Design and Implementation of OpenSHMEM Using OFI on the Aries Interconnect

Kayla Seager[1(✉)], Sung-Eun Choi[2], James Dinan[1], Howard Pritchard[3], and Sayantan Sur[1]

[1] Intel Corporation, Santa Clara, USA
kayla.seager@intel.com
[2] Cray Inc., Seattle, USA
[3] Los Alamos National Laboratory, New Mexico, USA

Abstract. Sandia OpenSHMEM (SOS) is an implementation of the OpenSHMEM specification that has been designed to provide portability, scalability, and performance on high-speed RDMA fabrics. Libfabric is the implementation of the newly proposed Open Fabrics Interfaces (OFI) that was designed to provide a tight semantic match between HPC programming models and various underlying fabric services.

In this paper, we present the design and evaluation of the SOS OFI transport on Aries, a contemporary, high-performance RDMA interconnect. The implementation of Libfabric on Aries uses uGNI as the lowest-level software interface to the interconnect. uGNI is a generic interface that can support both message passing and one-sided programming models. We compare the performance of our work with that of the Cray SHMEM library and demonstrate that our implementation provides performance and scalability comparable to that of a highly tuned, production SHMEM library. Additionally, the Libfabric message injection feature enabled SOS to achieve a performance improvement over Cray SHMEM for small messages in bandwidth and random access benchmarks.

1 Introduction

Current trends in high performance computing (HPC) system architecture pose new challenges and introduce new requirements for the system fabric. Dramatic increases in the number of cores and threads per node requires a host-fabric interface (HFI) that can process communication on behalf of many threads efficiently. At the same time, these *throughput-oriented* cores present new challenges to communication processing on the host processor [4]. Further, increases in the overall system scale in combination with a flattening trend in the amount of memory available per thread places additional stress on scalability requirements. In response to these challenges, a variety of novel solutions [11,13] and interfaces are being explored [5,9,14,27].

In addition to new techniques at the system fabric layer, the communication middleware and underlying communication software stack must also be

M. Gorentla Venkata et al. (Eds.): OpenSHMEM 2016, LNCS 10007, pp. 97–113, 2016.
DOI: 10.1007/978-3-319-50995-2_7

adapted to leverage and expose new functionality. The OpenFabrics Alliance recently introduced the OpenFabrics Interfaces (OFI) framework as a new, open-source software ecosystem designed to enable efficient usage of evolving high-performance fabrics [14]. OFI's libfabric component provides a communication interface that is designed for scalability, flexibility, and extensibility. In particular, typical scalability challenges, such as endpoint addressing, connection management, message processing, and memory registration are encapsulated within libfabric, allowing them to be optimized using fabric- and system-specific capabilities.

The OpenSHMEM specification is a recent initiative directed toward standardizing and extending the SHMEM* parallel programming model for future systems. OpenSHMEM defines a partitioned global address space (PGAS) data access library that can be used to establish one-sided access to read, write, and atomically update remote data. OpenSHMEM applications commonly require high throughput and the ability to perform remote data accesses asynchronously, thereby placing significant demands on the underlying system fabric.

In this work, we document our experiences with the development of an OpenSHMEM software stack using OFI on a contemporary HPC interconnect. We present an open source implementation of the OpenSHMEM 1.3 specification that targets the OFI libfabric interface and describe how libfabric can be used to improve the efficiency of OpenSHMEM middleware. We further describe the implementation of libfabric for the Cray® XC40™ system with the Aries interconnect that utilizes the uGNI [8] API. We evaluate the performance of our software stack using several communication and application benchmarks. The results indicate that the performance of the open-source SHMEM and libfabric is comparable to the highly tuned, production Cray SHMEM library. In addition, we show that the libfabric message injection feature enabled a performance improvement over Cray SHMEM for small messages in bandwidth and random access benchmarks.

Our paper starts with a description of the relevant background information and related work in Sect. 2. Next, we describe the design of our OpenSHMEM implementation and underlying OFI implementation for the Aries interconnect in Sects. 3 and 4, respectively. We present an experimental evaluation in Sect. 5 and conclude with Sect. 6.

2 Background and Related Work

Our work describes and analyzes the implementation of the OpenSHMEM specification using a modern fabric interface. In this section, we provide an overview of these topics and some of the most closely related works.

2.1 Fabric Interfaces

A variety of low-level communication APIs have been used in HPC for high performance networking, including the OpenFabrics Alliance (OFA) Verbs API,

PAMI [19], Portals [2], and uGNI [8]. Often, such low-level APIs are customized to leverage specific system architectures. Recently, the industry has trended toward exchanging system-specific APIs for open, portable *fabric interfaces* that provide a low-level interface to fabric services while minimizing ties to specific architectures. This approach promises to provide better portability for communication middleware, such as OpenSHMEM, while maximizing the exposure of application-level communication semantics to the fabric to enable aggressive optimization.

OpenFabrics Interfaces. The OpenFabrics Alliance (OFA) provides open-source software for high-performance networking applications that demand low latency and high bandwidth. Historically, the only fabric interface offered by the OFA was the Verbs API as defined in the InfiniBand* specification. As the InfiniBand specification was originally envisioned as a generic system I/O interconnect, there are semantic differences between Verbs and the requirements of PGAS libraries and languages. These semantic mismatches require unnecessary adaptations in PGAS implementations, such as OpenSHMEM, resulting in significant software overhead [21].

