SHCOLL - A Standalone Implementation of OpenSHMEM-Style Collectives API

Srđan Milaković[1](✉), Zoran Budimlić[1], Howard Pritchard[2], Anthony Curtis[3], Barbara Chapman[3], and Vivek Sarkar[4]

[1] Rice University, Houston, USA
srdjan@rice.edu
[2] Los Alamos National Laboratory, Los Alamos, USA
[3] Stony Brook University, Stony Brook, USA
[4] Georgia Institute of Technology, Atlanta, Georgia

Abstract. The performance of collective operations has a large impact on overall performance in many HPC applications. Implementing multiple algorithms and selecting optimal one depending on message size and the number of processes involved in the operation is essential to achieve good performance. In this paper, we will present SHCOLL, a collective routines library that was developed on top of OpenSHMEM API point to point operations: puts, gets, atomic memory update, and memory synchronization routines. The library is designed to serve as a plug-in to OpenSHMEM implementations and will be used by the OSSS OpenSHMEM reference implementation to support OpenSHMEM collective operations. In this paper, we describe the algorithms that have been incorporated in the implementation of each OpenSHMEM API collective routine and evaluate them on a Cray XC30 system. For long messages, SHCOLL shows an improvement by up to a factor of 12 compared to the vendor's implementation. We also discuss future development of the library, as well as how it will be incorporated into the OSSS OpenSHMEM reference implementation.

1 Introduction

OpenSHMEM includes both point-to-point communication and collective operations in its specification. These collectives involve synchronization (barriers), data movement (e.g. broadcast, alltoall) and computation (reductions).

A number of platforms provide hardware support for collective operations and vendor solutions will take advantage of this. For portable solutions where such hardware support is not available, it is desirable to provide software implementations of collectives. SHCOLL is such a library for community use, providing a number of algorithms for OpenSHMEM collectives. OpenSHMEM developers, or other developers working on similar problems, can then incorporate SHCOLL into their implementations to avoid reinventing the wheel.

The rest of the paper is organized as follows: in Sect. 2 we compare SHCOLL with related work; in Sect. 3 we introduce the OpenSHMEM specification; in

© Springer Nature Switzerland AG 2019
S. Pophale et al. (Eds.): OpenSHMEM 2018, LNCS 11283, pp. 90–106, 2019.
https://doi.org/10.1007/978-3-030-04918-8_6

Sect. 4 we discuss the implementation of OpenSHMEM that this work is based on; in Sect. 5 we elaborate the different algorithms provided by the SHCOLL library; in Sect. 6 we include and discuss the experimental results; and in Sect. 7 we discuss future work and ideas.

2 Related Work

Most of the previous work focuses on collective communication for the Message Passing Interface (MPI) such as work by Thakur et al. that investigates the performance of different algorithms in MPICH [2,26]. Also, some researchers designed algorithms for specific message sizes such as Rabenseifner's algorithm for large reductions [22] or Van de Gejin's algorithm for large broadcast [6]. Awan et al. investigated design and performance of non-blocking collectives in OpenSHMEM using MVAPICH2-X [3,5]. Jose et al. optimized performance of OpenSHMEM collective operations by developing a light-weight mapping between collective operations in OpenSHMEM and MPI [17]. In this paper, we focus on optimizing OpenSHMEM collective operations using only OpenSHMEM API operations.

3 OpenSHMEM

OpenSHMEM is a specification [20] in the Partitioned Global Address Space (PGAS) family for a distributed parallel programming library that focuses on fast, low-latency, communication using Remote Direct Memory Access (RDMA) to address remote variables directly.

'SHMEM" is a family of PGAS libraries that was developed by various vendors since the early 1990s, but unfortunately drifted from each other over time with subtly different behaviors and APIs that caused portability problems. This led, at least in the C language, to unwieldy preprocessor conditional macro definitions that attempted to iron out the differences [21].

OpenSHMEM is the process that unifies these "SHMEM" libraries under a common, agreed upon and ratified, specification.

4 The OSSS-UCX OpenSHMEM Implementation

4.1 Initial Implementation with GASNet

After the OpenSHMEM specification was first drafted around 2010, the reference implementation library was developed by the University of Houston [12]. This library used GASNet [8] as its communications substrate. GASNet is a portable communications library that was initially developed for use in UPC [23] but has found use in other projects, for example Chapel [13], Legion [7], and in a runtime for Fortran CoArrays [19].

Although GASNet supports a wide range of underlying networks (e.g. Infiniband, Cray Aries, Intel OmniPath, portable MPI), some functionality required by OpenSHMEM is not exposed to the programmer. In particular, GASNet does not, as yet, expose remote atomics, nor does it allow arbitrary memory registration, which would be required to support multiple symmetric heaps with different memory kinds in the future.

4.2 New Implementation with UCX

The current reference implementation, named "OSSS-UCX" after Open Source Software Solutions, for OpenSHMEM specification 1.4 (and beyond) is based on UCX [24]. UCX is a multi-party open-source project to produce a best-of-breed communications substrate that can be used by different HPC paradigms, but predominantly MPI and PGAS libraries and languages.

OSSS-UCX uses UCX for its communications. The OpenSHMEM API maps quite naturally to UCX's upper layer, called UCP ("P" for Protocol). UCP then drops to UCT ("T" for Transport) to target individual network layers. UCX also contains UCS for Operating System services, and UCM for memory management. By targeting UCP, OSSS-UCX does not have to concern itself with network details and thus will work on any network supported by UCX.

4.3 Process Management Interface

OSSS-UCX uses PMIx [11], the Process Management Interface for Exascale, as its launch mechanism. Open-MPI and the PMIx Reference Runtime Environment (PRRTE) [4] provide a launcher with a PMIx server that coordinates the initial bootstrap of information required by UCX for RDMA and atomics. The OpenSHMEM Processing Elements (PEs) contain PMIx clients that exchange information through the server. PMIx will also be used for fault-tolerance.

