From MPI to OpenSHMEM: Porting LAMMPS

Chunyan Tang[1], Aurelien Bouteiller[1(✉)], Thomas Herault[1], Manjunath Gorentla Venkata[2], and George Bosilca[1]

[1] Innovative Computing Laboratory, University of Tennessee, Knoxville, USA
{bouteill,bosilca,herault}@icl.utk.edu, ctang7@vols.utk.edu
[2] Oak Ridge National Laboratory, Oak Ridge, USA
manjugv@ornl.gov

Abstract. This work details the opportunities and challenges of porting a Petascale, MPI-based application —LAMMPS— to OpenSHMEM. We investigate the major programming challenges stemming from the differences in communication semantics, address space organization, and synchronization operations between the two programming models. This work provides several approaches to solve those challenges for representative communication patterns in LAMMPS, e.g., by considering group synchronization, peer's buffer status tracking, and unpacked direct transfer of scattered data. The performance of LAMMPS is evaluated on the Titan HPC system at ORNL. The OpenSHMEM implementations are compared with MPI versions in terms of both strong and weak scaling. The results outline that OpenSHMEM provides a rich semantic to implement scalable scientific applications. In addition, the experiments demonstrate that OpenSHMEM can compete with, and often improve on, the optimized MPI implementation.

1 Introduction

OpenSHMEM [12] is an emerging partitioned global address space (PGAS) library interface specification that provides interfaces for one-sided and collective communication, synchronization, and atomic operations. The one-sided communication operations do not require the active participation of the target process when receiving or exposing data, freeing the target process to work on other tasks while the data transfer is ongoing. It also supports some collective communication patterns such as synchronizations, broadcast, collection, and reduction

M. Gorentla Venkata et al. (Eds.): OpenSHMEM 2015, LNCS 9397, pp. 121–137, 2015.
DOI: 10.1007/978-3-319-26428-8_8

operations. In addition it provides interfaces for a variety of atomic operations including both 32-bit and 64-bit operations. Overall, it provides a rich set of interfaces for implementing parallel scientific applications. OpenSHMEM implementations are expected to perform well modern high performance computing (HPC) systems. This expectation stems from the design philosophy of OpenSHMEM on providing a lightweight and high performing minimalistic set of operations, a close match between the OpenSHMEM semantic and hardware-supported native operations provided by high performance interconnects and memory subsystems. This tight integration between the hardware and the programming paradigm is expected to result in close to optimal latency and bandwidth in synthetic benchmarks.

In spite of a rich set of features and a long legacy of native support from vendors, like SGI, Cray [3], and Quadrics (no longer available), OpenSHMEM has seen a slow and limited adoption. The current situation is analogous to a freshly plowed field ready for seeding, in this case, with an initial effort to design software engineering practices that enable efficient porting of scientific simulations to this programming model. This paper explores porting LAMMPS [11], a production-quality Message Passing Interface (MPI) based scientific application, to OpenSHMEM. Due to big differences in semantic and syntax between MPI [9] and OpenSHMEM, there is no straightforward one-to-one mapping of functionality. In particular, OpenSHMEM features one-sided communication and partitioned global address space, unlike most legacy MPI applications which employ two-sided MPI communication and a private address space for each MPI process. Furthermore, MPI provides explicit controls over communication patterns (e.g., communicator division and communication based on process grouping and topology) to improve programming productivity, while OpenSHMEM does not yet have support for these fine grained controls. Hence, transforming an MPI-based program into an OpenSHMEM-based one remains a difficult, and largely unexplored exercise.

