Implementing OpenSHMEM using MPI-3 one-sided communication

Jeff R. Hammond[1], Sayan Ghosh[2], and Barbara M. Chapman[2]

Argonne National Laboratory[1]
Argonne IL, 60439
jhammond@alcf.anl.gov
Dept. of Computer Science[2]
University of Houston
Houston, Texas
sgo@cs.uh.edu,chapman@cs.uh.edu

Abstract. This paper reports the design and implementation of OpenSHMEM over MPI using new one-sided communication features in MPI-3, which include not only new functions (e.g. remote atomics) but also a new memory model that is consistent with that of SHMEM. We use a new, non-collective MPI communicator creation routine to allow SHMEM collectives to use their MPI counterparts. Finally, we leverage MPI shared-memory windows within a node, which allows direct (load-store) access. Performance evaluations are conducted for shared-memory and InfiniBand conduits using microbenchmarks.

Keywords: SHMEM, MPI-3, RMA, one-sided communication

1 Introduction

SHMEM [1,10] is a one-sided communication interface originally developed for Cray systems but subsequently adopted by numerous vendors (SGI, Quadrics, IBM, Mellanox, etc.) for use in high-performance computing. OpenSHMEM [7] represents a community effort to standardize SHMEM in order to enable portable applications and grow the user base. There are essentially two kinds of SHMEM implementations: (1) platform-specific, proprietary i.e. closed-source, highly optimized implementations and (2) portable (at least to some extent), open-source reference implementations. Examples of the former include SGI-SHMEM and CraySHMEM, while the latter includes the OpenSHMEM reference implementation from University of Houston, based upon widely portable GASNet [5] and SHMEM based upon Portals4 [4], which is portable to the most common commodity networks. MVAPICH2-X [14] is not proprietary in the sense that it is freely available and not distributed by any platform vendor, but it is currently closed-source and only supports InfiniBand, for which it is highly optimized.

Among the current reference implementations – that is, the ones based upon GASNet and Portals4, respectively – are limited in portability only by their underlying conduits. GASNet has broad support of both commodity and HPC networks; we are not aware of any widely used platform that is not supported. Despite the wide portability of GASNet, it is not supported directly by vendors nor is it shipped in binary form for commodity systems (in part due to the different ways PGAS compilers use it, which require different configurations) and it lacks explicit support for atomic operations, which prevents the use of native hardware implementations when available. On the other hand, the Portals4 implementation is itself a reference implementation of the Portals4 specification [3], which aims to have optimized implementations where at least some of the features have customized hardware support. The Portals4 reference implementation currently supports shared memory, TCP/IP and InfiniBand. In contrast to the aforementioned conduits, MPI is supported on the widest variety of platforms, is supported by all major HPC vendors and can be trivially installed via the appropriate package management system on the most common platforms. Broad portability and vendor support are not the only advantages of an MPI-based implementation. The MPI ecosystem includes powerful performance and debugging tools as well as parallel math and I/O libraries (as just two examples), all of which are now available for use in SHMEM applications. Finally, because GASNet nor Portals4 provide collective operations, any implementation of these must be implemented on top of point-to-point operations inside of the SHMEM library. On the other hand, an MPI-based implementation immediately leverages many years of algorithm and software development of MPI collectives. The relatively recent release of the MPI-3 standard has made possible – for the first time – a direct implementation of SHMEM and PGAS programming models using the one-sided functionality therein. Prior to MPI-3, lack of important atomic operations (e.g. fetch-and-add and compare-and-swap), inconvenient memory model (designed to support non-cache coherent systems), and awkward synchronization functions made MPI one-sided communication an inadequate conduit for models like SHMEM. Additional criticism and analysis of the MPI-2 one-sided communication in the context of PGAS can be found in Bonachea and Duell [6] and Dinan, et al. [8]. With the features provided in MPI-3, it is possible to implement SHMEM relatively efficiently since essentially all SHMEM calls map directly to one or two MPI calls and the synchronization modes in MPI-3 are not excessive relative to SHMEM semantics. Given this, the limiting factor in SHMEM performance when implemented using MPI-3 is the quality of the MPI implementation. The purpose of this paper is to demonstrate the first implementation of OpenSHMEM using MPI-3 as the conduit. We are leaving in-depth performance analysis/profiling and discussion on RMA implementation quality for future work. We refer to our implementation of OpenSHMEM over MPI-3 as OSHMPI. We compare our implementation to existing implementations for shared memory and InfiniBand, which include those based upon GASNet and Portals4 as well as the optimized MVAPICH2-X implementation. We also compare OSHMPI with a vendor implementation of OpenSHMEM from Mellanox, called Mellanox-ScalableSHMEM which works with the OpenFabrics RDMA for Linux stack (OFED).

2 Background

In order to motivate our design choices, we summarize the important semantics of the SHMEM and MPI-3 models to understand how they must be reconciled in the implementation and the performance effects associated therewith.