OSSS-UCX also incorporates some third-party software to, for example, manage symmetric memory allocations.

5 Collective Operations Algorithms

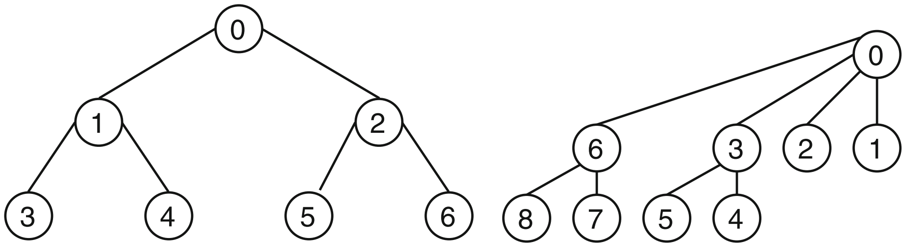
As mentioned earlier in Sect. 2, most of the previous work focuses on collective operations for Message Passing Interface (MPI). For the purpose of this paper, we have implemented all collective operations in OpenSHMEM. In MPI when a process is supposed to receive data, it must call a receive method. However, in OpenSHMEM that is not required because OpenSHMEM supports one-sided remote memory access. When a receive method returns in MPI, there is a guarantee that the data is delivered. Since OpenSHMEM does not have an analogous method, it is necessary to notify the remote node that the data transfer has completed. To ensure the transfer order between the data and the notification, it is required to call `shmem_fence` in between. Also, Cray SHMEM supports extensions to OpenSHMEM API that combine data transfer with data delivery notification (`shmemx_putmem_signal`) so in addition to an approach that uses shmem_fence, we also used the Cray SHMEM extensions to improve the performance. Additionally, for remote memory accesses in OpenSHMEM, there is no need to calculate the addresses for remote writes in the user code because remotely accessible memory locations have symmetric addresses.

5.1 Barrier

The barrier is a synchronization collective routine that registers the arrival of a PE at the barrier and blocks the execution until all other PEs arrive at the barrier [20]. The library we implemented supports three types of barrier algorithms: linear, tree and dissemination barrier.

In linear barrier, when a PE reaches the barrier, it will increment a counter at PE 0. When the counter reaches the number of PEs, PE 0 will notify all other PEs that they can continue with execution.

For tree barrier, all PEs are organized in a tree. When a non-root PE reaches the barrier, it will wait until the value of its local counter becomes equal to the number of children. Then the PE will increment a counter at the parent. When the counter at root PE becomes equal to the number of children, the root PE will notify its children, and the children will start propagating the notification to the leaf PEs. The library supports two types of trees, k-ary and k-nomial trees (Fig. 1).



(a) Binary tree (2-ary tree) (b) Trinomial tree (3-nomial tree)

Fig. 1. Examples of k-ary and k-nomial trees

Dissemination barrier belongs to the category of butterly barrier algorithms, and it has $\lceil \log p \rceil$ rounds [1]. In each round r $(0 \leq r < \lceil \log p \rceil)$, PE i will signal PE $(i + 2^k)$ % p and wait for a signal from PE $(i - 2^k)$ % p. After getting a signal in $\lceil \log p \rceil$ round, the PE can continue with execution.

5.2 Collect, Fcollect

Collect and fcollect are collective routines that concatenate blocks of data from multiple PEs to an array in every PE. Fcollect requires that the size of each block must be the same whereas block size for collect may vary [20]. For collect, we support linear, recursive doubling, ring and Bruck algorithm. In addition to the algorithms we support for collect, we support neighbor exchange algorithm for fcollect.

In the linear algorithm, each PE issues a put operation to all other PEs in a loop. After the data from a single PE is transferred, the PE increments a counter on all other PEs or calls a barrier depending on the number of PEs.

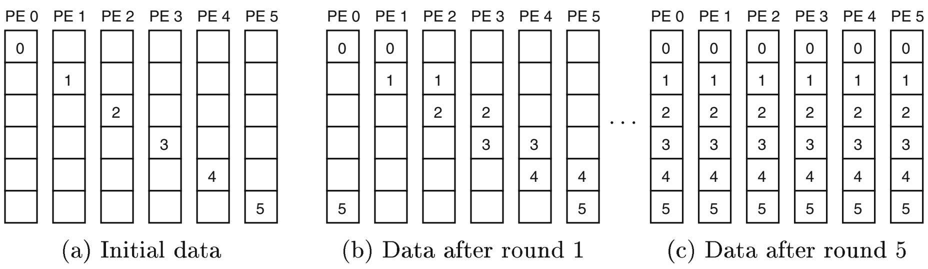
The ring algorithm (Fig. 2) requires $p-1$ rounds. In round r $(0 \le r < p-1)$, PE i sends block $(i-r)$ % p to PE $(i+1)$ % p and receives block $(i-r-1)$ % p from PE $(i-1)$ % p.

The neighbor exchange algorithm (Fig. 3) works only if p is even and it requires $\frac{p}{2}$ rounds. In the first round, PE i sends its block to i XOR 1. In odd rounds, even PEs send 2 blocks that were received in the previous round to PE $(i-1)$ % p and odd PEs send the blocks to PE $(i+1)$ % p. In even rounds, even PEs send 2 blocks that were received in the previous round to PE $(i+1)$ % p and odd PEs send the blocks to PE $(i-1)$ % p.

The recursive doubling algorithm (Fig. 4) works if p is a power of two and it requires $\log p$ rounds. In round r $(0 \le r < \log p)$, PE i sends the data that was received in the previous rounds to PE i XOR 2^r.