2 Related Work

Despite salient differences in key concepts, MPI-3 [4] provides advanced one-sided operations, and investigations are ongoing to understand how to port from MPI-1 to MPI-3 RMA; initial results have demonstrated significant speedup [8]. OpenSHMEM, as an open standard for all SHMEM library implementations [2], was first standardized in 2012. Despite its expected benefits, the scientific computing community is still in the initial phase of exploring the OpenSHMEM concepts, hence a limited number of works using OpenSHMEM are available. Pophale et al. [13] compared the performance and scalability of unoptimized OpenSHMEM NAS benchmarks with their MPI-1 and MPI-2 counterparts. They showed that even without optimization, the OpenSHMEM-based version of the benchmarks compares favorably with MPI-1 and MPI-2. Li et al. [7] re-designed an MPI-1 based mini-application with OpenSHMEM. They demonstrated a 17 % reduction in total execution time, compared to the MPI-based

design. Similar results have been demonstrated for the Graph500 benchmark [6], while in [5], the authors have refactored an Hadoop parallel sort into an hybrid MPI+OpenSHMEM application, demonstrating a 7x improvement over the Hadoop implementation.

The work presented in this paper also focuses on leveraging OpenSHMEM in an application. However, instead of working on a mini-app or benchmark, we tackle a realistic, large-scale HPC application. We aim at understanding the opportunities and challenges of using the OpenSHMEM programming paradigm to translate and design highly scalable scientific simulations. Our work reveals strengths and limitations in OpenSHMEM, demonstrates better performance than legacy MPI in a large-scale performance evaluation, and discusses the merits of some optimization strategies.

3 Background

This section presents the target application, LAMMPS. It also presents a summary explaining the major differences between OpenSHMEM and MPI.

3.1 LAMMPS

Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) is a classical molecular dynamics (MD) code developed by Sandia National Laboratories[1]. In essence, LAMMPS integrates Newton's equations of motion with a variety of initial and/or boundary conditions. The Newton's equations calculate the motion of collections of atoms, molecules, or macroscopic particles that interact via short- or long-range forces [11].

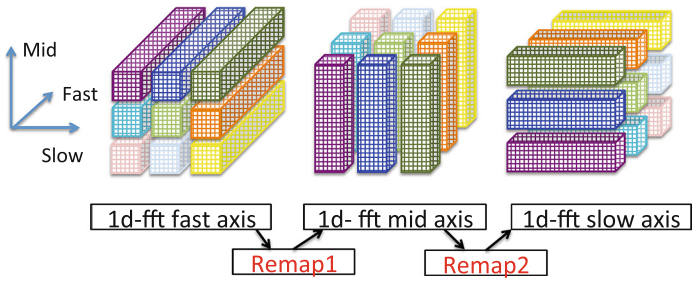


Fig. 1. Parallel 3D-FFT in 2D decomposition.

To compute the long-range Coulomb interactions, LAMMPS employs the three dimensional fast Fourier transform (3D-FFT) package, a transpose based parallel FFT developed by Steve Plimpton [10]. Transpose-based FFT performs

[1] http://lammps.sandia.gov/.

one dimension of 1D-FFT at a time, and transposes the data grid when needed. The 2D decomposition of parallel 3D-FFT is shown in Fig. 1. The three dimensions of the real space are labeled as Slow, Mid, and Fast axes, respectively. A data grid of size $l \times m \times n$ is divided between $1 \times p \times q$ processes. Hence the split size of the sub-domain is $nfast = l$, $nmid = m/p$, and $nslow = n/q$. Each process contains a sub-domain (pencil) of the data. First, each process performs the 1D-FFT along the Fast axis, where the data on this direction are local. The processes then transpose the Mid and Fast axes in order to perform the local 1D-FFT along the Mid axis. Finally, processes transpose the Mid and Slow axes and then perform a 1D-FFT along the Slow axis.

As an initial step to the analysis, we collected statistical information about MPI usage in LAMMPS with the MPI profiling tool (mpiP [14]). We used the *rhodopsin* test case (as provided by the LAMMPS test suite). The input file for this benchmark provides a basic problem size of 32 K atoms. The problem size for profiling is scaled to $8 \times 8 \times 8 \times 32$K atoms. In this strong scaling benchmark, the portion of time spent in MPI function over the total application time is 5.5 % for 256 processes, 12.6 % for 512 processes, 18.1 % for 1024 processes, 25.4 % for 2048 processes and 38.5 % for 4096 processes, respectively. Clearly, communication is a dominating factor in the overall performance, therefore reducing the communication time has the potential to improve LAMMPS performance significantly at scale.

