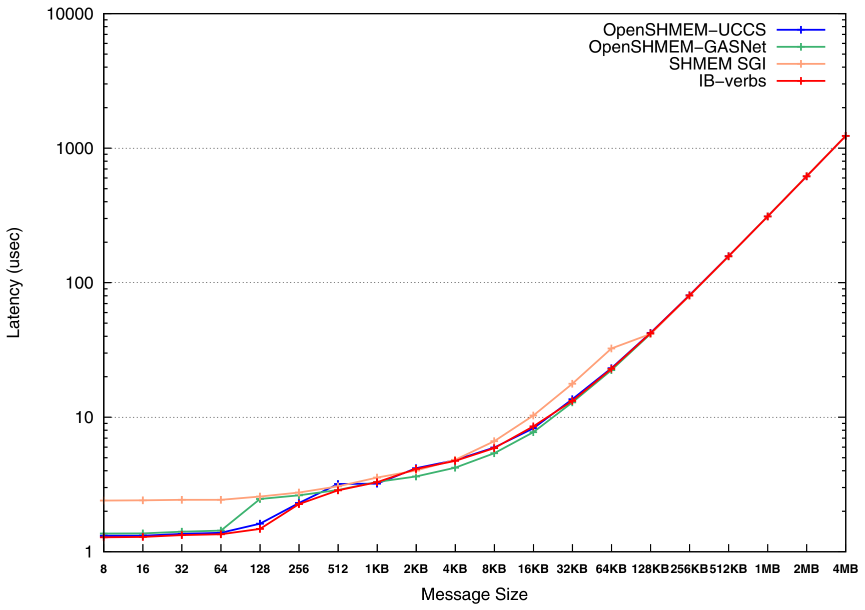


Fig. 4. SHMEM Put Latency

The first test measures the latency for put operations, by using a ping-pong approach. The first PE puts to a location on the second PE, which is simply waiting for the value at that location to fully update. Upon noticing the update, it then puts to a location on the first PE that it likewise is waiting on, upon receipt of which the ping-pong operation is complete. The time for the complete exchange is halved to determine the latency for a single put operation, in order to achieve a result similar to what the IB-verbs Perftest benchmark produces. This test found median latencies for increasingly large message sizes ranging from 8 bytes to 4 megabytes (based on powers of two). The results of this test are compared to that of IB-verbs, as seen in Figure 4. These results show performance close to IB-verbs for message sizes larger than 512 bytes, with *OpenSHMEM-UCCS* performing the closest to IB-verbs, the largest difference being only 3%. For small messages, *OpenSHMEM-UCCS* had the best performance, ranging from 2-11% slower than IB-verbs. In contrast, *OpenSHMEM-GASNet* was 1-67% slower, and SGI's implementation completed in 7-88% more time compared to IB-verbs.

The second test measures the latency for messages of varying sizes using gets. The median latency for a get of a particular size is recorded for all message sizes based on a power of two, starting from eight bytes and continually doubling up to four megabytes. The results are compared to those obtained using IB-verbs in Figure 5. It can be seen that the performance seen with IB-verbs can be closely matched in all implementations for all message sizes, with the UCCS version consistently performing the closest to IB-verbs with negligible overhead at its best and being 4% slower at its worst. The GASNet communication layer performed similarly, though latency starts to drag noticeably behind IB-verbs for increasingly large message sizes, resulting in higher overheads of 2-12%. SGI SHMEM performed similar to GASNet runs for larger message sizes, but experienced more overhead for smaller sizes resulting in a 6-22% overhead.

