Application-Level Optimization of On-Node Communication in OpenSHMEM

Md. Wasi-ur- Rahman[1(✉)], David Ozog[2(✉)], and James Dinan[2(✉)]

[1] Intel Corporation, Austin, USA
md.rahman@intel.com
[2] Intel Corporation, Boston, USA
{david.m.ozog,james.dinan}@intel.com

Abstract. The OpenSHMEM community is actively exploring threading support extensions to the OpenSHMEM communication interfaces. Among the motivations for these extensions are the optimization of on-node data sharing and reduction of memory pressure, both of which are problems that hybrid programming has successfully addressed in other programming models. We observe that OpenSHMEM already supports inter-process shared memory for processes within the same node. In this work, we assess the viability of this existing API to address the on-node optimization problem, which is of growing importance. We identify multiple on-node optimizations that are already possible with the existing interface, propose a layered library that extends the functionality of these interfaces, and measure performance improvement when using these techniques.

1 Introduction

High Performance Computing (HPC) system nodes continue to trend toward increasingly powerful and increasingly parallel processors, including many-core processors and accelerators. As a result, HPC application developers are looking beyond conventional parallel programming systems toward hybrid approaches that combine a communication library, such as MPI or OpenSHMEM, with an on-node programming model, such as OpenMP*. The resulting combination enables the application developer to tune for system-level effects, while also efficiently utilizing the capabilities and resources provided by the node-level architecture.

A primary feature that drives the success of Partitioned Global Address Space (PGAS) programming models is their ability to remotely access the memory of other processing elements (PEs) without explicit participation from the target PE. While PGAS programming models, such as OpenSHMEM, conveniently provide such one-sided remote access to the memory of *any* processing element (PE), communication with PEs that share *local* memory may suffer from unnecessary performance overheads. At its core, OpenSHMEM is a data copying library; thus, even when PEs are in the same node, communication via OpenSHMEM can result in the creation of multiple copies of the same data within a shared memory domain. Additional overheads may also arise from various sources: a complex software stack that is capable of supporting general

© Springer International Publishing AG 2018
M. Gorentla Venkata et al. (Eds.): OpenSHMEM 2017, LNCS 10679, pp. 99–113, 2018.
https://doi.org/10.1007/978-3-319-73814-7_7

remote memory access (RMA), the sheer memory replication cost of single process multiple data (SPMD) programming, and the synchronization mechanisms associated with large-scale programming models. As a result, the OpenSHMEM community is actively investigating library extensions to better support node-level optimization, including methods for integrating threading awareness within the library [3,8].

While hybrid programming is of interest to many programmers, maintainers of existing applications may prefer a more evolutionary approach to tuning on-node data sharing. We observe that OpenSHMEM provides a seldom-used function that allows the programmer to query a direct pointer to the remotely accessible memory of another PE within the same shared memory domain. While this functionality is supported by a number of OpenSHMEM implementations, it is challenging to use in its current form because the current OpenSHMEM interfaces don't expose the locality information needed by programmers to effectively utilize this capability.

In this work, we investigate the challenges and opportunities of the OpenSHMEM pointer query API. We develop a portable library, called `shnode`, that fills the gaps in the current interface and improves the usability of this interface. We believe the results of this work will highlight an evolutionary path for node-level tuning of applications. In addition, we hope that it will provide valuable insights to ongoing efforts to extend OpenSHMEM with new features such as hybrid programming and on-node based teams support. We evaluate the performance impact of our approach using several benchmarks, including a large-scale parallel sorting benchmark and observe that this approach to optimization of on-node communication can yield significant performance improvements.

The rest of the paper is organized as follows. Section 2 presents background on OpenSHMEM and the pointer query API. We highlight some of the key challenges for this work in Sect. 3. Design and implementation details of the `shnode` library are presented in Sect. 4. We present the details of our experimental evaluation in Sect. 5. Section 6 highlights some of the existing works in the literature and we conclude in Sect. 8.

2 Background

This work investigates shared memory optimizations in the context of OpenSHMEM [15], an HPC communication library that provides a partitioned global address space (PGAS) data model through one-sided read, write, and atomic update routines. In this section, we describe the typical execution models for OpenSHMEM programs and outline various techniques for exploiting on-node memory locality.

One very common use case for OpenSHMEM programs running on HPC clusters is to allocate one or more processing elements (PE) per compute node. Each compute node typically consists of multiple processing units and/or individual cores, so it may be advantageous to assign *multiple* PEs to each compute node to exploit the available parallelism. In the OpenSHMEM programming model,

each of these PEs designates a memory region for storing symmetric heap and local variable data.