The OFA has created a working group, called the OpenFabrics Interfaces Working Group (OFIWG), that aims to define a fabric interface that has a tight semantic map to various applications classes that use it, including PGAS programming models. Members of the PGAS community provided input into the design of the new fabric interfaces to help improve the mapping of PGAS features onto fabric interface features. The fabric library created from this effort is called *libfabric*. It consists of two logically distinct components: A set of fabric *providers* that implement the communication interfaces for a particular fabric hardware, and a general purpose *framework* that provides a plugin-like capability for providers. In the rest of the paper, we use the term *uGNI provider* to imply the specific implementation of libfabric interfaces for the Aries interconnect. Libfabric is freely available from Github [20], and is distributed via the OpenFabrics Enterprise Distribution (OFED) as well as popular Linux distributions.

Other Fabric APIs. The Portals interface [2] allows the user to describe actions that are performed on remote memory segments – possibly gated by message matching requirements – providing close alignment between HPC communication libraries and the underlying software or hardware implementation of the Portals layer. A variety of PGAS runtimes have been ported to use Portals, including OpenSHMEM [3]. The current Portals 4 specification [2] adds a lightweight non-matching interface to boost PGAS messaging rates. Additionally, it introduces logical rank-based addressing to simplify code paths, eliminate cache misses, and improve memory scaling. Members of the Portals community also participate in the effort to craft the OFI interface, resulting in adoption of multiple concepts from the Portals API.

OpenUCX [27], is another fabric framework that is being developed by as a collaboration outside the OpenFabrics umbrella. It aims to provide semantics

that target data centric and HPC programming models. UCCS [26] is a predecessor to OpenUCX and a detailed study of OpenSHMEM performance on UCCS was recently conducted [28]. Additionally a study of UCCS over uGNI on Gemini was evaluated in [16]. The authors of these studies observed similar performance results as we present; however, because of differences in the hardware and software environments, our results cannot be directly compared.

2.2 OpenSHMEM

OpenSHMEM [23] is a parallel programming model that defines a Single-Program, Multiple-Data (SPMD) execution model and an accompanying partitioned global address space (PGAS) communication library. OpenSHMEM allows the programmer to expose regions of memory for remote access using one-sided read, write, and atomic access routines.

Recently, the OpenSHMEM specification was introduced in an effort to standardize and extend the SHMEM⋆ communication library. SHMEM has been in use for over two decades, with implementations from most major HPC vendors, however the lack of an open specification has resulted in variations across implementations and has limited the ability of the user community to extend the programming model.

A reference implementation of the OpenSHMEM specification is available as open source [22] and is compatible with a wide range of system fabrics through the low-level GASNet API [6]. The OpenMPI communication middleware also recently added support for OpenSHMEM [12], called OSHMEM. OSHMEM leverages the MPI runtime and MPI collective implementations to provide a lightweight implementation. Mellanox⋆ Scalable SHMEM is a proprietary implementation that is available as a part of the HPC-X toolkit distributed by Mellanox. It is designed to work on Mellanox InfiniBand fabrics. Similarly, the MVAPICH2-X [17,18] SHMEM distribution is a closed source implementation that targets only Mellanox fabrics.

In this work, we utilize the open source Sandia OpenSHMEM (SOS) library [25], which is based on the earlier Portals SHMEM library [3]. SOS extends Portals SHMEM with support for the new OpenSHMEM 1.3 specification, as well as adding support for the OpenFabrics Interface libfabric communication layer [21]. While existing libfabric support was recently added, supplementary work has refined the mapping of OpenSHMEM to OFI, yielding additional portability and performance benefits. Furthermore, the codebase continues to evolve alongside the OpenSHMEM community as a sandbox and proof-of-concept for the latest OpenSHMEM proposals.

3 Design of OpenSHMEM for OFI

As shown in Fig. 1, Sandia OpenSHMEM (SOS) defines internal network data transport and shared memory layers. The SOS transport layer was designed to reduce the number of functions that must be implemented for each fabric, while

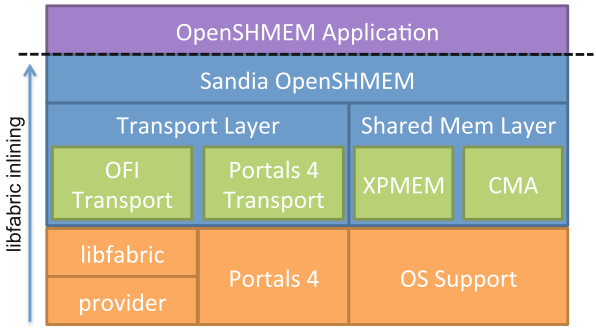


Fig. 1. Sandia OpenSHMEM library design, showing libfabric inlining to eliminate software overheads.

exposing core OpenSHMEM communication semantics so that the transport can optimize for scalability and performance. SOS provides support for both Portals 4 and OFI. This work focuses primarily on the OFI transport layer and extending it to support a broad range of libfabric capabilities and efficiently utilize the Aries system interconnect through the libfabric uGNI provider.

The SOS-OFI transport layer requires provider support for remote memory access (RMA) and remote atomics capabilities. The libfabric RMA and atomic APIs were designed to provide a direct mapping of performance sensitive PGAS operations to libfabric routines, with the intention of facilitating close alignment with fabrics that provide support for remote direct memory access (RDMA) and atomic capabilities. Libfabric further supports a direct build, shown in Fig. 1 where the implementation of the libfabric API routines are inlined into the middleware, enabling cross-call compiler optimizations and eliminating function call overheads. Sandia OpenSHMEM (SOS) also supports aggressive inlining within the implementation, which is used to reduce the middleware stack overheads to a single function call. In combination, these optimizations have been shown to significantly improve software overheads, and by extension small message latency and throughput [21].