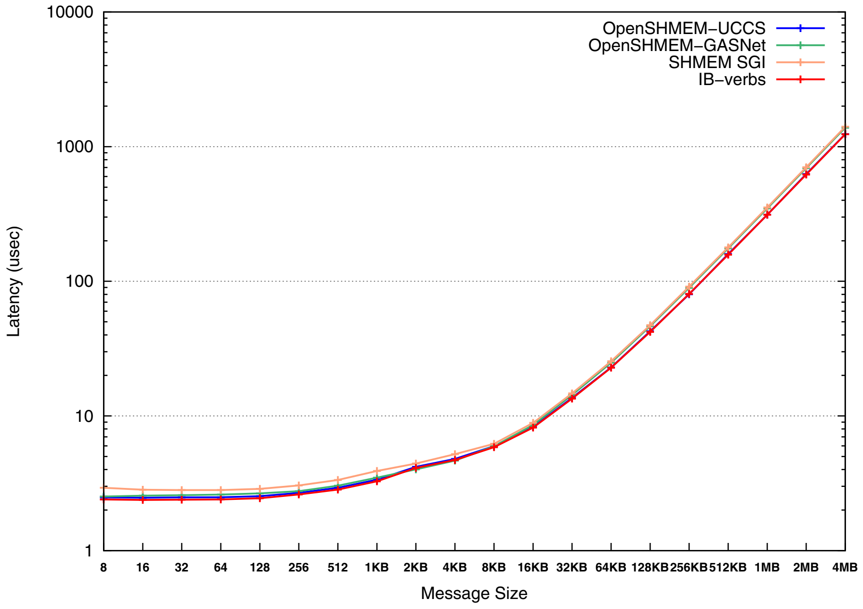
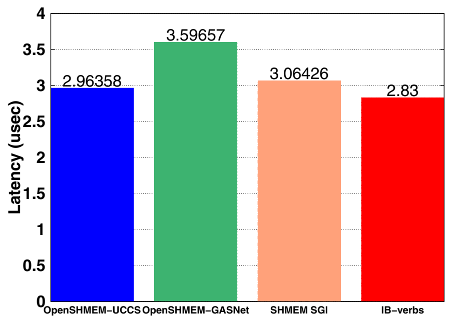
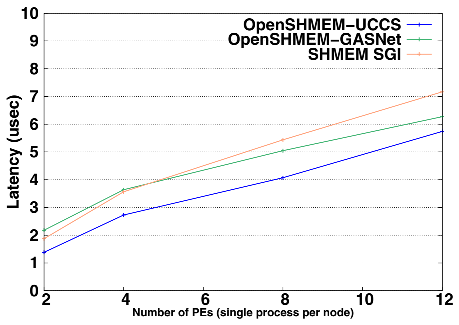


Fig. 5. SHMEM Get Latency

The third test measures the latency for atomics by using long long fetch-and-add operations. This test was the most straightforward one, simply finding the median time elapsed to perform one such operation and comparing the results to IB-verbs. The fourth test measures the time it takes to perform a barrier operation on all PEs. This test was performed using two, four, eight, and twelve PEs, with the median time recorded for only *OpenSHMEM-GASNet*, *OpenSHMEM-UCCS*, and SGI, as there is no equivalent test for IB-verbs. For the runs done using both of the OpenSHMEM implementations, a recursive doubling algorithm was used for the barrier itself. The results for the fetch-and-add tests are shown in Figure 6(a) and the barrier results are in Figure 6(b). When executing atomics, the UCCS communication layer consistently performed better than SGI's implementation, which in turn performed better than the GASNet layer. *OpenSHMEM-UCCS* took 5% more time to execute compared to IB-verbs, while



(a)



(b)

Fig. 6. SHMEM Long Long Fetch-and-Add (a) and Barrier All (b)