2.1 SHMEM

One-sided communication SHMEM one-sided communication operations are locally blocking, meaning they return once the local buffer is available for reuse. Single-element, contiguous and strided variants of Put (remote write) and Get (remote read) are provided. For each of these, the type is encoded in the function name, which enables compiler type-checking (for example, this is not available in MPI C code except via collaboration of compiler extensions and metadata in the MPI header file [11]). SHMEM also supports the common atomic operations of swap, compare-and-swap, add, fetch-and-add, increment and fetch-and-increment, all of which block on local completion, which entails a round trip in four of the six cases.

The synchronization primitives for one-sided operations are Fence and Quiet, with the former corresponding to point-wise ordering and the latter to ordering with respect to all remote targets. Some implementations specify that these ordering points correspond to remote completion, which is sufficient but not always necessary to ensure ordering. In any case, we choose the conservative interpretation - these functions enforce remote completion of one-sided operations.

Collective communication SHMEM provides Barrier, Broadcast, Reduction and Gather operations with type support encoded into the function names, just as for one-sided operations. Subsets of processes (referred as PEs or Processing Elements in SHMEM) are described using `(PE_start, logPE_stride, PE_size)` tuples, which is not fully general, but are still useful for some applications. With the exception of `shmem_barrier_all`, collectives take an argument `pSync`, which presumably allows for the implementation to avoid additional internal state for collectives.

2.2 MPI-3

Barrett, et al. discussed MPI-3 RMA semantics in detail in Ref. [2]; we summarize only the salient points related to SHMEM here.

One-sided communication The MPI 3.0 standard [17] represents a significant change from previous iterations of MPI, particularly with respect to one-sided communication (RMA). New semantics for memory consistency, ordering and passive-target synchronization were introduced, all three of which have a significant (positive) impact on an MPI-based implementation of SHMEM.

The unified memory model of MPI-3 RMA stipulates that direct local access and RMA-based access to the memory associated with a window[1] see the same data (but not necessarily *immediately*) without explicit synchronization. This model is not required, rather the user must query for it, but it should be possible for implementations to support this on cache-coherent architectures.

Prior to MPI-3, RMA operations were specified as unordered and the only means for ordering operations was to remote-complete them. This entails a rather high overhead and so MPI-3 now specifies that accumulate operations are ordered by default; these operations have always been specified as element-wise atomic, unlike Put and Get. The user can inform the implementation that ordering is not required but then the user is required to enforce ordering via remote completion in the application.

In MPI-2, passive target synchronization was specified in the form of an epoch delineated with calls to `MPI_Win_lock` and `MPI_Win_unlock`. Only upon returning from the latter call was any synchronization implied and it implied global visibility, i.e. remote completion. MPI-3 provides the user the ability to specify local and remote completion separately and to do so without terminating an epoch. These semantics are more consistent with SHMEM and ARMCI [18] as well as many modern networks.

Collective communication MPI collective communication occurs on a communicator, which is an opaque object associated with a group processes. Communication on one communicator is independent of communication on another, which enables strict separation of different sets of messages in the case of point-to-point and allows for a well-defined semantic for collective operations on overlapping groups of processes. In MPI-2, communicators could only be created collectively on the parent communicator, meaning that a subcommunicator to be derived from the default (world) communicator (containing all processes) could not be created without the participation of all processes. This precluded their use in SHMEM collectives unless all possible subcommunicators were created at initialization, which is obviously unreasonable.

MPI-3 introduced a new function for creating subcommunicators that is collective only on the group of processes that are included in the new communicator [9]. This enables subcommunicators associated with `(PE_start, logPE_stride, PE_size)` tuples to be created on the fly as necessary. Of course, creating subcommunicators on the fly is potentially expensive relative to a particular collective operation, so a high-quality implementation of SHMEM over MPI-3 would maintain a cache of these since it is reasonable to assume that they will be reused. The MVAPICH2-X implementation of OpenSHMEM does this internally despite not explicitly using the MPI-3 interface for collectives [16]. Another potential bottleneck in this process is the translation of the root PE (necessary only for broadcast operations) to a process in the subcommunicator, which is $O(N)$ in

[1] A window is the opaque memory registration object of MPI RMA upon which all one-sided operations act.

space and time [23]. However, we can avoid this translation routine if necessary due to the restricted usage necessary in SHMEM collectives.

3 Implementation Design

In this section, we outline the design of the mapping from SHMEM to MPI-3. The mapping of SHMEM functions to MPI-3 ones is mostly straightforward due to the flexibility MPI-3 RMA, but there are a few key issues that must be addressed.