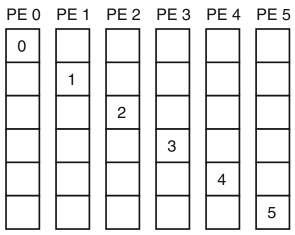
Like the recursive doubling algorithm, the Bruck algorithm [9] (Fig. 5) also requires $\lceil \log p \rceil$ rounds but, unlike the recursive doubling algorithm, it works even if p is not a power of two. First, each PEs copies its block to the beginning of its buffer. Then, in round r $(0 \le r < \lceil \log p \rceil)$, PE i sends 2^r blocks from the beginning of the buffer to PE $(i-2^r)$ % p. If the number of PEs is not a power of two, in the last round each PE will send $p-2^r$ blocks to its peer PE. After the data is exchanged between PEs, each PE is required to rotate the destination array by i blocks to the left. Additionally, we have implemented a variation of the Bruck algorithm that does not require rotation at the end but some of the messages are split into two parts because the data that should be sent is not contiguous.

For the collect algorithms, PEs do not have information about the offset for their blocks, so calculating the exclusive prefix sum before exchanging the data is necessary. Additionally, Bruck algorithm requires the total size of all elements so that value is broadcasted before the data exchange.

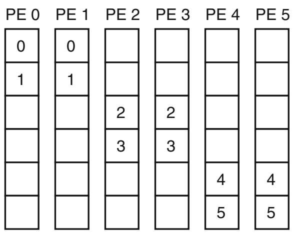


(a) Initial data (b) Data after round 1 (c) Data after round 5

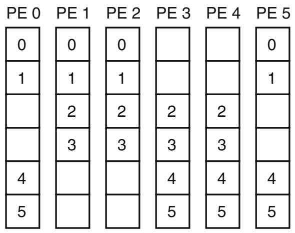
Fig. 2. Ring collect



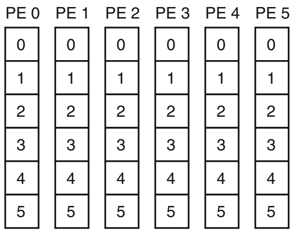
(a) Initial data



(b) Data after round 1



(c) Data after round 2



(d) Data after round 3

Fig. 3. Neighbor exchange collect

5.3 Broadcast

The broadcast is a collective routine that sends data from the root PE to all other PEs in the active set. The library supports three types of the broadcast algorithms: linear, tree, and Van de Geijn's algorithm.

In the linear algorithm, all PEs (except root PE) will call the get method to retrieve the data from the root. To ensure that the root has the data, a barrier is called before and after calling the get method.

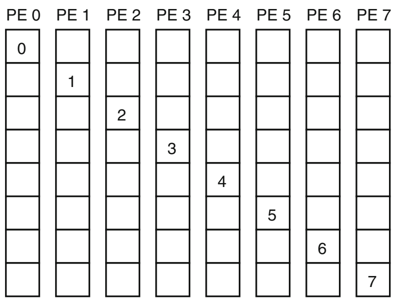
For tree broadcast, all PEs are organized in a tree with the PE that has the data as a root. When the root PE invokes broadcast it will send the data to its children, and the children will start propagating the data down the tree. The library supports two types of trees: k-ary and binomial trees (Fig. 1).

Van de Geijn's algorithm [6] is good for large messages. First, the data is scattered across all PEs and then it is concatenated using a method analogous to collect. For scattering, we use binomial scatter, and for collect, we use the ring algorithm (Fig. 2).

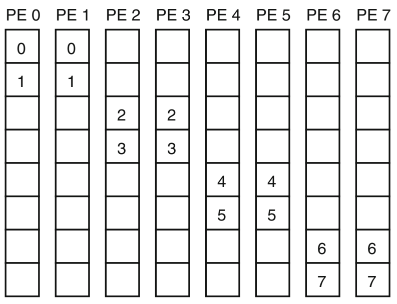
5.4 Alltoall, Alltoalls

Alltoall and alltoalls are collective routines in which each PE exchanges a fixed amount of data with all other PEs in the active set. The data that is exchanged in alltoall has to be contiguous whereas the data in alltoalls can be strided [20].

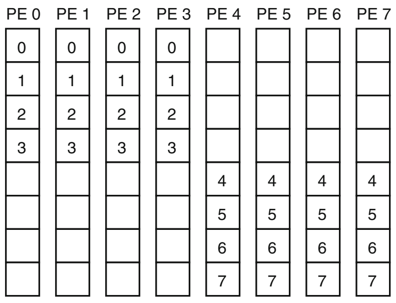
For both collectives, we support three algorithms: shift exchange, XOR pairwise exchange, and generalized pairwise exchange [25]. All three algorithms have