While the designated memory regions of each PE are *remotely accessible* by any other PE in the application, there is also the possibility that data may *reside locally* with respect to other on-node PEs. However, there is no guarantee that this data locality is exploited by the OpenSHMEM implementation. Even if an implementation does optimize for on-node PE locality, it still may be difficult for an application developer to optimize *outside* of the OpenSHMEM API. Multi-threading within a PE's address space can accomplish on-node parallelism with good data locality, but it is typically not straightforward to accomplish this across the address space of multiple PE's, despite the fact that their memory regions may reside on the same node.

The OpenSHMEM API includes a routine that enables on-node addressing, which may be useful for optimizing applications for memory locality. This routine, called `shmem_ptr`, returns the specified pointer to a symmetric buffer on the specific PE. Its function signature is:

```
void *shmem_ptr(const void *dest, int pe);
```

where `dest` is the local pointer to the symmetric data buffer, and `pe` is the PE id of the desired process. This routine returns a pointer to the "remote" symmetric data object in the local PE's address space. If a program has `dest` value for all symmetric regions of interest, and knowledge of which PEs are node-local, then shared memory optimizations are possible at the application level. Despite the availability of this function in the API, it is the opinion of these authors that it is underutilized across the OpenSHMEM programs. In Sect. 3 we argue why this underutilization may exist, and Sect. 4 presents how this routine is used in constructing a more user-friendly and general interface for achieving shared memory optimization in OpenSHMEM programs.

3 Challenges and Opportunities

The `shmem_ptr` routine enables shared memory accesses and optimizations in OpenSHMEM programs. However, there are challenges to using this routine in practice. For example, if an application wants to know which PEs are locally resident, then `shmem_ptr` gives only very limited information. This routine returns a null pointer whenever the input PE value is off-node. This requires looping over *all* PEs and storing the non-null IDs into a local structure. One goal of this paper is to abstract this procedure into a simpler interface that creates *teams* of processes that group together node-local PE subsets. Such an interface would enable applications to do memory operations within their local teams, which eliminates the overhead of the software stack involved in remote communication.

In addition to the locality knowledge that node-local teams provide, there are other requirements for useful shared memory programming. For instance, consider an algorithm that involves local computation/communication, followed by a collective operation. Instead of having all PEs participate in the collective,

the application may only require one PE per *team* to participate. We call this PE a *leader* in our design. Leader election algorithms constitute a well-known topic in distributed systems [2], in part because of their dependence on network topologies and system hierarchy/architecture. Leader election implementations are particularly important in OpenSHMEM, especially for checkpointing applications [1,9]. A goal of our API is to abstract leader selection and to enable customizable *multiple-leader* assignment on a per-node basis.

Perhaps the primary challenge with shared-memory programming lies in developing algorithms that effectively exploit data locality. Often *re-development* is necessary because existing legacy applications rely on algorithms that do not adequately account for locality. Performance improvement for this software is difficult without thoroughly considering data-layout, communication and synchronization strategies, and load balancing. Communication avoiding algorithms show great promise [7], and need to be incorporated to best exploit locality at the node-level. In the following section, we introduce the `shnode` API based on the `shmem_ptr` routine to bridge these gaps, enabling application developers to design algorithms that better exploit data locality.

4 Design and Implementation of `shnode`

In this section, we present the design and implementation of our proposed layered library for on-node data sharing, called *shnode*. The purpose of this library is to provide several APIs to application developers with which the application can benefit through avoidance of on-node communication.

As discussed in Sect. 2, we utilize the built-in routine, `shmem_ptr`, to design `shnode`. Since `shmem_ptr` returns the specific memory address for a symmetric data object on an on-node remote PE, it can provide the opportunity for the application developers to store these pointers for direct load and store operations as opposed to invoking remote memory access (e.g. `shmem_put`). To facilitate this, we propose the APIs listed in Listing 1.1.

To utilize the `shnode` library, application developers should follow the usual semantics of initialization and termination of `shnode` functionalities through the OpenSHMEM-like APIs, `shnode_init` and `shnode_finalize`. In the future, these functionalities can be incorporated and invoked from the OpenSHMEM initialization and finalize routines based on the input to an environment flag set by the user. After the initialization, the user needs to create the per-node team. Based on the remote data pointers returned by the `shmem_ptr` routine, `shnode` creates team of PEs on each node. These data references will be stored so that subsequent remote memory operations can be substituted with direct load and stores to the memory location residing in the on-node PE's symmetric heap. The API `shnode_create_team` is responsible for creating the team on each node consisting of all those PEs for which a non-NULL value is obtained through `shmem_ptr`. Figure 1(a) presents the team formation for a two node cluster running with 8 PEs per node. To add more data objects, a user can simply use `shmem_add_data` for the subsequent shared memory objects. We