Symmetric Heap: The use of symmetric variables is a unique concept in one-sided communication that deserves special mention [20]. SHMEM communication operations act on virtual addresses associated with symmetric variables, which include data in the symmetric heap (dynamically allocated) and statically allocated data, such as global variables and variables declared with the static attribute. Communication with stack variables is not supported within OpenSHMEM. On the other hand, the MPI window object is opaque and communication operations act on data in the window specified via offsets relative to the window base, which can be different on every PE. Whereas SHMEM requires all allocations from the symmetric heap be symmetric (i.e. uniform across all PEs), MPI supports the general case where PEs can pass different sizes (including zero) to the window constructor routine. Prior to MPI-3, the only window constructor routine was `MPI_Win_create`, which was a registration routine that took local memory buffers as input. This precluded the use of symmetric allocations for scalable metadata; it had to be assumed that the base address was different at every PE, thus requiring $O(N)$ metadata in the window object. MPI-3 provides a new routine for constructing windows that includes memory allocation (`MPI_Win_allocate`), hence permits the implementation to allocate symmetrically. It also permits the use of shared memory segments for intranode optimization. This is available explicitly to the user via `MPI_Win_allocate_shared` and implicitly in the case of `MPI_Win_allocate`. By explicit, we mean that the user can query the virtual address associated with a window segment in another PE on the same node and access it via load-store; the implicit case is where the implementation uses shared memory to bypass the network interface when the user makes MPI communication calls.

Put and Get: SHMEM performs communication against symmetric data, which can be either global, static or symmetric heap (*sheap*) data. The latter is quite easy to deal with; we allocate an MPI window of sufficient size (controlled by an environment variable) and allocate memory out of it. To access this data remotely, one merely translates the local address into a remote offset within the sheap window, which is not expensive. Global data is registered with MPI using `MPI_Win_create` at initialization using the appropriate operating system mechanism to get the base address and size of this region. We use one window for *bss* and *data* segments (global data can reside in both places) but two may be required in some cases. The lookup function (`__shmem_window_offset`) differentiates between the two windows. We use a much simpler special case as that of [8] because valid symmetric variables always fall within one of two windows. Because the symmetric heap window is allocated rather than just registered, it supports directly local access within a node, so for this case, we use direct access when all PEs reside in a single node. This can be generalized to multiple nodes using overlapping windows – one each for internode and intranode communication[2] – but this has not yet been implemented.

Fig. 1: The implementation of SHMEM put using MPI for one type-variant. The code is modified from the original for presentation purposes.

```
void __shmem_put(MPI_Datatype type, int type_size, void *target,
                 const void *source, size_t len, int pe)
{
    enum shmem_window_id_e win_id;
    shmem_offset_t offset;
    __shmem_window_offset(target, pe, &win_id, &offset));
    if (world_is_smp && win_id==SHEAP_WINDOW) {
        void * ptr = smp_sheap_ptrs[pe] + (target - sheap_base_ptr);
        memcpy(ptr, source, len*type_size);
    } else {
        MPI_Win win = (win_id==SHEAP_WINDOW) ? sheap_win : text_win;
        int n = (int)len; assert(len<(size_t)INT32_MAX);
        MPI_Accumulate(source, n, type, pe, offset, n, type, MPI_REPLACE, win);
        MPI_Win_flush_local(pe, win);
    }
}

void shmem_int_put(int *target, const int *source, size_t len, int pe)
{    __shmem_put(MPI_INT, 4, target, source, len, pe);    }
```

Synchronization: The synchronization primitives `shmem_fence` and `shmem_quiet` are both mapped to `MPI_Win_flush_all` in order to ensure pairwise and global ordering of one-sided operations. Because `shmem_fence` does not take a specific PE as an argument, the implementation would like have to maintain $O(N)$ state to implement the minimum synchronization required. We assume that the MPI implementation already tracks the remote processes and only flushes those that are the the target of communication and thus it is redundant for our implementation to do this. The assumption that `MPI_Win_flush_all` is an efficient way to implement `shmem_fence` may not always be true, but it is perhaps worth noting that the Portals4 implementation does something similar to avoid $O(N)$ state.

Atomics Atomic operations map from SHMEM to MPI similarly as with Put and Get. Table 1 specifies how each SHMEM function translates to an MPI function. Because `shmem_inc` and `shmem_finc` are just special cases of `shmem_add` and `shmem_fadd`, respectively, we do not list them.

[2] The need for two windows may be obviated in a future version of the MPI standard.

Table 1: Correspondance between SHMEM and MPI atomic operations.

SHMEM function	MPI function	Accumulate operation
shmem_cswap	MPI_Compare_and_swap	-
shmem_swap	MPI_Fetch_and_op	MPI_REPLACE
shmem_fadd	MPI_Fetch_and_op	MPI_SUM
shmem_add	MPI_Accumulate	MPI_SUM

Collective operations We follow the same approach as [16] with respect to non-collective communicator creation [9]. Figure 2 shows the code that is used to translate a SHMEM PE group triplet to an MPI subcommunicator. Only Broadcast requires the rank translation of the root; when an invalid rank (e.g. -1) is passed to this function, translation is skipped. The use of a cache for communicators is an obvious optimization but one that is not yet implemented in OSHMPI.