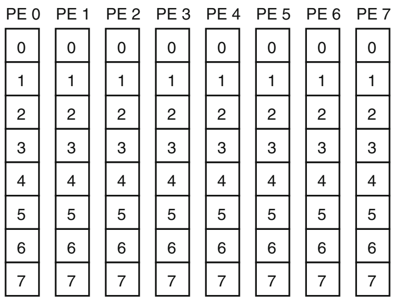
(a) Initial data



(b) Data after round 1



(c) Data after round 2



(d) Data after round 3

Fig. 4. Recursive doubling collect

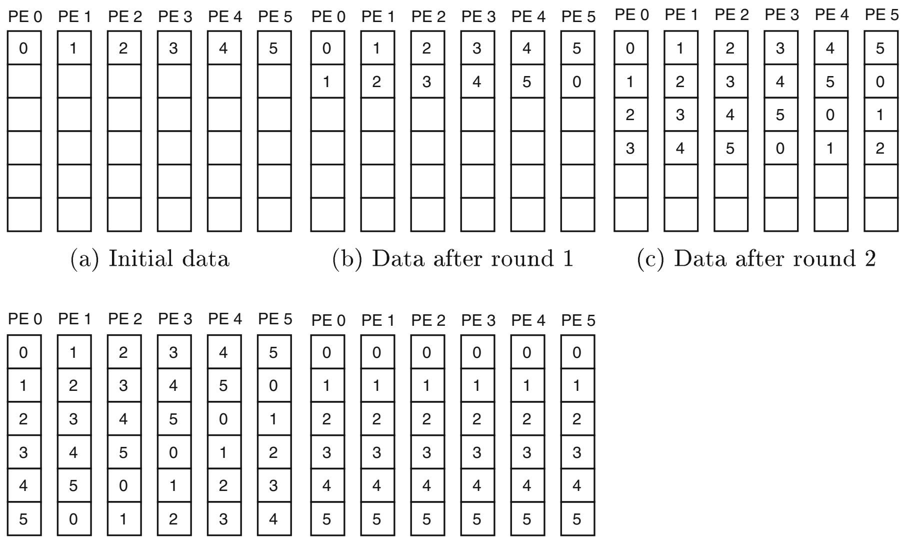
p rounds and in each round, a put is issued to a different PE (put to self is done using `memcpy`). However, each algorithm issues put in a different order.

Shift exchange is the simplest algorithm among the algorithms we implemented. In round r $(1 \leq r \leq p)$, PE i will send its data to PE $(i + r)$ % p. This algorithm tries to avoid the bottleneck that would happen if all PEs were writing to PE r in round r.

XOR pairwise exchange works only when the number of PEs is a power of 2. In each round of this algorithm, each PE has a partner PE and communicates exclusively with its partner PE. (it sends the data to the partner and it receives the data from the partner). In round r $(1 \leq r \leq p)$, the id of the partner PE for PE i is calculated as i XOR r.

Like XOR pairwise exchange, each PE has a partner in each round of generalized pairwise exchange. However, generalized pairwise exchange does not require the number of processes to be a power of 2. The problem of finding a partner can be solved by solving the edge-coloring problem in a complete graph. The complete algorithm can be found in [25].

After the data from a single PE is transferred, the PE increments a counter on all other PEs, or call a barrier depending on the number of PEs.


(a) Initial data (b) Data after round 1 (c) Data after round 2

(d) Data after round 3 (e) Data after rotation

Fig. 5. Bruck collect

In alltoall implementations, we used non-blocking put. However, the OpenSHMEM API [20] does not support non-blocking strided put so we implemented a naive version of non-blocking strided put which we use it in the alltoalls implementations.

5.5 Reductions

Reductions are a set of collective routines that perform associative arithmetic and logical operations across arrays on PEs from the active set [20]. The library we implemented supports a recursive doubling algorithm and Rabenseifner's algorithm.

For both algorithms, we have to first choose the greatest subset of PEs $\mathcal{P}'$ such that the number of nodes p' in the subset is a power of two. After choosing the subset, we assign a unique node from the subset a partner node, which is not in the subset, and then we perform reduction between the partner nodes.

Rabenseifner suggests that the new subset should be a union of even PEs less than $2 * (p - p')$ and PEs greater or equal to $2 * (p - p')$. If we have multiple PEs per node and use Rabenseifner's approach for choosing the subset $\mathcal{P}'$, the PEs from the subset will not be balanced across nodes. Consequently, the nodes that have more PEs than others will have to perform more reduce operations and they will have to exchange more data. To solve this problem, we use a different approach. First we assign a new id to each PE, which is calculated as $\text{id}_{new} = \lfloor \text{id}_{old} \times \frac{p'}{p} \rfloor$. Since $\frac{p}{2} < p' \leq p$, at most two PEs can have the same new id. The nodes that have the same new id are partners and the node that has a has lower id_{old} belongs to the $\mathcal{P}'$ subset.

After the data between partners is reduced, the recursive doubling algorithm uses the new ids. The communication pattern for recursive doubling reduce is the same as the communication pattern for recursive doubling collect. However, instead of concatenating the arrays, we perform reduction operations across arrays. In round r $(0 \leq r < \log p')$, PE i sends the array that was reduced in the previous rounds to PE i XOR 2^r and after receiving data from PE i XOR 2^r, PE i performs local reduction. After $\log p'$ rounds, PEs in the subset $\mathcal{P}'$ will have the reduced array, and the nodes from the subset will send the reduced data to their partners.

Like recursive doubling, Rabenseifner's algorithm also uses the new ids after the reduction between partners. The idea behind Rabenseifner's algorithm is similar to the idea behind Van de Geijn's algorithm from Sect. 5.3. First, a reduce scatter operation is performed so that each PE has a part of the final array, and then the array is concatenated using collect. Similar to recursive doubling, after the data is concatenated, only PEs from the subset $\mathcal{P}$ have the reduced array, so the nodes from the subset will send the reduced data to their partners.

6 Results

In this section, we present a performance evaluation of SHCOLL's collective functions and compare their performance against the equivalent OpenSHMEM functions provided by Cray SHMEM. Note, SHCOLL uses Cray's OpenSHMEM put and memory sychronization methods for data transfers Sect. 5.

6.1 Evaluation Platform and Software

All experimental results presented were collected on the NERSC Edison machine. Edison is a Cray⋆ XC30 with 2×12-core Intel® Xeon® Processors E5-2695 v2 and 64 GB DDR3 in each node. The system was running Cray's CLE 6.0.UP05 operating system. Cray's Intel Programming Environment 6.0.4 was used to compile SHCOLL and its performance tests. Cray's OpenSHMEM 7.6.2 was used for linking against SHCOLL and to obtain Cray OpenSHMEM performance results. Jobs were submitted to Edison using SLURM's *contiguous* option to try and get closely packed sets of nodes. The SLURM nodelists for the jobs indicated that the allocations obtained were generally closely packed, taking into account locations of service and I/O nodes within cabinets.