In this test case, LAMMPS has 167 sites with calls to MPI functions. Doing a complete port of MPI-based LAMMPS to OpenSHMEM is a large undertaking which is unlikely to be completed in short order. Instead we chose to focus our efforts on the most data exchange intensive site, the `remap_3d` function, which is responsible for the transpose operations in the 3D-FFT. The `remap_3d` function employs a set of MPI point-to-point operations (`MPI_Send`, `MPI_Irecv`, `MPI_Waitany`) which concentrate the largest share of the overall transmitted MPI byte volume (32.45 % for 4096 processes). Consequently, any communication performance improvement in this function is expected to yield a sizable overall acceleration.

The resulting application is a hybrid MPI-OpenSHMEM application, where the legacy MPI application is accelerated with OpenSHMEM communication routines in performance critical sections.

3.2 OpenSHMEM Vs. MPI

To transform the above MPI-based implementation into one based on OpenSHMEM, we must account for four salient differences between the two programing models.

Address Space Differences: MPI considers a distributed address space, where each process has a private memory space, in which addresses are purely local. In contrast, OpenSHMEM features a partitioned global address space, in which each processing element (PE) has a private memory space, but that memory space is mapped to symmetric addresses, that is, if a program declares a global

array `A`, the local address of the element `A[i]` is the relevant parameter in a communication call to target `A[i]` at every PE. The main advantage of the latter model is that it enables the origin process in the operation to compute all the remote addresses for the operation, which it can then issue directly without first converting the addresses to the target's address space. This is a major feature that eases the development of applications using the one-sided communication paradigm.

Communication Semantics Differences: OpenSHMEM is based on one-sided communication. This is in contrast to the two-sided communication employed in MPI LAMMPS. Although MPI has provided one-sided communication since MPI-2 (i.e., `MPI_Get` and `MPI_Put`), two-sided communication programs dominate the common practice of using MPI. In two-sided operations, every Send operation matches a Receive operation. Each process provides a buffer, in the local address space, in which the input or output are to be accessed. In contrast, in the one-sided model, communications are issued at the origin, without any matching call at the target. Therefore, the operation progresses without an explicit action at the target, which is then considered to be *passive*, and, the origin PE must be able to perform all the necessary steps to issue and complete the communication without explicit actions from the target. In particular, the target doesn't call an operation to provide the address of the target buffer, so the origin must be able to locate this memory location (which is usually simple, thanks to the symmetric address space).

Synchronization Differences: Although MPI provides an explicit synchronization operation (*i.e.*, `MPI_Barrier`), most MPI two-sided communication operations carry an implicit synchronization semantic. As long as a process does not pass a particular memory location as a buffer in one of the MPI communication functions, the process knows it enjoys an exclusive use of that memory location, and it cannot be read from, nor written to, by any other process, until it is explicitly exposed by an MPI operation. Similarly, when the Send or Recv operations complete, the process knows that any exposure of that memory has ceased, and once again that memory can be considered exclusive. However, this is not true in the one-sided communication model, where a process can be the target of an operation at any time and the symmetric memory is exposed by default, at all times. As a consequence, when a process performs a `shmem_put`[2], it cannot rely on a corresponding `MPI_Irecv` to implicitly synchronize on the availability of the target buffer, and that synchronization must now be explicit. OpenSHMEM indeed provides a more flexible set of explicit synchronization operations, like `shmem_fence`, which guarantees ordering between two operations issued at that origin; `shmem_quiet`, which guarantees completion of all operations issued at that origin (including remote completion); and `shmem_barrier_*`, a set of group synchronizations that also complete all operations that have been posted

[2] For brevity, in this paper we use the simplified nomenclature `shmem_put`, `shmem_get`, which are not actual OpenSHMEM functions, but refer to the actual typed operations (like `shmem_double_put`, `shmem_long_get`, etc.).

prior to the barrier. Although the explicit management of synchronization can be complex, compared to the implicit synchronizations provided by the two-sided model, the cost of synchronizing can be amortized over a large number of operations, and thereby has the potential to improve the performance of applications that exchange a large number of small messages.