Table 2 shows the mapping from SHMEM to MPI with respect to collective operations. Because the `shmem_collect` routine provides only the count at each PE, the MPI implementation requires an `MPI_Allgather` to form the vector of counts. The translation from SHMEM reduction operators to their MPI counterparts is trivial and is left as an exercise for the reader.

Fig. 2: Code to create an MPI sub communicator associated with a PE subgroup.

```
void __shmem_acquire_comm(int pe_start, int pe_logs, int pe_size,
                          MPI_Comm * comm, int pe_root, int * broot)
{
    if (pe_start==0 && pe_logs==0 && pe_size==shmem_world_size) {
        *comm  = SHMEM_COMM_WORLD; *broot = pe_root;
    } else {
        MPI_Group strgrp;
        int * pe_list = malloc(pe_size*sizeof(int)); assert(pe_list!=NULL);
        int pe_stride = 1<<pe_logs;
        for (int i=0; i<pe_size; i++) pe_list[i] = pe_start + i*pe_stride;
        MPI_Group_incl(SHMEM_GROUP_WORLD, pe_size, pe_list, &strgrp);
        MPI_Comm_create_group(SHMEM_COMM_WORLD, strgrp, pe_start, comm);
        if (pe_root>=0) *broot = __shmem_translate_root(strgrp, pe_root);
        MPI_Group_free(&strgrp);
        free(pe_list);
    }
}
```

Table 2: Mapping of SHMEM collectives to MPI collective functions.

SHMEM	MPI
shmem_barrier	MPI_Barrier
shmem_broadcast	MPI_Bcast
shmem_collect	MPI_Allgatherv
shmem_fcollect	MPI_Allgather
shmem_<op>_to_all	MPI_Allreduce(op)

4 Results

In this section, we evaluate the performance of OSHMPI versus other implementations of OpenSHMEM (GASNet, Portals4, MVAPICH2-X and Mellanox). While these are not the only OpenSHMEM implementations available, they are a representative set and sufficient to make a reasonable evaluation of the quality of our implementation and of MPI-3 as a conduit. In particular, the comparison of OSHMPI using the MPI-3 implementation found in MVAPICH2 to the OpenSHMEM implementation in MVAPICH2-X is particularly useful, since this uses at least some of the same implementation features and thus exposes more of the semantic differences. However, as will be shown below, there appear to be implementation issues that prevent MPI-3 from achieving its full potential, i.e. not all the differences are due to semantics. Most of the test cases are taken from publicly available benchmarks or example codes packaged with the OpenSHMEM reference API.

The evaluation platform used is a dual-socket AMD 6128 (8 cores/socket) cluster with QDR InfiniBand from Mellanox and 64 GB of memory per node. We use the latest release of each of the implementations considered. OSHMPI uses MPICH 3.1b2 for SMP, and MVAPICH2 2.0a for distributed cases. The OpenSHMEM reference implementation uses GASNet 1.20 configured for GASNet “smp” and “ibv” conduits only; this implementation is referred to as GASNet. For SHMEM-Portals and Portals4 (henceforth, Portals4), the repository trunk is used, configured with `--with-implementation=ib --enable-ib-shmem`.

MVAPICH2-X 2.0a provides the OpenSHMEM implementation, which is denoted as MVAPICH2-X. Mellanox-ScalableSHMEM version 2.0 is also included in our evaluation (configured with only `--with-oshmem`), denoted as MLNX.

4.1 OpenSHMEM versus MPI-3 – implementation effects

Figure 3 compares the message rate for messages from 8 byte to 8 MB using tests written for the MPI-3 and OpenSHMEM interfaces and implemented with MVAPICH2 and MVAPICH2-X, respectively. The purpose of this test is to elucidate differences in the implementation of the two protocols within a presumably similar implementation. The differences need not necessarily be so large but MVAPICH2 inherits an implementation design for one-sided that is not targeting one-sided networks, whereas the OpenSHMEM implementation clearly exploits the one-sided nature of InfiniBand in a direct way.

4.2 Latency and message-rate evaluation

In this section, we evaluate the performance of the OSHMPI, Portals4, GASNet, Mellanox-ScalableSHMEM and MVAPICH2-X implementations using the OSU microbenchmarks [15]. These tests measures the message rate, average latencies for varying message sizes and types of one-sided operations. Figures 4a, 4b, 4c and 4d show the average latencies on a shared memory system or across the network on distributed nodes with increasing data sizes from 1 to 2^{20} bytes.

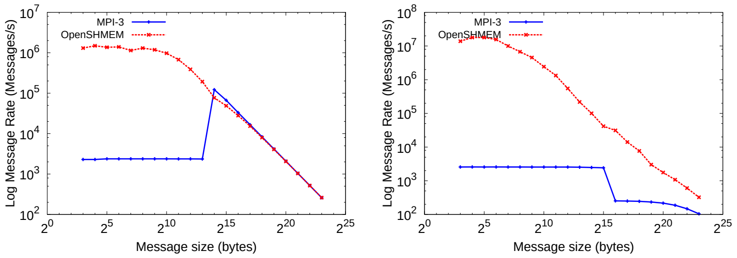


Fig. 3: Internode and intranode (both with 2 PEs) message rate (`long` *puts*) of MPI-3 and OpenSHMEM interfaces as implemented with MVAPICH2 and MVAPICH2-X, respectively.