3.1 Launch, Wire-Up, and Memory Registration

Careful setup and resource management is crucial for achieving scalability and reducing overheads. SOS supports the PMI-1 and PMI-2 process management interfaces (PMIs) [1], and we have added support for the Cray process manager. For stand-alone builds, SOS includes a built-in PMI-1 option that can be used to attach to any PMI-1 compliant job launcher.

Mapping of OpenSHMEM PE numbers to network addresses is typically facilitated through a scalable libfabric address vector (AV); however, for portability reasons, SOS supports both the *map* and *table* AV modes. The map mode provides the broadest compatibility; however, it requires the middleware to maintain a table that maps PE numbers to fabric interface (FI) addresses obtained

Table 1. Memory registration and remote addressing models supported by the SOS OFI transport and resulting overheads.

	Remote virtual addressing	Remote offset addressing
Scalable MR		Offset calculation
Basic MR	Key tables	Key tables, base address table, offset calculation

through the PMI exchange. When a communication operation is performed, the target PE number must be first converted to the corresponding FI address using this table. The AV table mode provides better scalability and performance opportunities by allowing the PE number to be used directly in communication operations and performing address resolution within the provider. For networks that require a translation table, the provider is able to map the table in a shared segment, improving the memory scalability. Further, the AV table mode provides a mechanism to take advantage of networks that offload or regularize address resolution. We use AV table in our study since it is supported by OFI-uGNI.

In libfabric, remote memory access (RMA) operations require both a protection key and a destination address. Libfabric provides two different models for exposing memory regions for remote access, referred to as *scalable* and *basic* memory registration (MR), that establish different key and destination address semantics. We have implemented support for both modes in SOS, and we further take advantage of systems that can support mapping the symmetric heap and data segments at the same base addresses across all PEs, referred to as remote virtual addressing. The combination of these two features results in the matrix shown in Table 1. In the scalable MR mode with remote virtual addressing, SOS exposes the full address space of the PE for efficient remote access. In all other modes, the heap and data segments are exposed separately.

Basic memory registration is the most portable model and allows the provider to determine the memory protection key, and requires the application to provide destination virtual address for the RDMA operation. This results in SOS exchanging protection keys and maintaining a key table as the key may be different on different PEs. Protection keys are required by some networks to enable access to remote memory. Additionally, basic memory registration support requires the SOS middleware to maintain tables containing the symmetric heap and data segment base addresses of all PEs. When performing an RMA operation, a local offset calculation is performed to convert the symmetric address passed to the OpenSHMEM routine into an offset relative to the symmetric heap or data segment base. This is then added to the target PE's base address before performing the libfabric communication operation.

In contrast, scalable memory registration allows the user to select the protection key, eliminating the key table overheads. Addressing in the scalable memory registration model is performed relative to the beginning of the memory segment exposed at the target PE. In the remote virtual addressing model, the full address space is exposed and the symmetric address passed to the OpenSHMEM routine

can be directly used by libfabric. When remote virtual addressing is not available, the symmetric address is converted into an offset relative to the local base address and this offset is passed directly to the libfabric communication routine. Thus, scalable memory registration eliminates both the key and base address tables.

In this paper, we use the Basic memory registration path due to current uGNI provider limitations. In the future, we may explore adding the scalable memory registration feature to the provider in order to expose more optimal code paths in OpenSHMEM.

3.2 One-Sided Communication Operations

OpenSHMEM one-sided put operations are mapped to the libfabric write API and different strategies are used depending on the message size. OpenSHMEM defines both blocking and nonblocking put operations; blocking operations return after local completion, whereas nonblocking operations provide no completion guarantee. For messages below the injection threshold of the fabric, the fi_inject_write routine is called; in all other cases, the fi_write routine is called. The inject-write routine provides immediate local completion and the provider is responsible for any buffering needed to ensure reliable message delivery. For blocking put operations whose message size is greater than the injection threshold and less than the user-selectable SMA_BOUNCE_SIZE parameter, the user's data is copied to a temporary bounce buffer and the operation provides immediate local completion. As shown in Sect. 5, we have observed that bounce buffering can provide significant performance improvements for applications that rely on blocking put operations; however, this optimization can be disabled when not needed to reduce the memory footprint of SOS. Finally, larger messages are issued directly using the fi_write operation and are fragmented according to the maximum transmission unit (MTU) of the fabric.

The OpenSHMEM atomic operations are divided into three categories, non-fetching, fetching, and comparison atomics. Currently, all OpenSHMEM atomic operations are blocking. The non-fetching atomics perform a remote update without returning a result and are implemented using the fi_inject_atomic and fi_atomic routines using the same strategy as described for blocking put operations. While all OpenSHMEM atomic routines are scalar and map to inject-atomics, SOS does implement vector atomics in the transport layer that is only utilized by the OpenSHMEM collectives API. The fetching atomic and comparison atomic operations are implemented using fi_fetch_atomic and fi_compare_atomic operations. However, since these blocking operations return the prior contents of the destination buffer, they cannot return until the operation has completed and neither message injection nor bounce buffering is used.

Finally, the OpenSHMEM get operations are implemented directly using the libfabric fi_read routine. The runtime must wait for blocking get operations to complete before returning. In the nonblocking case, the routine returns immediately and the application completes the get operation with a subsequent call to the shmem_quiet routine.