Listing 1.1. Proposed fundamental APIs for shnode.

```
/* initialization */
int shnode_init();

/* team creation based on a symmetric data object */
int shnode_create_team(void *data);

/* addition of other symmetric data objects */
int shnode_add_data(void *data);

/* check to see whether the remote pe is a team member */
int shnode_is_team_member(int rem_pe);

/* retrieval of the memory address of an on-node PE */
void *shnode_get_member_remote_addr(int rem_pe, void *data);

/* check to see whether self is the leader of the team */
int shnode_am_team_leader();

/* destroy */ int shnode_finalize();
```

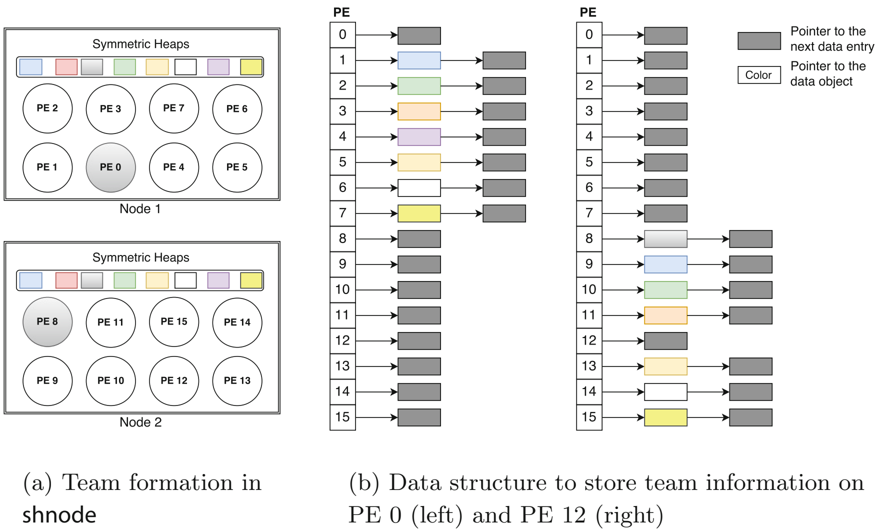
assign the lowest rank PE as the team leader for each node. The purpose of the leader is further explained in Sect. 4.2.

To store the team information on each PE, we design a simple data structure mapping each PE to a list of the data object references returned by shmem_ptr. Figure 1(b) illustrates this for the team presented in Fig. 1(a). To track a specific data object, we maintain another list that maps the corresponding data object to the location it is stored in the PE-mapped data structure. This is helpful for fast retrieval of the requested reference when multiple data objects are stored in the data structure. The shnode_create_team operation is invoked only once at the beginning of the application execution; thus, does not incur significant overheads to the execution time of the application.

After successful team creation, the user can utilize the shnode_get_member_remote_addr to retrieve the data reference stored in the shnode team table. Using the remote location address, the user can perform direct load and store, replacing remote memory operations.

4.1 Better Overlapping Between Communication And Computation

Since shnode provides the memory addresses for symmetric data objects on on-node PEs, it provides the opportunity to the application developer to replace the remote memory operations with the direct memory operations, such as memcpy. Although this eliminates overhead caused by the remote operations, it still has the drawbacks of invoking memory transfers. One of the alternatives for the application developers is to perform swapping of the pointers instead of copying the content. In this way, users can eliminate any software overhead caused by large memory to memory data transfers. However, in many applications, this approach might require a significant effort to re-write the application to maintain correctness.



(a) Team formation in shnode

(b) Data structure to store team information on PE 0 (left) and PE 12 (right)

Fig. 1. Design and implementation details of shnode

The other alternative is to customize the remote memory calls in such a way so that the intra-node data transfers are invoked separately from the inter-node ones; thereby optimizing the overlap between communication and computation. Scheduling the intra-node memory operations at the end will ensure better overlap between computation and long-delayed inter-node memory operations. Application developers can utilize the team information from shnode to refine the communication operations in this way.

4.2 Designing shnode Collective Helper Routines

With the assignment of a team leader PE per node, we can also optimize the collective communication by designing helper routines for each collective operations. Figure 2 presents one such use case. Our current implementations of these routines assume a power-of-two number of process elements per node and the process launcher launches each of the PEs sequentially from the first node to the last node in the cluster.