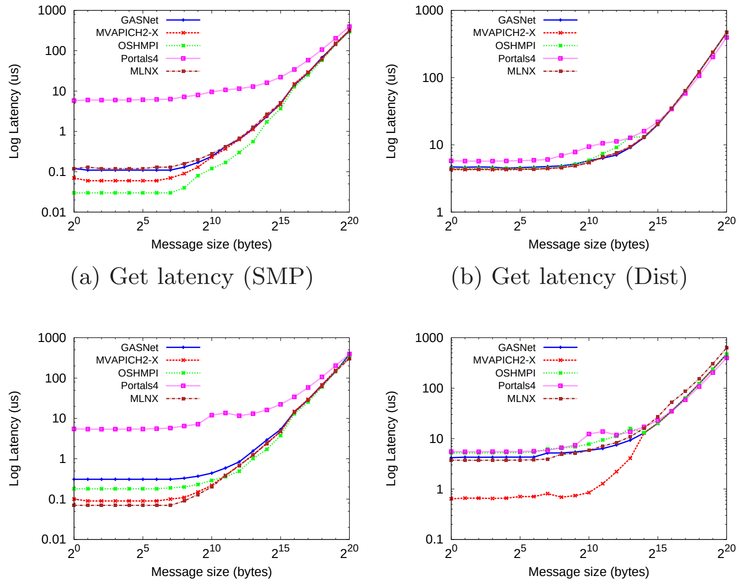
Figure 5 shows aggregate unidirectional put injection rate with message size varying from 1 to 2^{22} bytes on shared-memory and distributed nodes.

The shared-memory performance of OSHMPI is generally superior as compared to others, which is due to the use of MPI-3 shared memory windows that allows direct load-store access on the target memory without any additional overhead. Portals4 cannot do this due to the lack of XPMEM support and GASNet appears to require additional overhead, either due to locking or copying through shared segments. Mellanox-ScalableSHMEM has performance close to GASNet for distributed nodes, the sharp drop/rise in the performance plot (Figure 5) suggests message-transfer protocol crossover at certain sizes. On the other hand, OSHMPI suffers from poor message rate/latency on distributed nodes. This is mostly due to the implementation quality but a small portion of the overhead can be attributed to the requirement of two MPI function calls to implement a blocking Put operation – since `MPI_Put` is nonblocking, it must be followed by `MPI_Win_flush_local` – which may entail more software overhead than implementations that use only a single call to the conduit API.

The operation rate test for OpenSHMEM atomic routines are similar to the Put message-rate test. The benchmark measures the performance of atomic fetch-operate routines supported in OpenSHMEM by issuing back-to-back atomic operations of a type from the origin to the target PE. Figure 6 shows the average latency and aggregate message rate per atomic operation for all the atomic operations between two PEs on two nodes.

4.3 SHMEM Barrier performance

Barriers are used extensively in parallel programs, perhaps unnecessarily in some cases, but they are nonetheless an essential collective operation that needs to be efficient. Figure 7 shows barrier latencies for the shared memory and distributed cases (on 2, 4, 8, and 16 PEs). OSHMPI and MVAPICH2-X have nearly identical performance, indicating the same underlying implementation (using the MPI collective infrastructure in MVAPICH2). The OSHMPI routine `shmem_barrier` also performs local and remote synchronization, meaning a memory barrier and a



(a) Get latency (SMP)

(b) Get latency (Dist)

(c) Put latency (SMP)

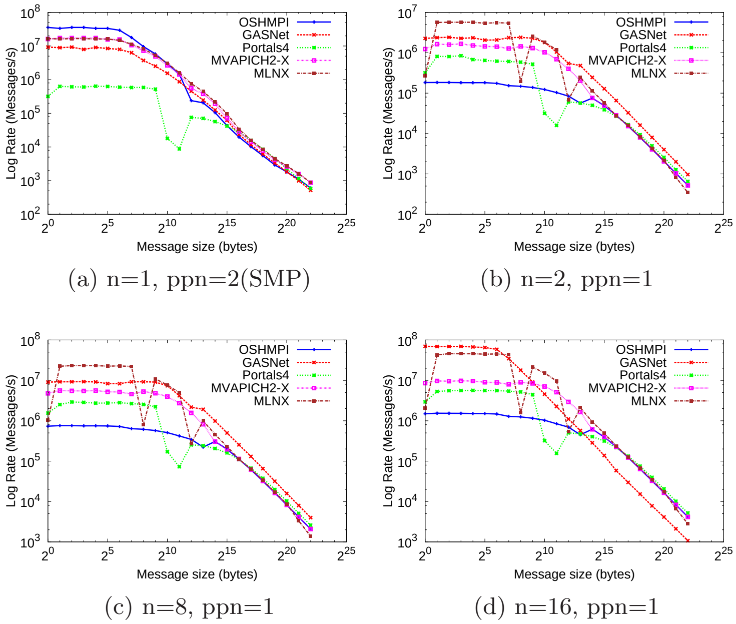
(d) Put latency (Dist)