The OSU OpenSHMEM benchmark tests [3] were initially used for comparing the performance of SHCOLL against the vendor's OpenSHMEM implementation. However, there were limitations in the OSU tests which reduced their usefulness for this evaluation: they don't include all OpenSHMEM collectives, iteration count and transfers are not easily configurable, and they don't check for correctness, even during the warm-up phase. For thes reason, we decided to write tests specifically for this evaluation.

6.2 Barrier

In Fig. 6, timings for SHCOLL's `shcoll_barrier` are compared to Cray's `shmem_barrier` for 1 to 512 nodes using 1 and 24 PEs/node. The plot reports time per iteration in milliseconds. The vendor's `shmem_barrier` performs significantly better at all node counts both for the 1 PE and 24 PEs per node runs. This is expected as Cray OpenSHMEM makes use DMAPP API collective calls [10,14] to access the Aries collective engine (CE) [16] for the inter-node stage of the barrier operation. The significant jump from 256 to 512 nodes can be attributed to the fact that at 512 nodes, the job spans more than a single electrical group of the Cray XC30.

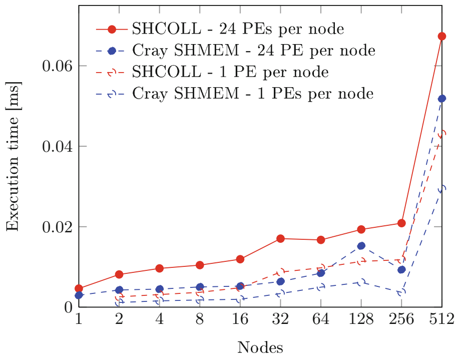


Fig. 6. Barrier

6.3 Broadcast

Figure 7 compares time for broadcast operations for 4, 1KB, 1MB, and 256MB byte transfer sizes for SHCOLL's `shcoll_bcast32` and the vendor's `shmem_broadcast32` functions for 1 to 512 nodes, and 1 and 24 PEs per node. The plot gives times for a broadcast operation plus a subsequent `shmem_barrier_all`, to ensure we are timing the full transfer to all participating PEs, and not just the time spent in the broadcast operation by the root PE. For the 4-byte broadcast, SHCOLL uses the *k-nomial* algorithm. The Cray OpenSHMEM broadcast significantly outperforms the SHCOLL implementation, particularly for the 1 PE per node case. This indicates the Cray implementation may be employing the Aries CE to do the broadcast by using its reduction engine with only the root PE supplying a non-zero value. For the 1 KB broadcast, the *k-nomial* algorithm gives optimal performance as well. Using Cray's put-with-signal operation gives best performance for the *k-nomial* algorithm. This helps particularly for 256 and 512 nodes, where SHCOLL performs significantly better than the Cray implementation. The 24 PEs per node timings

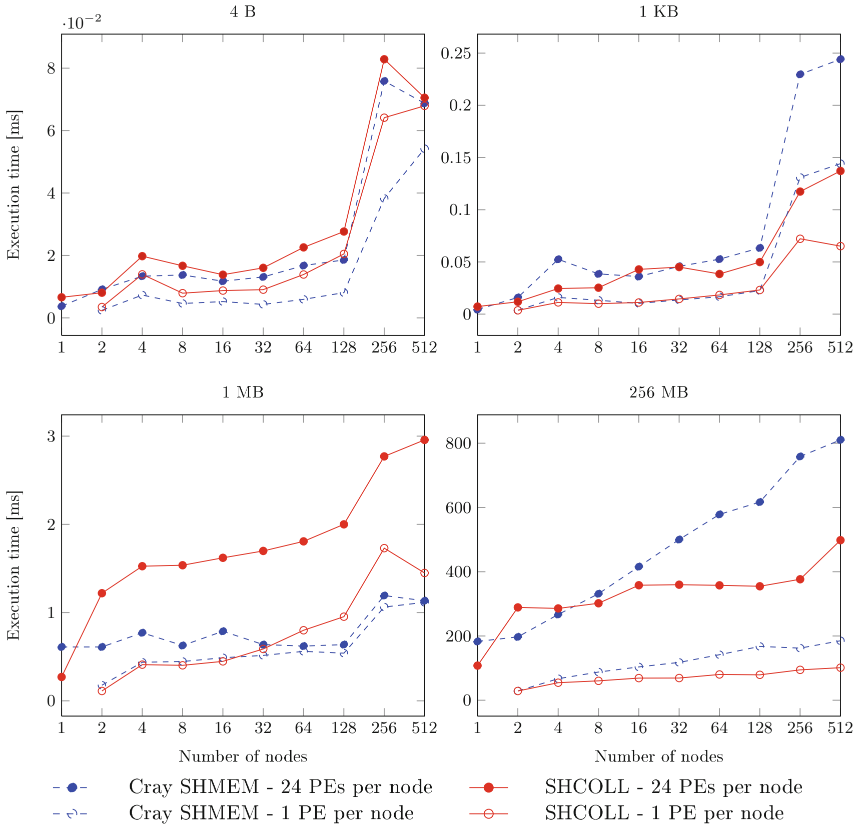


Fig. 7. Broadcast

show a similar performance difference between the Cray and SHCOLL broadcast implementations. For the 1 MB transfer size, the *binomial tree* algorithm gave the best results for SHCOLL, although the Cray implementation shows better performance. For the 256 MB broadcast, SHCOLL uses the *Van de Geijn'st* algorithm. The results for the Cray implementation are similar to those obtained using the *binomial tree* method. The *Van de Geijn's* gives better performance for both the 1 and 24 PE cases compared to the vendor's implementation.