As shown in Fig. 2, we can re-design the collective operations considering the hierarchy of nodes achieved from the shnode library. Each collective operation can divide its tasks in three sub-tasks. As an example, we explain here a division of tasks for a reduction operation. In the first sub-task, all the PEs communicate with the on-node leader so that the leaders in each node gets the reduced values from all the PEs on that node. In the second sub-task, a reduce operation is strided over only the leaders across the nodes. This significantly reduces the communication overheads. The stride is calculated from the process per node which is assumed to be a power-of-two value. Finally, all the leaders pass the globally reduced value to the on-node PEs to complete the operation.

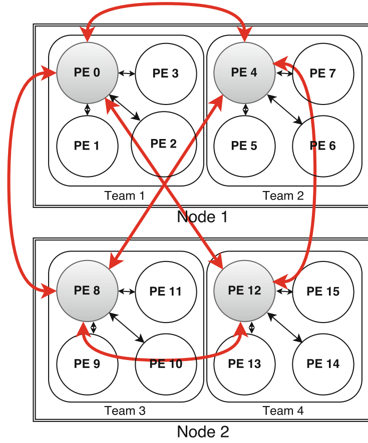


Fig. 2. Multiple leader based collective communication design for `shnode`

For large collective operations with a higher number of PEs per node, a single leader per node might not yield the maximum benefits possible. In such cases, we create sub-teams within teams and assign the lowest rank of each sub-team as the leader for that team. This reduces the overhead for each leader and thus a balance between the number of teams and the number of PEs per teams is achieved. As an example, Fig. 2 presents two teams per node with different leaders instead of the default one leader. We present evaluations for different numbers of leaders in Sect. 5.3.

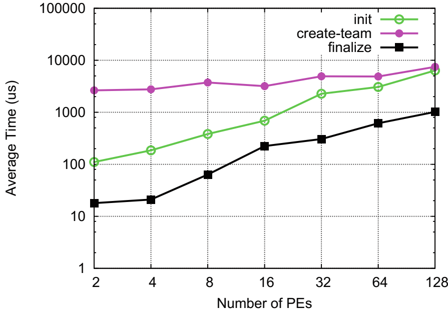
5 Performance Evaluation

In this section, we present our evaluation of different benchmarks and applications utilizing the `shnode` library and compare them with the default approach. We present our evaluations in three different categories: (1) Evaluating `shnode` with a micro-benchmark, (2) Performance improvement in collectives, (3) Evaluation of applications.

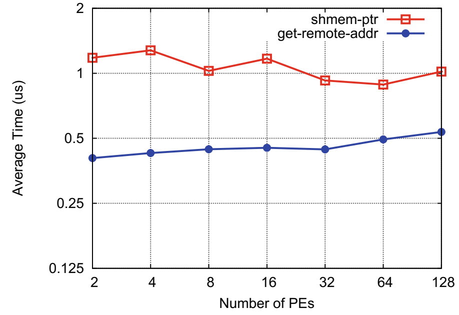
5.1 Experimental Setup

For our evaluation, we have used the NERSC Cori supercomputer, which is the 6th fastest supercomputer on the TOP500 [18] list, published in June, 2017. It is a Cray* XC40 system with 2,388 Intel® Xeon® E5-2698 v3 (Haswell) processor nodes at 2.3 GHz and 9,688 Intel® Xeon Phi™ 7250 (Knights Landing, KNL) processor nodes with 68 cores per node at 1.4 GHz. Each of the KNL nodes have 96 GB of DDR4 memory. All the compute nodes run a light-weight kernel based on the SuSE* Linux* Enterprise Server distribution.

Throughout our experiments, we have used the KNL nodes in Cori. We have implemented `shnode` on top of Cray* SHMEM v7.5.5 and used the same for our evaluations and comparisons.



(a) Profiling basic APIs in shnode



(b) Comparing with shmem_ptr

Fig. 3. Profiling analysis for `shnode` APIs

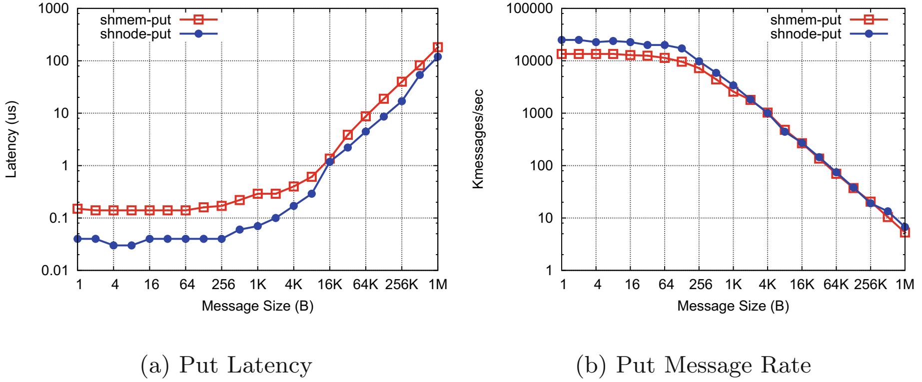
5.2 Evaluating Shnode with Micro-benchmark