Collective Operations Differences: MPI proposes a fully featured interface to manage fine grain grouping of processes that participate in communication operations. The communicator concept permits establishing arbitrary groups of processes which can then issue communication operations (collective or otherwise) that span that subgroup only. In contrast, OpenSHMEM attempts to avoid having to perform message matching to reduce the latency, and consequently does not have such fine-grained control over the group of processes participating in a collective operation. OpenSHMEM features two kinds of collective operations, those that involve all PEs (like `shmem_barrier_all`), and those that involve a structured group of PEs, in which processes involved in the communication are those whose identifier is separated by a constant stride, which must be a power of 2. This lower flexibility in establishing groups of processes on which collective operations can operate can cause additional complications. Of particular relevance is the case where a collective synchronization is needed. Because the OpenSHMEM group synchronization operations are operating on groups that do not necessarily match the original grouping from the MPI application (like the *plans* in the `remap_3d` function), one is forced to either (1) use point-to-point synchronizations to construct one's own barrier that maps the PE group at hand, which could prove more expensive than an optimized barrier operation; or (2) use a global synchronization operation that spans all PEs, thereby synchronizing more processes than are necessary, with the potential outcome of harming performance with unnecessary wait time at theses processes.

4 Methodology

In this section we expose the challenges we have faced when porting the `remap_3d` LAMMPS function from MPI to OpenSHMEM. We then discuss a number of optimization strategies for the explicit management of process synchronization.

4.1 MPI Features in the `remap_3d` Function

The `remap_3d` function we focus on in this work employs non-blocking communication primitives to exchange data between MPI processes of a particular subgroup. Peers and message sizes are calculated and obtained from the remap *plan* data structure. Each *plan* corresponds to a subset of processes (in each direction) and is mapped to an MPI communicator. Three nested `for` loops enclose the `MPI_Irecv`, `MPI_Send`, and `MPI_Waitany` calls and form a many-to-many, irregular communication pattern. Of particular interest is the usage of the `MPI_Send` function in the original code, which sequentializes the multiple *sends* from a process, thereby creating a logical ordering and implicit synchronizations between groups of processes.

During the 3D-FFT transpose, the data to be transferred for each PE comes from the *in* buffer, however from non-contiguous memory locations. The MPI implementation packs the scattered data to a contiguous buffer before performing the send operation. Similarly, the receive process stores the incoming data in a temporary contiguous *scratch* buffer as well. Only after the `MPI_Irecv` is issued can the data reach the target *scratch* buffer. The received data is then unpacked to the corresponding locations in the *out* buffer.

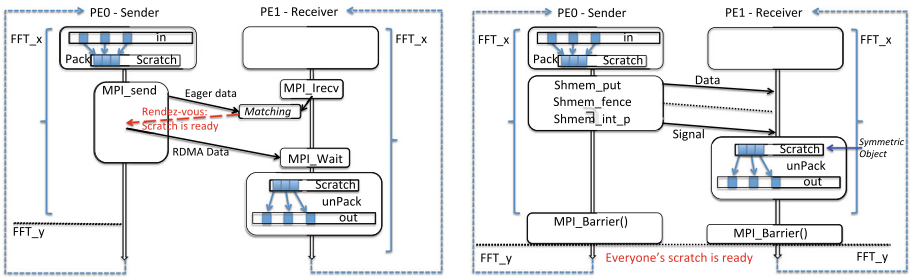


Fig. 2. Communication pattern between two processes, with MPI (left), and OpenSHMEM (right). For simplification, in the figure, a process is shown as having an exclusive role; in reality every process is both a sender and a receiver with multiple peers during each FFT iteration.

4.2 Initial Porting Effort

In this section we discuss the porting effort to create a basic OpenSHMEM code which is functionally equivalent to the original MPI `remap_3d` function. Different performance optimizations will be presented in later sections.