The `SHMEM_USE_OPT_MASSIVE_BCAST` environment variable was used to check for the best timings using Cray OpenSHMEM. At some PE counts and transfer sizes setting the environment variable helped, in which case timings were taken with it set.

6.4 Reduce

Figure 8 shows times for OpenSHMEM `shmem_double_sum_to_all` and the SHCOLL equivalent for 8 and 2 KB reductions. Timings include a preceding

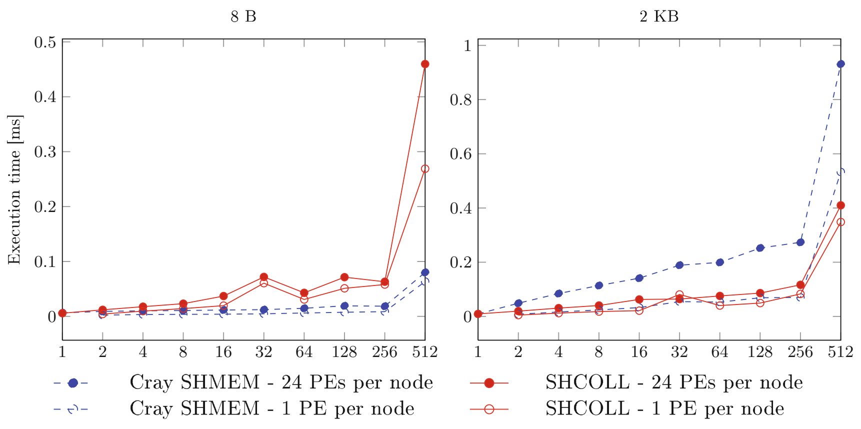


Fig. 8. Reduction

`shmem_barrier_all` to ensure the `pSync` array is properly armed. The results for the 8 byte reduction show that Cray's implementation is making use of the Aries CE, consequently performing significantly better than SHCOLL's *recursive doubling* approach. For 2 KB reductions SHCOLL uses *recursive doubling* for 1 PE per node (power of two), and the *Rabenseifner* algorithm for 24 PEs per node. This algorithm gives better results for all node counts, leading to superior performance for SHCOLL in this case. Note the Aries CE can't be efficiently used for these size reductions. Performance is similar to Cray when using recursive doubling.

The Cray OpenSHMEM `SHMEM_USE_LARGE_OPT_REDUCE` variable was set when it gave better performance.

6.5 Fcollect

Figure 9 presents timing results for `shmem_fcollect32` and its SHCOLL equivalent for 4 and 16 KB per PE operations. As with the reduction tests, a `shmem_barrier_all` is included in the *fcollect* timing loop. For the 4 bytes per PE operation, SHCOLL employees the *Bruck* algorithm and makes use of Cray's put-with-signal extension to OpenSHMEM [18]. The SHCOLL implementation at this transfer size gives comparable performance to the Cray implementation for 1 PE per node up to 128 nodes, and better performance beyond. For 24 PEs per node, the SHCOLL approach yields much better performance. The significant difference at 24 PEs per node verses 1 PE per node hints that the Cray algorithm may be doing something suboptimal - perhaps leading to network congestion - particularly as the performance deteriorates significantly at higher node counts.

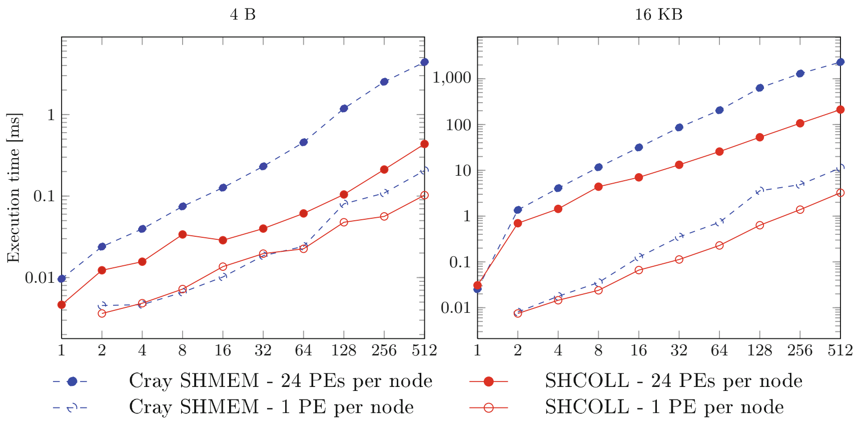


Fig. 9. Fcollect

For 16 KB size transfers and 24 PEs per node, the *ring* algorithm gives the best results for SHCOLL, likely due to the pipelining effect offered by this algorithm. Using this algorithm, SHCOLL performs much better than the vendor implementation, especially at 16 and higher node counts. For 16 KB per PE operations and 1 PE per node, we use the `linear` method up to 256 nodes and the `Bruck` algorithm for 512 nodes.

6.6 Collect

Results for timing of `shmem_collect32` and `shcoll_collect32` are presented in Fig. 10. The *collect* method involves more inter-PE data exchange as each PE supplies its contribution to the transfer, and the implementation must assemble this information in order to do the actual data exchange correctly. For 4 byte per PE (in these tests each PE contributes the same amount of data), SHCOLL uses the *recursive doubling* algorithm and Cray's put-with-signal feature for 1 PE per node, and *linear* for low node counts and *Bruck* for higher node counts. The *Bruck* algorithm yields significantly better results than the method used by the vendor, as shown by the 24 PE/node results at nodes counts of 16 and higher.

For the 16 KB, the *linear* method was optimal up to 32 nodes, with the *Bruck* algorithm performing better for higher node counts. Both algorithms give superior performance to the approach used in the vendor implementation.