In this section, we evaluate `shnode` with two different micro-benchmarks. First, we write a micro-benchmark to do a profiling analysis for four of the fundamental APIs that we have proposed in Sect. 4 - `shnode_init`, `shnode_create_team`, `shnode_get_member_remote_addr`, and `shnode_finalize`. Out of these four APIs, application developers might need to use `shnode_get_member_remote_addr` multiple times throughout the application execution, whereas, the remaining APIs would only be invoked once during the runtime of the application.

We conduct the profiling analysis in two KNL nodes with a varying number of PEs per node (from 1 to 64). We measure the average execution time for each of these APIs across all PEs. As shown in the Fig. 3(a), with 128 PEs, the initialization, team creation, and finalize routines take only about 0.1 s which does not incur significant overheads. In Fig. 3(b), we compare the `shnode_get_member_remote_addr` with the default `shmem_ptr` routine. We can see that with our implementation, we can reduce the query operation cost by 50% on average across different number of PEs. Also, this routine scales well because of the design choices for the data structures in `shnode`.

We also evaluate the basic put and get performance using micro-benchmark and present these results in Figs. 4 and 5. We modify the OSU micro-benchmarks [17] for `shmem_put` and `shmem_get` to incorporate the `shnode` APIs and compare the modified put and get performances with the default ones. We conduct these experiments on a single node with two PEs.

As shown in Fig. 4(a), `shnode` can perform 3–4.6x faster compared to `shmem_put` for small message sizes (up to 2K). For larger message sizes, the benefit reduces to 1.5–2.35x. Similar benefits are observed for `shnode` based get compared to `shmem_get`, as shown in Fig. 5. We also measure the message rate for put and present these results in Fig. 4(b). Here, an average benefit of 1.35x is observed for `shnode` put compared to the `shmem_put`. Since the `shnode` implementation of put and get performs a direct memory copy to/from the remote



(a) Put Latency (b) Put Message Rate

Fig. 4. Performance comparison between SHMEM and `shnode` put operations

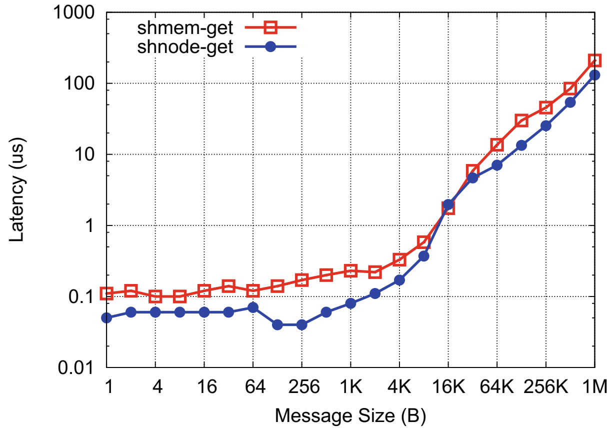


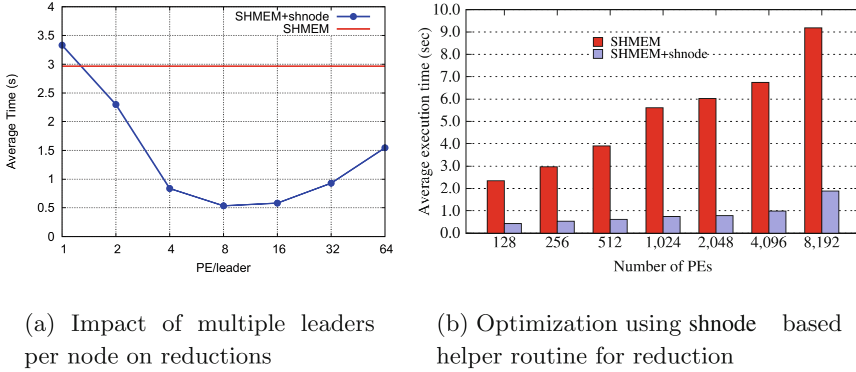
Fig. 5. Performance comparison between SHMEM and `shnode` get operations

address, this approach obtains significant performance benefits compared to the default ones.

5.3 Performance Improvement in Collective Routines

In this section, we present the performance comparisons between default collective routines and the `shnode` based helper routines. We implement our `shnode` based helper collective routines for reduction and collect and present the results here. We also evaluate the impact of multiple leaders per node on each of these collectives.