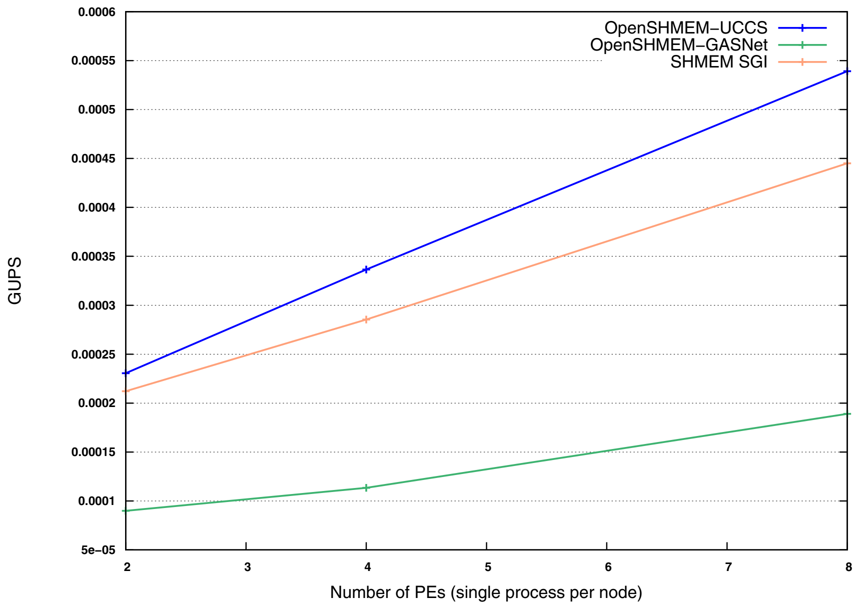


Fig. 7. GUPS

SGI took 8% more and *OpenSHMEM-GASNet* took 27% more time. On barriers, *OpenSHMEM-UCCS* again performed the best, with *OpenSHMEM-GASNet* performing 9-58% slower, and SGI performing 25-36% slower.

The final test is based on the RandomAccess benchmark from the High Performance Computing Challenge (HPCC) [16], used to measure the performance of the memory architecture of a system in terms of Giga UPdates per Second (GUPS). This is determined by the number of random memory locations that can be updated in one second, which can be used to give an idea of peak performance of the system with respect to random memory access. For each random location, a value is retrieved through a get, a new value stored with a put, and another value incremented with an atomic operation. This test was run for two, four, and eight PEs, where the amount of work done for each run was multiplied by the number of participating PEs. The GUPS achieved from these tests can be seen in Figure 7. The UCCS communication layer achieved the highest performance in all runs, with SGI achieving 82-92% of the GUPS when compared agaisnt the UCCS layer. The GASNet layer, however, performed almost three times slower than the UCCS layer, reaching between 35% and 39% of the GUPS that the UCCS layer achieved. This is likely due to the extra overhead GASNet incurs on its communication, particularly atomic operations, by relying on active messages for successful execution of its calls. SGI's SHMEM, on the other hand, likely saw its relative performance difference due to the greater latency for small put operations.

6 Conclusion and Future Work

This paper presented *OpenSHMEM-UCCS*, an OpenSHMEM reference implementation whose communication is based on UCCS, a low-level communication middleware. An evaluation with microbenchmarks and the RandomAccess benchmark for GUPS showed that it outperformed the current reference implementation (*OpenSHMEM-GASNet*) and the state-of-the-art SGI SHMEM. Particularly, the *OpenSHMEM-UCCS* RMA and atomic operations outperformed *OpenSHMEM-GASNet* and SGI's implementation. For example, for the widely used put operation, *OpenSHMEM-UCCS* outperformed the current reference implementation by as much as 53%, and SGI's implementation by as much as 83%. When running the RandomAccess benchmark for measuring GUPS, using *OpenSHMEM-GASNet* resulted in only 35-39% the GUPS achieved by *OpenSHMEM-UCCS*, while SGI's implementation achieved 82-92% of the performance. These results were able to be achieved due to a focus on minimizing software overheads in the communication path while focusing on the needs and capabilities of the underlying network hardware. The previous implementation with GASNet relied heavily on active messages for atomic and some remote memory operations, whereas the UCCS communication layer provides semantics that directly map to low-level atomic and RDMA primitives of network hardware. This allows for a much tighter and streamlined flow that can achieve results much closer to what the network hardware supports.

Moving forward with UCCS and its integration with OpenSHMEM, we plan to extend the UCCS library to support intra-node communication as well as additional transport layers such as such Cray uGNI and IBM PAMI. Moreover, we plan to extend the UCCS InfiniBand transport layer to support InfiniBand Dynamically Connected Transport, extended AMOs, and on demand memory registration.