As shown in Fig. 1, SOS supports shared memory through XPMEM and Linux cross-memory attach (CMA). When enabled, shared memory is used to improve the performance of put and get operations. Atomic operations are always performed through the transport layer in order to ensure atomicity.

3.3 Ordering and Remote Completion Operations

All communication operations in libfabric are nonblocking; completion of issued operations is established using either event counters or completion queues. When they are created, the programmer selects which events will be captured by a particular event counter or completion queue. Event counters and completion queues are then bound to a fabric endpoint. Thus, for a given operation, the type of completion that will be generated is determined by the fabric endpoint on which the operation was issued and the type of event that the operation generates. In SOS we mainly use counters for completion, but a queue is used for bounce buffering and error handling.

Completion queues provide a full event structure for each completed operation, with detailed information including a "context" value that was supplied when the original operation was performed. The context is typically used to forward a reference to a middleware object (e.g. a request object) from the communication operation to the full event. In SOS, full events are used only when a put or non-fetching atomic operation utilizes a temporary bounce buffer. In this case, a pointer to the bounce buffer is included as the context and is used when processing the remote completion event to return the bounce buffer to a free pool. The number of operations issued using a bounce buffer is tracked by a variable within the SOS runtime and is used to wait for pending operations to complete when performing an OpenSHMEM fence or quiet operation.

Full completion events incur an overhead to allocate space in the event queue and populate the event with the information from the operation that completed. In contrast, event counters capture no information regarding specific operations that have completed. Instead, the counter is simply incremented upon completion of the operation, resulting in lower overhead than a full event. SOS establishes two counters for tracking completion of read and write operations separately. Operations that do not return a result, including puts and non-fetching atomics are accounted for using the write counter (with the exception of operations using a bounce buffer; those are tracked separately using a completion queue). All other operations are accounted for using the read counter. Within the SOS middleware, two variables are used as counters to track the number of operations of each kind that have been issued.

Separate read and write counters are used to optimize blocking communication operations. Blocking put and non-fetching atomic operations that are buffered using either the inject or bounce buffer method return immediately. Large blocking put operations must wait for completion before returning. Similarly, blocking fetching operations of any size must wait for completion prior to returning. By using separate counters for these classes of operations, we allow operations to overtake each other. This can provide significant benefit in cases where small

fetching operations are combined with large puts. It is possible to use additional counters to optimize blocking operations based on the operation type and size (e.g. to separate fetching and comparison atomics from gets). We plan to investigate the impact of such refinements during future performance tuning.

The OpenSHMEM quiet operation must wait for remote completion of all pending blocking and nonblocking operations, whereas the fence operations must only ensure ordering of remote updates. Currently, SOS waits for completion of all pending communication in both quiet and fence operations. In the future, we plan to leverage the separation of read and write counters to optimize these operations. In this model, the quiet operation waits for completion of both remote writes and reads, whereas the fence operation waits only for completion of remote writes.

3.4 Notification API

The OpenSHMEM wait API allows the programmer to wait for an update to a location in symmetric memory. When shared memory optimizations are not used, all updates arrive through the network and the wait implementation can block on a network event rather than polling the target memory location. When supported by the OFI provider, the SOS OFI transport binds an event counter to one or more regions of exposed memory that is incremented whenever a remote update occurs. In this mode, the implementation of the OpenSHMEM wait operation blocks on a communication event, allowing the provider to optimize resource utilization.

4 Libfabric for the Aries Network

The libfabric provider, shown as the bottom-most layer in Fig. 1, is responsible for mapping the libfabric APIs to the underlying system. In this section we give an overview of the implementation of the libfabric API utilized by SOS. We refer readers to previous work for further details on the implementation [7,24].

The provider implementation for the Aries interconnect utilizes the Generic Network Interface (uGNI) library [8], a low-level interface that exposes the capabilities of the Aries NIC. The uGNI provider utilizes the Aries NIC's fast memory access (FMA) hardware for small messages, as well as the bulk transfer engine (BTE) for offloading large message transfers. FMA descriptors are used to initiate remote loads, stores and atomic operations. FMA descriptors are bound to local Aries hardware-provided completion queues (hCQ) to enable notifications for the completion of remote memory access.

4.1 Addressing and Memory Registration

The uGNI provider supports both the OFI map and table address vector (AV) modes. For both modes, the address entry is represented by the uGNI device address and an identifier for utilizing the hardware protection, in combination

with information about the endpoint and RDMA credentials. AV map mode uses a hash table to store address entries, whereas AV table mode uses a growable vector of address entries.

The uGNI provider supports the OFI basic memory registration (MR) mode, including a configurable memory registration cache. Memory regions are registered with uGNI via a call to uGNI_MemRegister, which returns a handle that is encoded in the key for the memory region. The memory registrations are stored in a red-black tree for fast access in cases where an existing registration satisfies the requested memory region. To further reduce the number of registrations with uGNI, all registrations are rounded up to the nearest page size. Additionally, adjacent memory regions are coalesced into a single, larger entry to further reduce the number of registrations. The memory registration cache also supports lazy deregistration when a memory region is closed. Lazy deregistration holds on to the uGNI memory handle until a configurable limit is reached, after which memory regions are deregistered via a call to uGNI_MemDeregister.

4.2 Issuing and Completing Communication Operations

The OFI RMA operations (fi_write and fi_read) with data size less than 8 KB in size are sent using Aries FMA functionality as a control message payload. Larger transfers are handled using the Aries BTE. The switch-over point can be adjusted using a GNI provider specific fi_open_ops method on a fi_domain object.