From two-sided to one-sided: The two-sided `MPI_Send` and `MPI_Irecv` pair can be transformed into a one-sided operation by either issuing a `shmem_put` at the sender, or a `shmem_get` at the receiver. In this test case, `shmem_put` is selected because it improves the opportunity for communication and computation overlap. With the `shmem_get` semantic, the operation completes when the output buffer has been updated (that is, the communication has completed). Essentially, `shmem_get` is a blocking operation, and there are no non-blocking variants in the current OpenSHMEM specification. Therefore, in a single threaded program, `shmem_get` leaves little opportunity for initiating concurrent computations. The `shmem_put` semantic is more relaxed: the `shmem_put` operation blocks until the input buffer can be reused at the origin. It does not block until the remote completion is achieved at the target, which means that the operation may complete while communication is still ongoing (especially on short messages). More importantly, the approach closely matches the original MPI code logic, in which computation is executed at the receiver after posting non-blocking `MPI_Irecv`

```
1  ...
2  shmem_int_max_to_all(&scratch_size, &scratch_size, 1, 0, 0, nprocs,
3                       pWrk1, pSync1);
4  plan->scratch = (FFT_DATA *) shmem_malloc(scratch_size*sizeof(FFT_DATA));
5  ...
```

Listing 1.1. Allocating the *scratch* buffers in the Partitioned Global memory space, as needed for making them targets in `shmem_put`.

```
1  ...
2  plan->remote_offset = (int *) shmem_malloc(nprocs*sizeof(int));
3  for( i = 0; i < plan->nrecv; i++)
4    shmem_int_p(&plan->remote_offset[me], plan->recv_bufloc[i],
5                plan->recv_proc[i]);
6  shmem_fence();
7  ...
```

Listing 1.2. Exchanging the offsets in the target scratch buffers; a parameter to `shmem_put` that was not required with `MPI_Send`.

operations so as to overlap the communication progress, meanwhile blocking sends are used, which have the same local completion semantic as `shmem_put` operations.

Allocating the Symmetric Scratch Buffers: In `shmem_put`, the target data buffer (the *scratch* buffer in our case) must be globally addressable. Hence, the memory space referenced by `plan->scratch` is allocated in the partitioned global address space of OpenSHMEM (line 4 in Listing 1.1). The user is responsible for ensuring that the `shmem_malloc` function is called with identical parameters at all PEs (a requirement to ensure that the allocated buffer is indeed symmetrical). As a consequence, although processes may receive a different amount of data during the irregular communication pattern, the *scratch* buffer must still be of an identical size at all PEs, which must be sufficient to accommodate the maximum size across all PEs. This concept simplifies the management of the global memory space at the expense of potentially increasing the memory consumption at some PEs. Another benign consequence is the need to determine, upon the creation of the *plan* structure, the largest memory space among all PEs (line 2 in Listing 1.1).

Irregular Communication Patterns and Target Offsets: Except for the above communication semantics difference, most of the communication parameters at the sender remain the same (*e.g.*, send buffer address and size, target peer processes). A notable exception is that, unlike in the two-sided model in which the receiver is in charge of providing the target buffer address during the `MPI_Irecv`, in the OpenSHMEM version it must be known at the origin before issuing the `shmem_put`. Although the *scratch* buffers are in the symmetric address space, and it is therefore simple to compute the start address of this buffer, the particular offset at which a process writes into the *scratch* buffer is dependent

upon the cumulative size of messages sent by processes whose PE identifier is lower. As the communication pattern is irregular, that offset is not symmetric between the different PEs and cannot be inferred at the origin independently. Ultimately, this complication stems from the different synchronization models between one-sided and two-sided operations: as the sender doesn't synchronize to establish a rendezvous with the receiver before performing the remote update, in the one-sided model, the target address must be pre-exchanged explicitly. Fortunately, we noted that, in the `remap_3d` function, the offset in the target buffer is invariant for a particular *plan*, hence we only need to transmit the offset in the target buffer to the sender once, when the `remap_3d` *plan* is initially created. An extra buffer named `plan->remote_offset` is allocated in the *plan*, and used to persistently store the offsets in the peers target buffer locations, as shown in Listing 1.2. When the same *plan* is executed multiple times, `shmem_put` operations can thereby be issued directly without further exchanges to obtain the target offset.