Fig. 4: Get/Put latencies on 2 PEs of one node (SMP) and two nodes (Dist).

remote flush of all outstanding communication operations, which is not explicitly required by the OpenSHMEM specification but is implied by examples programs therein. Barrier illustrates a significant benefit of using MPI as a conduit for SHMEM; collective operations are heavily optimized in MPI implementations and often use the best available algorithms. Building collectives on top of point-to-point operations in a SHMEM-oriented conduit may make it difficult or even impossible to achieve the same performance as MPI. For example, MPI leverages hardware implementations of collectives on systems such as IBM Blue Gene. Alternatively, a SHMEM-oriented conduit may not provide the most appropriate point-to-point operations for implementing synchronous collectives, thereby requiring the SHMEM collective implementation to poll on memory locations or use other inefficient protocols.

4.4 Solving 2D heat equation

The 2D heat benchmark predicts the heat distribution, resulting from conduction in a 2D domain and could be solved iteratively using - Jacobi, Gauss-Seidel and Successive Over-relaxation methods. The benchmark code (`shmem_2dheat.c`) is available with the OpenSHMEM reference API package. In the 2D heat benchmark, data is distributed evenly across PEs (plus one or more rows to facilitate ”ghost” transfers from neighbors). Rows of data are communicated between adjacent PEs, with the communication overhead of $(2 * npes - 1)$ per iteration. The



(a) n=1, ppn=2(SMP)

(b) n=2, ppn=1

(c) n=8, ppn=1

(d) n=16, ppn=1

Fig. 5: Unidirectional Put message-rate on 2, 8, and 16 PEs within a node (top-left) and across nodes.

results of the 2D heat benchmark on 128, 256, 512, and 1024 PEs for OSHMPI, MVAPICH2-X, GASNet, Mellanox-ScalableSHMEM and Portals4 on a (32K × 32K) matrix are shown in Figure 8. For this particular benchmark, some of the GASNet runs terminated with a segmentation fault, hence we are unable to show GASNet results beyond 256 PEs.

5 Observations

For shared memory systems, the performance of OSHMPI and MVAPICH2-X is comparable, which is not surprising given that the implementations are documented to use the same optimizations. The Portals4 intranode performance is not surprisingly slow given that XPMEM could not be used due to the inability to install this kernel module because it requires elevated privileges. [3] Additional performance artifacts are seen in Portals4 for messages between 1 and 64 KiB in internode tests, which may be the result of protocol crossover effects.

In distributed case however, both in terms of latency and message rate, OSHMPI is noticeably worse than other implementations. One can only assume that the MPI-3 implementation in MVAPICH2 is not as optimized as the

[3] XPMEM is a Linux kernel module (originally developed by SGI) that enables a process to attach memory segments from another process to it's address space.