The Aries FMA hardware is also used to provide fast atomic operations. Currently, the uGNI provider only supports libfabric atomic operations that are implemented directly by the Aries hardware. This includes 32- and 64-bit versions of *min*, *max*, *sum*, bitwise *OR*, bitwise *AND*, bitwise *XOR*, read, write, *compare-and-swap* and masked *compare-and-swap*. In addition, the uGNI provider exposes the Aries *AND-and-XOR* atomic operation.

The uGNI provider checks for Aries completion events from all active hCQs upon most calls into the libfabric library as well as from an independent progress thread, if automatic progress is requested. Callback functions are used to generate a corresponding libfabric completion event, which is placed on the appropriate completion queue (represented by a singly-linked, double-ended list). The Aries hardware does not directly support completion counters. Completion counters are implemented similarly to completion queues; the callback simply increments the appropriate counter value.

5 Evaluation

We compare the performance of SOS using the OFI transport and uGNI provider with the performance of Cray's SHMEM implementation for the Aries network. Experiments were conducted on the NERSC "Cori" system, which is a Cray® XC40™ with 1,630 compute nodes. Compute nodes are comprised of two Intel® Xeon™ "Haswell" processors (E5-2698 v3) with 32 cores total (16 cores/socket) with hyperthreading disabled, and with 128 GB of memory per node.

The system was running Cray* Linux Environment (CLE) version 5.2up04 and Slurm* version 15.0.8.11. Libfabric (master@3dddae68) was used for the experiments. Libfabric was built using gcc version 5.2.0 with optimization level -O2. No special configuration options were used. Sandia OpenSHMEM (master@a3662791) was configured to use the uGNI provider, but otherwise no special optimizations were used. Cray MPT* 7.3.1 was used for the Cray SHMEM results.

We note that Cray SHMEM is built on top of DMAPP [8], rather than uGNI. DMAPP* is a communication API optimized to support the small (e.g. 8-byte) transfers typical of high-performance PGAS compilers. As a consequence of this, DMAPP relies on a different hardware mechanism in the Aries NIC for managing PCI-e downstream posted write credits (a deadlock avoidance mechanism (DLA)) than uGNI. In contrast, uGNI is optimized for larger transfers more typical of message passing applications including MPI and Lustre's LNET, as well for allowing efficient sharing of DLA resources and FMA descriptors among processes. Note the DLA mechanism was not present in older Cray® XE™ systems, thus making comparison of results presented here with apparently similar results from Cray® XE™ not particularly meaningful.

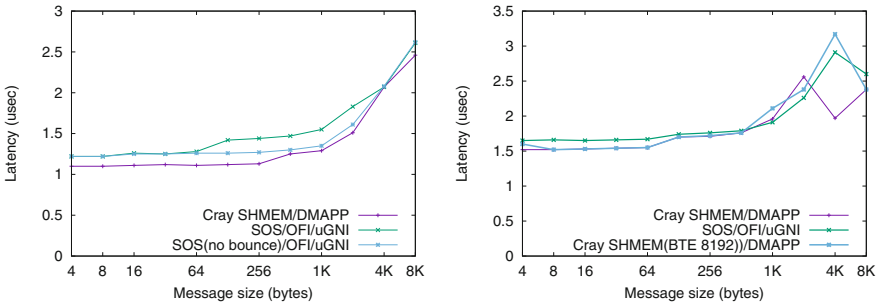
We conduct our evaluation using the SOS communication microbenchmark suite that is included in the SOS distribution, the scalable integer sort (ISX) benchmark [15], and the HPCC random access benchmark [10]. For communication microbenchmarks requiring just two nodes, measurements were taken using nodes connected to the same Aries router.

5.1 Latency Results Using SOS Microbenchmarks

The SOS put latency microbenchmark uses two processes, where one of the processes performs a loop of shmem_putmem() and shmem_quiet() operations. Figure 2a shows results for this test. The figure compares the PUT latency of Cray SHMEM with SOS with and without bounce buffering. Excluding the effects of buffering, the latency of SOS is about 150 nsecs more than that attained using Cray SHMEM. Trace data of the 8-byte put latency runs, in addition to comparison of comparable tests written directly to DMAPP and uGNI, indicate the major contributions to extra overhead for SOS can be attributed to the additional overhead within the uGNI library required to manage DLA credits and support sharing of hardware resources (FMA descriptors) between different processes.

The SOS buffering between 128 and 2048 bytes results in significantly higher overhead for SOS put operations compared to Cray SHMEM. As will be shown below for the streaming benchmark, the bounce buffers can sometimes lead to improved results for SOS.

The SOS get latency also uses two processes, where one of the processes performs a loop of shmem_getmem() operations. Figure 2b compares the results obtained using Cray SHMEM and SOS. Results for Cray SHMEM using an Aries BTE threshold at 8192 bytes are also shown. For small get operations between 4 and 64 bytes, SOS again shows an additional 150 nsecs compared to



(a) Put latency (b) Get latency

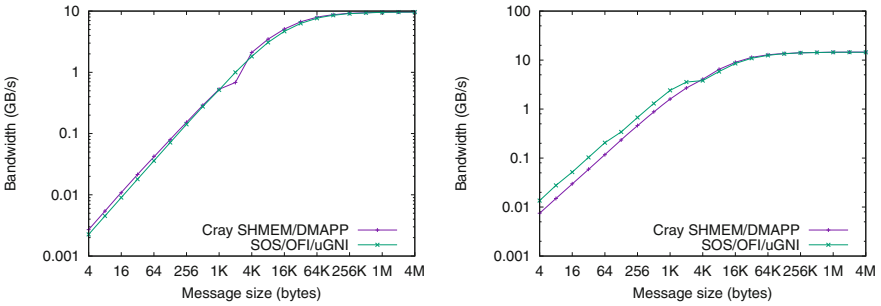
Fig. 2. Latency measurements for Sandia OpenSHMEM and Cray SHMEM.