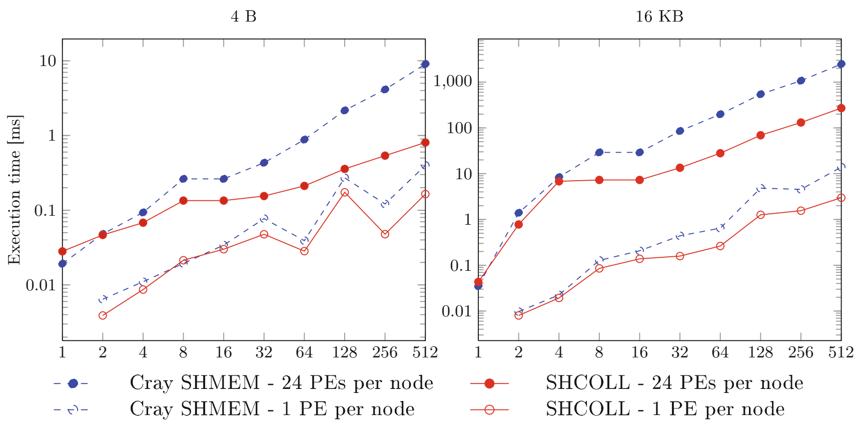


Fig. 10. Collect

6.7 Alltoall

Figure 11 compares performance of Cray OpenSHMEM `shmem_alltoall32` against that of SHCOLL's equivalent `shcoll_alltoall32` function. The *color-pairwise* exchange method generally performed best for all transfer sizes. At low node or PE counts, the Cray put-with-signal approach works well, but a barrier based synchronization is employed for higher numbers of processes. The algorithm could be more efficient if the underlying network (and the OpenSHMEM API), supported a put-with-counter mechanism [15]. The vendor implementation [18] modestly outperforms the SHCOLL implementation suggesting that the Cray implementation is similar to that used by SHCOLL.

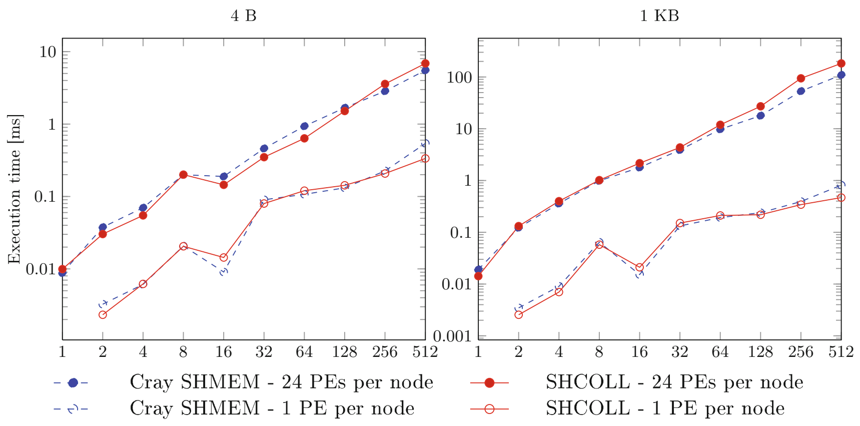


Fig. 11. Alltoall

6.8 Alltoalls

Figure 12 shows timing results for Cray's OpenSHMEM implementation's shmem_alltoalls32 against SHCOLL's shcoll_alltoalls32. As with the other experiments, the timed loop includes a shmem_barrier_all to keep the pSync array properly armed. Results for 4 byte and 128 byte per PE contributions are shown. For the single PE per node tests, it was found that the *xor-pairwise exchange* method gave good results for both transfer sizes. For the 24 PEs/node case, the *shift_exchange* method with barrier synchronization works best for the 4 byte exchange, while for the 128 byte transfer size, the *color-pairwise exchange* was superior. SHCOLL gives significantly better performance for the 4 byte per PE operation at all node counts for both 1 and 24 PEs per node, while showing modestly better results for the 128 byte per PE case.

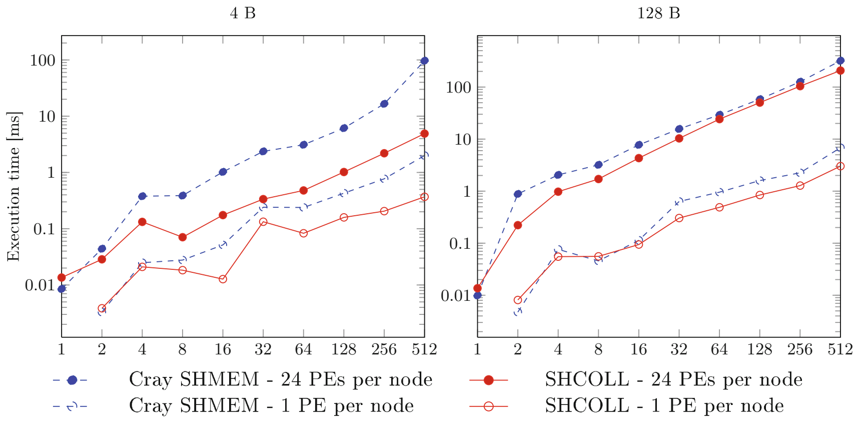


Fig. 12. Alltoalls

The performance of the SHCOLL algorithms was also helped by the use of what is effectively a non-blocking implicit shmem_iputX_nbi function:

```
void shmem_iput32_nbi(void* dest, const void* source, ptrdiff_t dst,
                      ptrdiff_t sst, size_t nelems, int pe) {
    uint32_t* dest_ptr = (uint32_t*)dest;
    const uint32_t* source_ptr = (const uint32_t*)source;
    for (int i = 0; i < nelems; i++) {
        shmem_put32_nbi(dest_ptr, source_ptr, 1, pe);
        dest_ptr += dst; source_ptr += sst;
    }
}
```

This approach was used for the data movement part of the shcoll_alltoalls32 implementation.