Acknowledgments. This work is supported by the United States Department of Defense and used resources of the Extreme Scale Systems Center located at the Oak Ridge National Laboratory.

References

1. 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 2010, New York, NY, USA (2010)
2. Poole, S.W., Hernandez, O., Kuehn, J.A., Shipman, G.M., Curtis, A., Feind, K.: OpenSHMEM - Toward a Unified RMA Model. In: Encyclopedia of Parallel Computing, pp. 1379–1391 (2011)
3. Pophale, S.S.: SRC: OpenSHMEM library development. In: Lowenthal, D.K., de Supinski, B.R., McKee, S.A. (eds.) ICS, p. 374. ACM (2011)
4. Bonachea, D.: GASNet Specification, v1.1. Technical report, Berkeley, CA, USA (2002)
5. Shamis, P., Venkata, M.G., Kuehn, J.A., Poole, S.W., Graham, R.L.: Universal Common Communication Substrate (UCCS) Specification. Version 0.1. Tech Report ORNL/TM-2012/339, Oak Ridge National Laboratory, ORNL (2012)
6. Graham, R.L., Shamis, P., Kuehn, J.A., Poole, S.W.: Communication Middleware Overview. Tech Report ORNL/TM-2012/120, Oak Ridge National Laboratory, ORNL (2012)
7. Yoon, C., Aggarwal, V., Hajare, V., George, A.D., Billingsley III, M. GSHMEM: A Portable Library for Lightweight, Shared-Memory, Parallel Programming. In: Partitioned Global Address Space, Galveston, Texas (2011)
8. Jose, J., Kandalla, K., Luo, M., Panda, D.K.: Supporting Hybrid MPI and OpenSHMEM over InfiniBand: Design and Performance Evaluation. In: Proceedings of the 2012 41st International Conference on Parallel Processing, ICPP 2012, pp. 219–228. IEEE Computer Society, Washington, DC (2012)
9. Brightwell, R., Hudson, T., Pedretti, K., Riesen, R., Underwood, K.D.: (Portals 3.3 on the Sandia/Cray Red Storm System)
10. Mellanox Technologies LTD.: Mellanox ScalableSHMEM: Support the OpenSHMEM Parallel Programming Language over InfiniBand (2012), `http://www.mellanox.com/related-docs/prod_software/PB_ScalableSHMEM.pdf`
11. Mellanox Technologies LTD.: Mellanox Messaging (MXM): Message Accelerations over InfiniBand for MPI and PGAS libraries (2012), `http://www.mellanox.com/related-docs/prod_software/PB_MXM.pdf`
12. Ho Lam, B.C., George, A.D., Lam, H.: TSHMEM: Shared-Memory Parallel Computing on Tilera Many-Core Processors. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, pp. 325–334 (2013), `http://www.odysci.com/article/1010113019802138`
13. Castain, R.H., Woodall, T.S., Daniel, D.J., Squyres, J.M., Barrett, B., Fagg, G.E.: The Open Run-Time Environment (OpenRTE): A Transparent Multi-Cluster Environment for High-Performance Computing. In: Di Martino, B., Kranzlmüller, D., Dongarra, J. (eds.) EuroPVM/MPI 2005. LNCS, vol. 3666, pp. 225–232. Springer, Heidelberg (2005)
14. Yoo, A.B., Jette, M.A., Grondona, M.: SLURM: Simple linux utility for resource management. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2003. LNCS, vol. 2862, pp. 44–60. Springer, Heidelberg (2003)
15. Buntinas, D., Bosilica, G., Graham, R.L., Vallée, G., Watson, G.R.: A Scalable Tools Communication Infrastructure. In: Proceedings of the 22nd International High Performance Computing Symposium, HPCS 2008 (2008)
16. HPCC: RandomAccess Bechmark (2013), `http://icl.cs.utk.edu/hpcc/index.html`