Cray SHMEM. This overhead gradually decreases until the point where the respective implementations switch to using the Aries BTE - 4096 bytes for Cray SHMEM and 8192 bytes for SOS. Overall, SOS shows comparable performance with Cray SHMEM in terms of latency, with small overheads attributed to differences between DMAPP and uGNI on Aries.

5.2 Bandwidth Results Using SOS Microbenchmarks

The SOS bi-directional write bandwidth microbenchmark is performed on two processes, where both processes repeatedly perform a stream of shmem_putmem() operations within a fixed window size before performing a shmem_quiet() to ensure remote completion. Figure 3b shows the throughput results between nodes. For small message sizes, SOS utilizes the libfabric inject feature to accelerate the small message pathway through the uGNI provider. After reaching the inject threshold of 64 bytes, SOS switches to bounce buffering until 2 KB in order to immediately achieve local completion without stalling outgoing transactions. We find these two features give noticeable improvement, achieving an average of *61%* relative performance improvement compared to Cray SHMEM. At 4 KB the BTE engine is utilized by both SOS and Cray SHMEM. This transition levels out the results; SOS keeps pace with Cray SHMEM with a *2%* average relative deviation.

The SOS uni-directional read bandwidth microbenchmark is performed on two processes, where one process repeatedly reads from the remote process through a shmem_getmem() operation. In this case remote completion is implied upon return. Figure 3a shows that Cray SHMEM and SOS have comparable results. For get results SOS was tuned to exercise the BTE engine at 2 KB which enables a temporary performance gain over Cray SHMEM's default 4 KB BTE switch. Overall SOS shows competitive performance numbers that are on average within *5%* relative to Cray SHMEM's performance.



(a) Uni-directional bandwidth (b) Bi-directional bandwidth

Fig. 3. Bandwidth measurements for Sandia OpenSHMEM and Cray SHMEM.

5.3 Random Access Benchmark (GUPs)

The Random Access Benchmark (GUPs) is intended to assess the ability of an interconnect and its associated network software stack to efficiently handle many small, concurrent load/store accesses to a data table distributed across multiple nodes in a systems. The modified version of the HPCC RandomAccess benchmark employs various aggregation algorithms, all of which encounter scalability challenges. Instead, a version of the benchmark was written based on the serial and OpenMP variants. In this version, each PE executes a series of shmem_longlong_g/shmem_longlong_p operations to load an element from the table, XOR the element with a locally generated value, then write the updated element back in to the table. The number of updates per PE scales as the size of the table. The global table size scales linearly as the number of PEs in the job.

The benchmark was run using a local table size of 32 MB, with each PE executing 16 million updates. Verification of the update run was accomplished by rerunning the algorithm, but using shmem_set_lock/shmem_clear_lock on an array nPES in size to implement a critical region around the update procedure. A run is considered successful if 1% or fewer elements are found to be inconsistent. Figure 4 presents the global update rate (giga-updates/sec) when using Cray SHMEM and SOS. For this experiment, SOS was enhanced to allow for backing the symmetric heap with large pages. Rather than calling mmap with MAP_ANON and a NULL file descriptor, a file was created on one of the node-local CLE large page file systems, and subsequently mapped in to the process address space using mmap. For GUPs style memory access patterns, the Aries I/O MMU works best with large pages. The Xeon 2 MB native large page size was used to back the SOS symmetric heap in these tests. Note Cray SHMEM backs the symmetric heap with large pages by default. For jobs using 32 PEs or fewer, the Aries network is not involved as all get/put operations are handled via XPMEM cross mappings. Above 32 PEs, the Aries network is used for a portion of the remote memory updates. As the job size grows, a greater proportion of the updates target off-node memory and hence exercise the Aries network. SOS performance compares favorably to the Cray SHMEM implementation, particularly when using large pages. The combi-

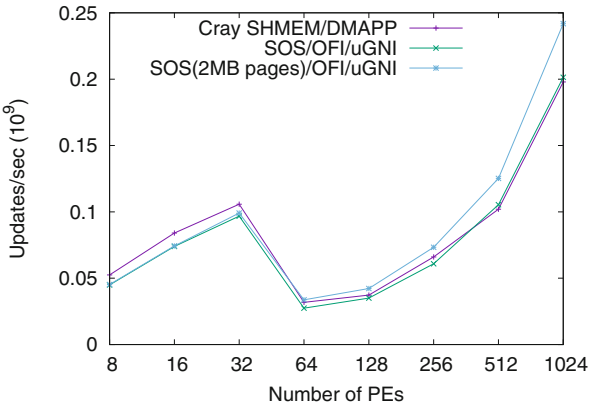
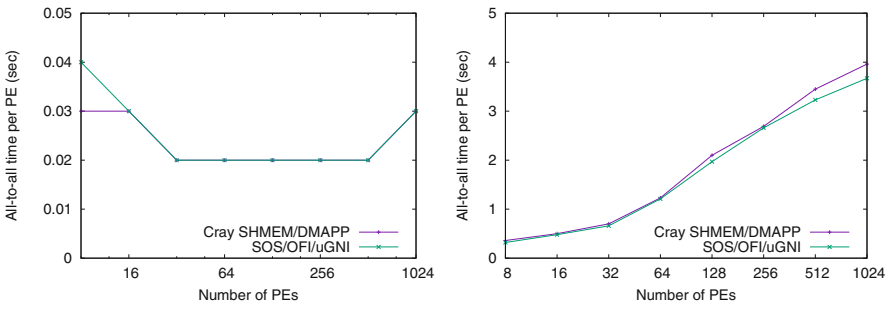


Fig. 4. Giga-updates per second (GUPs) for the RA benchmark on SOS and Cray SHMEM.

nation of the use of the libfabric fi_inject_writedata function in the implementation of shmem_longlong_p and the use of large pages for the symmetric heap, helps SOS to realize a higher update rate than Cray SHMEM at the larger job sizes.