Figure 6 presents the corresponding experiments on the reduction, specifically a sum-based reduction for integer data types (`int_sum_to_all`). First, we analyze the impact of multiple leaders per node on this reduction and present this result in Fig. 6(a). We conduct this experiment on four KNL nodes where



(a) Impact of multiple leaders per node on reductions

(b) Optimization using shnode based helper routine for reduction

Fig. 6. Performance improvement potentials for reductions with `shnode`

we vary the number of PEs per leader from 1 to 64. We allocate a 10 MB buffer to use as the data source for the reduction. Experimental results presented in this section are averaged over 10 iterations.

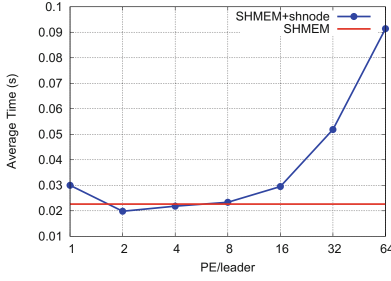
As shown in Fig. 6(a), we achieve the most optimal performance for `int_sum_to_all` with 8 PEs per leader. Thus, with 64 PEs running on each node, we observe the most optimal result with 8 leaders per node, where each of them are responsible for communicating with the 7 other PEs. We also observe that the default SHMEM implementation for reduction could not take advantage of such hierarchical work distribution provided by `shnode`. With this optimum value for the number of leaders, we conduct another experiment where we increase the total number of PEs from 128 (2 nodes) to 8 K (128 nodes). From the evaluation results presented in Fig. 6(b), we see that with the `shnode` implementation on top of SHMEM, we can achieve up to 4.87x benefit compared to the default SHMEM approach for `int_sum_to_all`.

We implement the same for `fcollect` collective routine and present the results in Fig. 7. We use a similar setup to the reduction experiment.

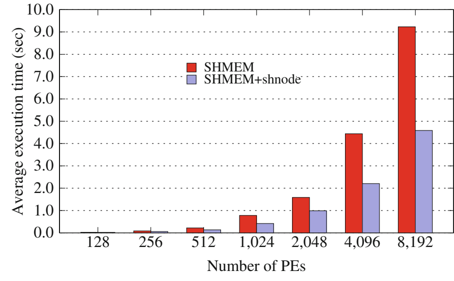
Unlike `int_sum_to_all`, we can see in Fig. 7(a) that for `fcollect`, the optimum performance is achieved with 2 PEs per leader (32 leaders per node for 64 PEs in a node). We also see that the default SHMEM implementation for `fcollect` performs better compared to the `shnode` implementation with more PEs per leader. We conduct this experiment on two KNL nodes with 128 total PEs. We also conduct a strong scale experiment for `fcollect` similar to the `int_sum_to_all`. For 128 nodes running 8 K PEs, we observe that the `shnode` implementation out-performs the default SHMEM implementation by 2x.

5.4 Evaluation of Applications

In this section, we evaluate an application, Integer Sort [11] (ISx) to highlight the performance improvements achievable using `shnode`. ISx represents a class of the bucket sort algorithms which perform an all-to-all communication

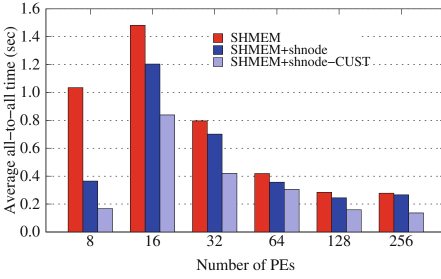


(a) Impact of multiple leaders per node on fcollect

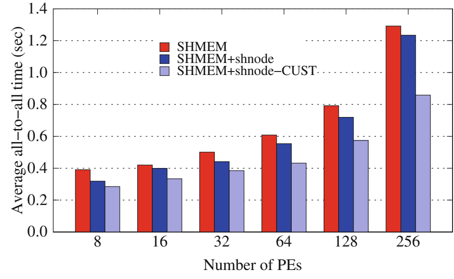


(b) Optimization using shnode based helper routine for fcollect

Fig. 7. Performance improvement potentials for fcollect with `shnode`



(a) Strong scaling



(b) Weak scaling

Fig. 8. Performance improvement for ISx with SHNODE library and customized communication scheduling

pattern. In this evaluation, we present two different implementations of ISx utilizing `shnode`, one with no additional changes in the communication pattern (presented as SHMEM+shnode) and the other with customized communication scheduling, where the node-local transfers are separated out and invoked only at the end of the execution (presented as SHMEM+shnode-CUST). We conduct both strong and weak scale experiments for ISx on 4 nodes with varying number of PEs. Figure 8 presents these results.