7 Conclusion and Future Work

In this paper we have shown that implementing multiple algorithms and selecting the optimal one depending on message size and the number of processes involved in the operation is essential to achieving good performance. Currently, the optimal algorithm for both transfer size and the number of PEs involved in the collective is chosen manually. In future we plan to develop methods to better automate the selection of the optimal algorithm for a particular message size and number of processes. Also, to improve the performance for flat OpenSHMEM applications that use collective operations, we plan add topology aware collectives using PMIx [11]. We further plan to integrate SHCOLL into a future OSSS OpenSHMEM collective plugin framework.

Acknowledgments. This research was funded in part by the United States Department of Defense, and was supported by resources at Los Alamos National Laboratory. This publication has been approved for public, unlimited distribution by Los Alamos National Laboratory, with document number LA-UR-18-27273.

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

References

1. Introduction to barrier algorithms. https://6xq.net/barrier-intro/
2. MPICH. https://www.mpich.org
3. MVAPICH2-X. http://mvapich.cse.ohio-state.edu/
4. PMIx Reference RunTime Environment. https://github.com/pmix/prrte
5. Awan, A.A., Hamidouche, K., Chu, C.H., Panda, D.: A case for non-blocking collectives in OpenSHMEM: design, implementation, and performance evaluation using MVAPICH2-X. In: Gorentla Venkata, M., Shamis, P., Imam, N., Lopez, M.G. (eds.) OpenSHMEM 2014. LNCS, vol. 9397, pp. 69–86. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26428-8_5
6. Barnett, M., Shuler, L., van De Geijn, R., Gupta, S., Payne, D.G., Watts, J.: Interprocessor collective communication library (intercom). In: Proceedings of the Scalable High-Performance Computing Conference, pp. 357–364. IEEE (1994)
7. Bauer, M.E.: Legion: programming distributed heterogeneous architectures with logical regions (2014)
8. Bonachea, D.: GASNet specification, v1.1. Technical report, Computer Science Department, University of California, Berkeley (2002)
9. Bruck, J., Ho, C.T., Kipnis, S., Upfal, E., Weathersby, D.: Efficient algorithms for all-to-all communications in multiport message-passing systems. IEEE Trans. Parallel Distrib. Syst. **8**(11), 1143–1156 (1997)
10. ten Buggencate, M., Roweth, D.: DMAPP: an API for one-sided programming models on baker systems. In: Proceedings of Cray User Group (2010)
11. Castain, R.H., Solt, D., Hursey, J., Bouteiller, A.: Pmix: process management for exascale environments. In: Proceedings of the 24th European MPI Users' Group Meeting, EuroMPI 2017, pp. 14:1–14:10. ACM, New York (2017). http://doi.acm.org/10.1145/3127024.3127027
12. Chapman, B., et al.: Introducing OpenSHMEM: SHMEM for the PGAS community. In: Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model, PGAS 2010, pp. 2:1–2:3. ACM, New York (2010). http://doi.acm.org/10.1145/2020373.2020375

13. Cray, Inc.: Chapel Language Specification. Technical report, Cray, Inc. (2010)
14. Cray Inc.: Using the GNI and DMAPP APIs (2011)
15. Dinan, J., Cole, C., Jost, G., Smith, S., Underwood, K., Wisniewski, R.W.: Reducing synchronization overhead through bundled communication. In: Poole, S., Hernandez, O., Shamis, P. (eds.) OpenSHMEM 2014. LNCS, vol. 8356, pp. 163–177. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-05215-1_12
16. Faanes, G., et al.: Cray cascade: a scalable HPC system based on a dragonfly network. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC 2012), November 2012
17. Jose, J., Kandalla, K., Zhang, J., Potluri, S., Panda, D.: Optimizing collective communication in openshmem. In: 7th International Conference on PGAS Programming Models, p. 185 (2013)
18. Knaak, D., Namashivayam, N.: Proposing OpenSHMEM extensions towards a future for hybrid programming and heterogeneous computing. In: Gorentla Venkata, M., Shamis, P., Imam, N., Lopez, M.G. (eds.) OpenSHMEM 2014. LNCS, vol. 9397, pp. 53–68. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26428-8_4
19. Namashivayam, N., Eachempati, D., Khaldi, D., Chapman, B.M.: OpenSHMEM as a portable communication layer for PGAS models: a case study with coarray fortran. In: 2015 IEEE International Conference on Cluster Computing, CLUSTER 2015, Chicago, IL, USA, 8–11 September 2015, pp. 438–447 (2015). http://dx.doi.org/10.1109/CLUSTER.2015.66
20. OpenSHMEM Specification Committee: OpenSHMEM Specification. http://www.openshmem.org/site/Specification
21. Poole, S.W., Hernandez, O., Kuehn, J.A., Shipman, G.M., Curtis, A., Feind, K.: OpenSHMEM - toward a unified RMA model. In: Padua, D. (ed.) Encyclopedia of Parallel Computing. Springer, Boston (2011). https://doi.org/10.1007/978-0-387-09766-4_490
22. Rolf Rabenseifner: A new optimized MPI reduce algorithm. https://fs.hlrs.de/projects/par/mpi//myreduce.html
23. Chauvin, S., Saha, P., Cantonnet, F., Annareddy, S., El-Ghazawi, T.: UPC Manual (2003)
24. Shamis, P., et al.: UCX: an open source framework for HPC network APIS and beyond. In: 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects, pp. 40–43, August 2015
25. Tam, A., Wang, C.L.: Efficient scheduling of complete exchange on clusters. In: 13th International Conference on Parallel and Distributed Computing Systems (PDCS 2000), Las Vegas, vol. 4 (2000)
26. Thakur, R., Rabenseifner, R., Gropp, W.: Optimization of collective communication operations in MPICH. Int. J. High Perform. Comput. Appl. **19**(1), 49–66 (2005)