5.4 Scalable Integer Sort Benchmark (ISx)

ISx [15] is a scalable integer sort benchmark using a bucket-sort algorithm. The core communication pattern is an all-to-all exchange of locally sorted keys. The all-to-all exchange is implemented using `shmem_int_put` to deliver the sorted keys to the target PE. The offset into the target array (allocated from the symmetric heap), is determined using a `shmem_longlong_fadd`. A final `shmem_barrier_all` call is invoked to ensure all data has been exchanged. The benchmark allows for both strong and weak scaling. With weak scaling, the number of keys per PE is fixed. For these experiments, ISx was built both with Cray SHMEM and SOS. Except for specifying dynamic linking, no special compiler or linker options were used.



(a) Strong scaling (b) Weak scaling

Fig. 5. Performance of ISx using Sandia OpenSHMEM and Cray SHMEM.

Figure 5a presents the time spent in the all-to-all exchange in the case of strong scaling, sorting a total of 2^{27} keys. Except for the 8 PE run, the time spent in the all-to-all exchange pattern is the same whether using Sandia SHMEM or Cray SHMEM. Figures 5b shows the time spent in the all-to-all exchange pattern for the weak scaling case. For weak scaling, the time in the all-to-all operation is essentially the same, with SOS showing a small performance improvement.

6 Conclusions and Future Work

Sandia OpenSHMEM (SOS) is the first PGAS middleware to demonstrate the new OpenFabrics Interface communication API on a modern HPC system. Significant effort was invested in both SOS and the uGNI provider to broaden the set of supported performance and portability features, including support for additional memory registration and addressing modes. Overall we found OFI to be closely aligned with the requirements of OpenSHMEM, yielding efficient mappings between the OpenSHMEM middleware and the lower-layer libfabric interfaces.

We evaluated the performance of our implementation on a Cray® XC40™ system and demonstrated comparable latency and scalability to the production Cray SHMEM library. SOS with OFI achieved comparable or better bandwidth and random access (GUPs) performance than Cray SHMEM. For small messages, the OFI inject functionality used by SOS resulted in an improvement of up to *61%* in bi-directional bandwidth. In addition, the SOS bounce buffering optimization enabled further improvements in the small-to-medium message regimes. We hope to further improve upon these results with additional performance and scalability tuning.

The OpenSHMEM community is actively working to extend the OpenSHMEM model with new tools that will allow users to leverage future extreme scale systems. We hope that the new, open source platform that we have presented will provide a useful environment for evaluating and developing new extensions to the OpenSHMEM parallel programming model.

Acknowledgements. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. We also thank the OpenFabrics Interfaces Working Group (OFIWG) and its attendees, whose participation has enabled the cooperative design of the libfabric interfaces. This publication has been approved for public, unlimited distribution by Los Alamos National Laboratory, with document number LA-UR-16-24359.

evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance.

References

1. Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Krishna, J., Lusk, E., Thakur, R.: PMI: a scalable parallel process-management interface for extreme-scale systems. In: Keller, R., Gabriel, E., Resch, M., Dongarra, J. (eds.) EuroMPI 2010. LNCS, vol. 6305, pp. 31–41. Springer, Heidelberg (2010). doi:10.1007/978-3-642-15646-5_4
2. Barrett, B.W., Brightwell, R., Hemmert, S., Pedretti, K., Wheeler, K., Underwood, K., Riesen, R., Maccabe, A.B., Hudson, T.: The portals 4.0.2 network programming interface. Technical report SAND2013-3181, Sandia National Laboratories, April 2013
3. Barrett, B.W., Brigthwell, R., Hemmert, K.S., Pedretti, K., Wheeler, K., Underwood, K.D.: Enhanced support for OpenSHMEM communication in Portals. In: 19th Annual Symposium on High Performance Interconnects, August 2011
4. Barrett, B.W., Hammond, S.D., Brightwell, R., Hemmert, K.S.: The impact of hybrid-core processors on MPI message rate. In: Proceedings of 20th European MPI Users' Group Meeting, EuroMPI 2013, pp. 67–71 (2013)
5. Birrittella, M.S., Debbage, M., Huggahalli, R., Kunz, J., Lovett, T., Rimmer, T., Underwood, K.D., Zak, R.C.: Intel ® Omni-path architecture: enabling scalable, high performance fabrics. In: Proceedings of 23rd Annual Symposium on High-Performance Interconnects, pp. 1–9, August 2015
6. Bonachea, D.: GASNet specification, v1.1. Technical report UCB/CSD-02-1207, University of California, Berkeley, October 2002
7. Choi, S.E., Pritchard, H., Shimek, J., Swaro, J., Tiffany, Z., Turrubiates, B.: An implementation of OFI libfabric in support of multithreaded PGAS solutions. In: Proceedings of 9th International Conference on Parititioned Global Address Space Programming Models, PGAS 2015, September 2015
8. Cray Inc.: Using the GNI and DMAPP APIs. Technical report S-2446-3103, Cray Inc. (2011)
9. Derradji, S., Palfer-Sollier, T., Panziera, J.P., Poudes, A., Atos, F.W.: The BXI interconnect architecture. In: Proceedings of IEEE 23rd Annual Symposium on High-Performance Interconnects, pp. 18–25, August 2015
10. Dongarra, J., Luszczek, P.: Introduction to the HPCChallenge benchmark suite. Technical report ICL-UT-05-01, ICL (2005)
11. Flajslik, M., Dinan, J., Underwood, K.D.: Mitigating MPI message matching misery. In: Kunkel, J.M., Balaji, P., Dongarra, J. (eds.) ISC High Performance 2016. LNCS, vol. 9697, pp. 281–299. Springer, Heidelberg (2016). doi:10.1007/978-3-319-41321-1_15
12. Gabriel, E., et al.: Open MPI: goals, concept, and design of a next generation MPI implementation. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J. (eds.) EuroPVM/MPI 2004. LNCS, vol. 3241, pp. 97–104. Springer, Heidelberg (2004). doi:10.1007/978-3-540-30218-6_19
13. Girolamo, S.D., Jolivet, P., Underwood, K.D., Hoefler, T.: Exploiting offload enabled network interfaces. In: Proceedings of 23rd Annual Symposium on High-Performance Interconnects. IEEE, August 2015