For strong scale experiment, we use the number of items to sort equal to 1.5 billion and vary the number of PEs from 8 to 256. As shown in Fig. 8(a), with the customized communication pattern, we can achieve 2x benefits compared to the default implementation over SHMEM with 256 PEs. Since ISx overlaps communication with computations, the `shnode` implementation without the customized communication pattern does not observe much benefit (around 5%) compared to the default implementation. For weak scaling experiments as presented in Fig. 8(b), we observe a performance benefits of 1.5x for 256 PEs. In this experiment, we fix the number of items per PE to 32 M.

Similar to the strong scale experiments, the shnode implementation without customization in communication pattern achieves only 5% benefit over the default implementation.

6 Related Work

Namashivayam et al. [14] explore how shmem_ptr can be used on the Intel® Xeon Phi™ processor to better exploit shared memory and enable vectorization opportunities. This work focuses on single-node performance in the native mode of the Xeon Phi, in which applications run directly on the many-core device. The authors report substantial performance improvements in the latency and bandwidth of one-sided operations, across several reduction algorithms, and in the NAS Integer Sort (IS) and Scalar Penta-diagonal (SP) solver parallel benchmarks. Our paper extends this work by defining a coherent interface that enables applications to exploit shared memory outside of the OpenSHMEM API.

The shnode interface for gathering on-node groups of PEs is similar to the idea of OpenSHMEM *teams* and *spaces*, which was introduced by Welch et al. [19] and also proposed in [13]. The APIs for discovering local PEs [4] and building a team in Cray-SHMEM [5] also provide methods to find out local PEs in a predefined team. The shnode interface provides an easy way to store the pointers that can be used later to access the symmetric data objects. Another challenge in the design of shnode is that leaders must be described using the current OpenSHMEM collectives active set notation, which places significant (e.g. power of two stride) restrictions on which PEs can participate in a given collective. The choice of multiple leaders presented in this paper provides more flexibility to utilize shnode team interfaces with additional performance benefits.

There is also analogous work within the Message Passing Interface (MPI) that reflects our interface for shared memory-oriented programming. Hoefler et al. [12] first introduced the (perhaps initially surprising) notion of doing hybrid parallel programming of MPI with *itself* via the MPI+MPI paradigm. This work extends the MPI one-sided interface to include shared memory windows and associated communicators to enable interprocess communication via MPI. Our work in OpenSHMEM similarly enables on-node interprocess communication via the shnode interface, with a relatively simple API built from the shmem_ptr routine. Other work in the PGAS community further builds off the capabilities of shared memory in MPI-3 [10,21].

7 Future Work

Our measurements from Sect. 5 show very promising performance improvements when using the shnode API, yet there remain considerable possibilities for future work. For instance, the shnode concept could (and should) be implemented within the OpenSHMEM software layer for all viable routines, such as collectives and the RMA functions. We present our shnode implementations

outside the OpenSHMEM layer as a proof-of-concept for what should be implemented *within* an OpenSHMEM library. We have so far only implemented a handle of routines from the OpenSHMEM specification (namely, `fcollect`, `int_sum_to_all`, `broadcast`, and `put/get`), but many other routines are also compatible.

We believe that `shnode` will primarily benefit application developers who require processing data across PEs that are grouped into shared-memory teams. Our results from Fig. 8 show a notable performance improvement for a real-world application, ISx. Other applications may also greatly benefit from `shnode`, but may require restructuring to achieve communication avoidance at the compute node-level. For instance, we observe that the OpenSHMEM stencil algorithm from the Parallel Research Kernels suite [20] may need to be restructured to reduce synchrony between global iterations. This may be possible, for example, by *over-decomposing* the grid domain to avoid starvation due to synchronous iterations.

Due to `shnode`'s performance improvement of reductions (shown in Fig. 6), we believe MapReduce calculations [6] will also greatly benefit because of their heavy use of reduction collectives and the inter-process communication involved in intermediate shuffling operations. For instance, the MapReduce-MPI library [16] centers around several calls to an integer sum reduction, which is the same procedure measured in Fig. 6 above. Future work will quantify the performance gain from reducing and shuffling in shared memory using `shnode`.

8 Conclusion

This paper has introduced an interface for OpenSHMEM that alleviates the challenges involved with programming in shared-memory. Our implementation, `shnode`, supports the formation of node-local *teams* within which applications can easily do shared memory operations. We present an API for creating these teams, as well as for nominating a leader process or multiple leader processes. Overall, `shnode` shows very good performance improvement across RMA microbenchmarks, OpenSHMEM collectives, and the ISx application. Our performance results show that the number of leaders has a substantial impact on performance, depending on the communication algorithm being deployed. Future work for this research will involve shifting `shnode` capabilities to within the OpenSHMEM software layer, implementing the other variants of collectives and RMA operations, and exploring how to restructure existing applications to better exploit shared memory.