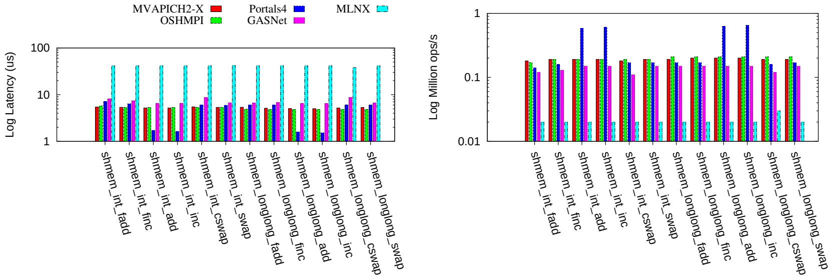


Fig. 6: Atomic latency and operations rate between 2 PEs across 2 nodes

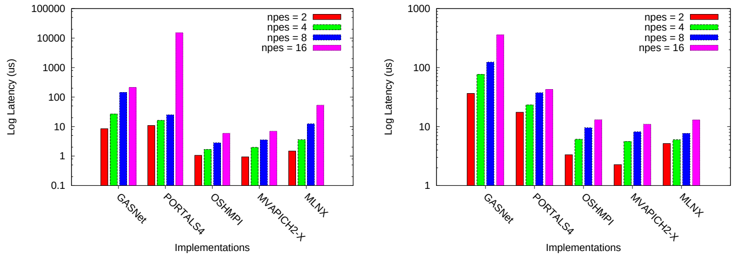


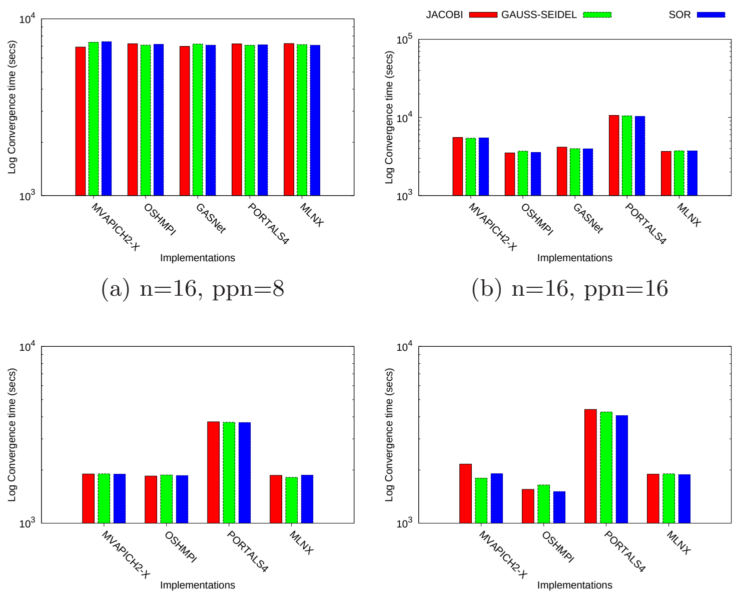
Fig. 7: Barrier latencies on 2, 4, 8, and 16 PEs within a node (left) and across nodes (right). n=1,ppn=*npes* for SMP and n=*npes*,ppn=1 for Dist.

SHMEM implementation in MVAPICH2-X (see Figure 3). For larger messages, however, both the direct and indirect (that is, via MPI-3) performance is similar, so the MPI-3 implementation appears to be related to the short-message implementation, which one hopes will be optimized in future releases of MVAPICH2. GASNet performs the best for short-to-medium messages but is not as robust as the others when a larger number of PEs are used; we were unable to run the 2D heat test for 256 PEs or larger with this implementation.

The atomic latency and message rate performance of MVAPICH2-X and OSHMPI for distributed nodes are found to be very similar, as evident from Figure 6, suggesting that the MVAPICH2 SHMEM and MPI-3 implementations are of similar quality.

We notice significant latency variations across SHMEM implementations of barrier routines (shown in Figure 7). Particularly, GASNet and Portals4 latencies are a minimum ∼10x to that of OSHMPI and MVAPICH2-X on two nodes. GASNet performance significantly degrades for 16 distributed PEs.

We had also performed other collective tests – broadcast, reduce and collect – and observed that GASNet performance was substantially worse for collect/reduce (for both SMP and distributed nodes). On the other hand, OSHMPI and MVAPICH2-X have identical performance for both shared and distributed cases, which is not surprising given their use of the same infrastructure internally.



(a) n=16, ppn=8

(b) n=16, ppn=16

(c) n=64, ppn=8

(d) n=64, ppn=16

Fig. 8: 2D-Heat benchmark performance of SHMEM implementations on 128/256/512/1024 PEs for 32K × 32K matrix

6 Related Work

Since the introduction of SHMEM for Cray T3D, there have been several other implementations, including QSHMEM [21], HP-SHMEM [12], SGI-SHMEM [24], GPSHMEM [19], and IBM TurboSHMEM [13], each with distinct API specifications. In addition to MVAPICH2-X and Portals-SHMEM, Gator-SHMEM [25] and Mellanox ScalableSHMEM [22] are additional implementations of the OpenSHMEM API.

7 Conclusions and Future Work

This paper describes the initial design and implementation of an OpenSHMEM implementation using MPI-3 as a communication conduit. With the recent improvements in the MPI-3 specification, particularly related to RMA, MPI is now a suitable conduit for PGAS programming models like SHMEM. The simplicity of our implementation indicates a good semantic match between the two models. Additionally, the performance is similar to existing PGAS runtimes such as GASNet, Portals4, Mellanox-ScalableSHMEM and MVAPICH2-X for many cases, although clearly there is room for improvement for distributed memory. On the other hand, the intranode, i.e. shared-memory, performance was excellent and in many cases better than the others, suggesting that MPI shared-memory windows are an effective way to optimize one-sided communication within a node.

The performance of collective operations was excellent with the MPI implementation, as one might expect given the substantial investment in these over the last 20 years. While SHMEM is primarily about one-sided communication, SHMEM applications may rely upon collective operations, particularly in certain mathematical procedures (e.g. Krylov solvers), where dot products are essential.

In the future, we will generalize our intranode optimizations to work in the general case where PEs are spread across multiple nodes, i.e. shared-memory access will be used within a node while MPI operations will be used between nodes. This usage is permitted using the unified memory model of MPI-3 that can be supported on cache-coherent systems. OSHMPI currently lacks intranode optimizations for atomics and strided operations but it is straightforward to add these, the former using compiler intrinsics instead of inline assembly to maintain a high degree of portability. MPI datatypes will be used to support SHMEM operations on more than 2^{31} elements, which may be required on 64-bit systems with abundant memory.