14. Grun, P., Hefty, S., Sur, S., Goodell, D., Russell, R., Pritchard, H., Squyres, J.: A brief introduction to the OpenFabrics interfaces - a new network API for maximizing high performance application efficiency. In: Proceedings of 23rd Annual Symposium on High-Performance Interconnects, August 2015
15. Hanebutte, U., Hemstad, J.: ISx: a scalable integer sort for co-design in the exascale era. In: 2015 9th International Conference on Partitioned Global Address Space Programming Models (PGAS), pp. 102–104, September 2015
16. Janjusic, T., Shamis, P., Venkata, M.G., Pool, S.W.: OpenSHMEM reference implementation using UCCS-uGNI transport layer. In: Proceedings of 8th International Conference on Partitioned Global Address Space Programming Models, PGAS 2014 (2014)
17. Jose, J., Kandalla, K., Luo, M., Panda, D.K.: Supporting hybrid MPI and OpenSHMEM over InfiniBand: design and performance evaluation. In: Proceedings of 41st International Conference on Parallel Processing, ICPP 2012, pp. 219–228, September 2012
18. Jose, J., Zhang, J., Venkatesh, A., Potluri, S., Panda, D.K.D.K.: A comprehensive performance evaluation of OpenSHMEM libraries on InfiniBand clusters. In: Poole, S., Hernandez, O., Shamis, P. (eds.) OpenSHMEM 2014. LNCS, vol. 8356, pp. 14–28. Springer, Heidelberg (2014). doi:10.1007/978-3-319-05215-1_2
19. Kumar, S., Mamidala, A.R., Faraj, D.A., Smith, B., Blocksome, M., Cernohous, B., Miller, D., Parker, J., Ratterman, J., Heidelberger, P., Chen, D., Steinmacher-Burrow, B.: PAMI: a parallel active message interface for the Blue Gene/Q supercomputer. In: Proceedings of 26th International Parallel Distributed Processing Symposium, IPDPS 2012, pp. 763–773, May 2012
20. Libfabric. http://ofiwg.github.io/
21. Luo, M., Seager, K., Murthy, K.S., Archer, C.J., Sur, S., Hefty, S.: Early evaluation of scalable fabric interface for PGAS programming models. In: Proceedings of 8th International Conference on Partitioned Global Address Space Programming Models, PGAS 2014 (2014)
22. Reference OpenSHMEM implementation. https://github.com/openshmem-org/openshmem
23. OpenSHMEM application programming interface, version 1.3, February 2016. http://www.openshmem.org
24. Pritchard, H., Harvey, E., Choi, S.E., Swaro, J., Tiffany, Z.: The GNI provider layer for OFI libfabric. In: Proceedings of Cray User Group Meeting, CUG 2016, May 2016
25. Sandia OpenSHMEM. https://www.github.com/Sandia-OpenSHMEM/
26. Shamis, P., Venkata, M.G., Kuehn, J.A., Poole, S.W., Graham, R.L.: Universal common communication substrate (UCCS) specification, version 0.1. Technical report ORNL/TM-2012/339, Oak Ridge National Laboratory (2012)
27. Shamis, P., Venkata, M.G., Lopez, M.G., Baker, M.B., Hernandez, O., Itigin, Y., Dubman, M., Shainer, G., Graham, R.L., Liss, L., Shahar, Y., Potluri, S., Rossetti, D., Becker, D., Poole, D., Lamb, C., Kumar, S., Stunkel, C., Bosilca, G., Bouteiller, A.: UCX: an open source framework for HPC network APIs and beyond. In: Proceedings of IEEE 23rd Annual Symposium on High-Performance Interconnects, pp. 40–43, August 2015
28. Shamis, P., Venkata, M.G., Poole, S., Welch, A., Curtis, T.: Designing a high performance OpenSHMEM implementation using universal common communication substrate as a communication middleware. In: Poole, S., Hernandez, O., Shamis, P. (eds.) OpenSHMEM 2014. LNCS, vol. 8356, pp. 1–13. Springer, Heidelberg (2014). doi:10.1007/978-3-319-05215-1_1