* Other names and brands may be claimed as the property of others.
Intel and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries. Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully 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. Arya, K., Garg, R., Polyakov, A.Y., Cooperman, G.: Design and implementation for checkpointing of distributed resources using process-level virtualization. In: 2016 IEEE International Conference on Cluster Computing (CLUSTER), pp. 402–412, September 2016
2. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations, and Advanced Topics, vol. 19. Wiley, New York (2004)
3. ten Bruggencate, M., Roweth, D., Oyanagi, S.: Thread-safe SHMEM extensions. In: Poole, S., Hernandez, O., Shamis, P. (eds.) OpenSHMEM 2014. LNCS, vol. 8356, pp. 178–185. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-05215-1_13
4. Cray: shmem_local_ptr. http://docs.cray.com/man/xe_libsmam/72/cat3/shmem_local_ptr.3.html
5. Cray: shmem_team_translate_pe. http://docs.cray.com/man/xe_libsmam/72/cat3/shmem_team_translate_pe.3.html
6. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
7. Demmel, J.: Communication-avoiding algorithms for linear algebra and beyond. In: IPDPS, p. 585 (2013)
8. Dinan, J., Flajslik, M.: Contexts: a mechanism for high throughput communication in OpenSHMEM. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, pp. 10:1–10:9. ACM, New York (2014). http://doi.acm.org/10.1145/2676870.2676872
9. Garg, R., Vienne, J., Cooperman, G.: System-level transparent checkpointing for OpenSHMEM. In: Gorentla Venkata, M., Imam, N., Pophale, S., Mintz, T.M. (eds.) OpenSHMEM 2016. LNCS, vol. 10007, pp. 52–65. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50995-2_4
10. Hammond, J.R., Ghosh, S., Chapman, B.M.: Implementing OpenSHMEM using MPI-3 one-sided communication. In: Poole, S., Hernandez, O., Shamis, P. (eds.) OpenSHMEM 2014. LNCS, vol. 8356, pp. 44–58. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-05215-1_4
11. Hanebutte, U., Hemstad, J.: ISx: a scalable integer sort for co-design in the exascale era. In: 9th International Conference on Partitioned Global Address Space Programming Models, pp. 102–104, September 2015
12. Hoefler, T., Dinan, J., Buntinas, D., Balaji, P., Barrett, B., Brightwell, R., Gropp, W., Kale, V., Thakur, R.: MPI + MPI: a new hybrid approach to parallel programming with MPI plus shared memory. Computing **95**(12), 1121–1136 (2013). http://dx.doi.org/10.1007/s00607-013-0324-2
13. 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
14. Namashivayam, N., Ghosh, S., Khaldi, D., Eachempati, D., Chapman, B.: Native mode-based optimizations of remote memory accesses in OpenSHMEM for Intel Xeon Phi. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, pp. 12:1–12:11, PGAS 2014. ACM, New York (2014). http://doi.acm.org/10.1145/2676870.2676881

15. OpenSHMEM Application Programming Interface, Version 1.3, February 2016. http://www.openshmem.org
16. Plimpton, S.J., Devine, K.D.: MapReduce in MPI for large-scale graph algorithms. Parallel Comput. **37**(9), 610–632 (2011). http://dx.doi.org/10.1016/j.parco.2011.02.004
17. The Ohio State University: OSU Microbenchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/
18. Top500 Supercomputing System. http://www.top500.org
19. Welch, A., Pophale, S., Shamis, P., Hernandez, O., Poole, S., Chapman, B.: Extending the OpenSHMEM memory model to support user-defined spaces. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, PGAS 2014, pp. 11:1–11:10. ACM, New York (2014). http://doi.acm.org/10.1145/2676870.2676884
20. Van der Wijngaart, R.F., Kayi, A., Hammond, J.R., Jost, G., St. John, T., Sridharan, S., Mattson, T.G., Abercrombie, J., Nelson, J.: Comparing runtime systems with exascale ambitions using the parallel research Kernels. In: Kunkel, J.M., Balaji, P., Dongarra, J. (eds.) ISC High Performance 2016. LNCS, vol. 9697, pp. 321–339. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41321-1_17
21. Zhou, H., Idrees, K., Gracia, J.: Leveraging MPI-3 shared-memory extensions for efficient PGAS runtime systems. In: Träff, J.L., Hunold, S., Versaci, F. (eds.) Euro-Par 2015. LNCS, vol. 9233, pp. 373–384. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48096-0_29