Acknowledgment

This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. We are thankful to all the anonymous reviewers who helped us improve the paper.

References

1. R. Bariuso and Allan Knies. Shmem user's guide, 1994.
2. Brian Barrett, Torsten Hoefler, James Dinan, Rajeev Thakur, Pavan Balaji, Bill Gropp, and Keith Douglas Underwood. Remote memory access programming in MPI-3. Preprint, Argonne National Laboratory, Apr 2013.
3. Brian W. Barrett, Ron Brightwell, Scott Hemmert, Kevin Pedretti, Kyle Wheeler, Keith Underwood, Rolf Riesen, Arthur B. Maccabe, , and Trammell Hudson. The Portals 4.0 message passing interface. (SAND2013-3181), April 2013.
4. Brian W. Barrett, Ron Brigthwell, K. Scott Hemmert, Kevin Pedretti, Kyle Wheeler, and Keith D. Underwood. Enhanced support for OpenSHMEM communication in Portals. *High-Performance Interconnects, Symposium on*, 0:61–69, 2011.
5. Dan Bonachea. GASNet specification, v1.1. Technical Report UCB/CSD-02-1207, U.C. Berkeley, 2002.
6. Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *Int. J. High Perform. Comput. Netw.*, 1:91–99, August 2004.
7. Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing OpenSHMEM: SHMEM for the

PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, page 2. ACM, 2010.

8. J. Dinan, P. Balaji, J. R. Hammond, Sriram Krishnamoorthy, and V. Tipparaju. Supporting the Global Arrays PGAS model using MPI one-sided communication. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, may 2012.
9. James Dinan, Sriram Krishnamoorthy, Pavan Balaji, Jeff Hammond, Manoj Krishnan, Vinod Tipparaju, and Abhinav Vishnu. Noncollective communicator creation in MPI. *Recent Advances in the Message Passing Interface*, pages 282–291, 2011.
10. Karl Feind. Shared memory access (shmem) routines. In *Cray User Group, CUG'05*, 1995.
11. Dmitri Gribenko and Alexander Zinenko. Enabling Clang to statically check MPI type safety. In *International Conferences on High Performance Computing (HPC-UA)*, October 2012.
12. HP. HP Alphaserver SC 40. `http://h18002.www1.hp.com/alphaserver/archive/sc/sys_sc40_features.html`.
13. IBM. HPC Toolkit. `https://computing.llnl.gov/mpi/klepacki.pdf`, 2004.
14. J. Jose, K. Kandalla, Miao Luo, and D.K. Panda. Supporting hybrid MPI and OpenSHMEM over InfiniBand: Design and performance evaluation. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 219–228, 2012.
15. Jithin Jose, Krishna Kandalla, Miao Luo, and Dhabaleswar K Panda. Supporting hybrid mpi and OpenSHMEM over InfiniBand: Design and performance evaluation. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 219–228. IEEE, 2012.
16. Jithin Jose, Krishna Kandalla, Jie Zhang, Sreeram Potluri, and Dhabaleswar K. Panda. Optimizing collective communication in OpenSHMEM. October 2013.
17. MPI Forum. MPI: A message-passing interface standard. Version 3.0., November 2012.
18. Jarek Nieplocha and Bryan Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *Parallel and Distributed Processing*, pages 533–546, London, UK, 1999. Springer-Verlag.
19. Krzysztof Parzyszek, Jarek Nieplocha, and Ricky A Kendall. A generalized portable SHMEM library for high performance computing. Technical report, Ames Lab., Ames, IA (US), 2000.
20. Stephen W Poole, Oscar Hernandez, Jeffery A Kuehn, Galen M Shipman, Anthony Curtis, and Karl Feind. OpenSHMEM - toward a unified RMA model. In *Encyclopedia of Parallel Computing*, pages 1379–1391. Springer, 2011.
21. Quadrics. Quadrics/SHMEM programming manual, 2001.
22. Gilad Shainer, Todd Wilde, Pak Lui, Tong Liu, Michael Kagan, Mike Dubman, Yiftah Shahar, Richard Graham, Pavel Shamis, and Steve Poole. The co-design architecture for exascale systems, a novel approach for scalable designs. *Computer Science-Research and Development*, pages 1–7, 2013.
23. Jesper Larsson Träff. Compact and efficient implementation of the MPI group operations. pages 170–178, 2010.
24. Michael Woodacre, Derek Robb, Dean Roe, and Karl Feind. The SGI AltixTM 3000 global shared-memory architecture, 2005.
25. Changil Yoon, Vikas Aggarwal, Vrishali Hajare, Alan D George, and Max Billingsley III. GSHMEM: A portable library for lightweight, shared-memory, parallel programming. *Proceedings of Partitioned Global Address Space, Galveston, Texas*, 2011.