Signaling `shmem_put` *Operations Completion:* The original MPI program can rely on the implicit synchronization carried by the explicit exposure of the memory buffers between the `MPI_Irecv`, `MPI_Send`, and `MPI_Waitany` operations in order to track completion of communication operations. In the OpenSHMEM-based implementation, the target of a `shmem_put` is passive, and cannot determine if all the data has been delivered from the operation semantic only. To track the completion of the `shmem_put` operations, we add an extra array `plan->remote_status` which contains the per origin readiness of the target buffer. The statuses are initialized as `WAIT` before any communication happens. After the `shmem_put`, the sender uses a `shmem_int_p` to update the status at the target to `READY`. A `shmem_fence` is used between `shmem_put` and `shmem_int_p` to ensure the order of remote operations. The target PEs snoop the status memory locations to determine the completion of the `shmem_put` operations from peers.

Signaling Target Buffer Availability: It is worth noting that the same remap *plan* is used multiple time during the 3d-FFT process. Hence the scratch buffer could be overwritten by the next `shmem_put`, if the sender computes faster and enters the next iteration before the data is unpacked at the receiver. It is henceforth necessary to protect the scratch buffer from early overwrite. In the initial version, we implemented this receive buffer readiness synchronization in OpenSHMEM with an `MPI_Barrier` (this version of the code thereafter referred to as *Hybrid-Barrier*). We use an `MPI_Barrier` rather than any of the `shmem_barrier_*` functions because we need to perform a synchronization in a subgroup whose shape is not amenable to OpenSHMEM group operations, while `MPI_Barrier` is able to force a synchronization on an arbitrary group. Figure 2 reflects the schematic structure of applying `shmem_put` for point-to-point communication in `remap_3d`.

An arguably more natural way of preventing that overwrite would be to synchronize with `shmem_int_p` when the receiver scratch buffer can be reused, and have the sender issue a `shmem_wait` to wait until that target buffer becomes available. However, each process updates the scratch buffers at multiple targets, and this strategy would result in ordering these updates according to the code

flow order, rather than performing the `shmem_put` operations opportunistically on the first available target. Hence, it would negate one of the advantages over the traditional `MPI_Send` based code, without making the overlap of computation and communication more likely when uneven load balance can make some targets significantly slower than others. We will discuss alternative approaches that avoid this caveat in the optimization Sect. 4.3.

4.3 Optimizations

Non-blocking `shmem_put` *to Avoid Packing Data:* As mentioned above, the MPI implementation of LAMMPS packs the non-contiguous data into a contiguous buffer before sending. A root reason comes from the higher cost of sending multiple short MPI messages compared to a single long one. The one-sided model, and the decoupling of the transfer and synchronization offer an opportunity for reaching better performance with OpenSHMEM when multiple small messages have to be sent, henceforth opening the possibility to directly transfer small, scattered data chunks without first copying into an intermediate pack buffer. As an example, `shmem_put` demonstrates a great performance advantage for sending multiple small messages when it is implemented over the `dmapp_put_nbi` function, a non-blocking implicit *put* in the Cray DMAPP API [1].

Still based on the *HybridBarrier*, a new version (called *NoPack*) removes the data packing. Each PE transfers the scattered data of size *nfast* to its peers directly, without resorting to an intermediate pack buffer. To transfer a full sub-domain of size $nslow \times nmid \times nfast$, a number of $nslow \times nmid$ `shmem_put` operations are required.

Eliminating Synchronization Barriers: In the *HybridBarrier* version, MPI point-to-point communications are replaced with OpenSHMEM one-sided communication semantics. To enforce synchronization between a sender and a receiver, a polling mechanism is employed. However, even with such a mechanism, an additional synchronization, in the initial OpenSHMEM version implemented with an `MPI_Barrier`, is still needed to prevent the next iteration from issuing `shmem_put` and modifying the dataset on which the current iteration is computing. This barrier could cause some performance loss, especially at large scale or when the load is not perfectly balanced.

In the *ChkBuff* version, the barrier is replaced by a fine grain management of the availability of the *scratch* buffer. A pair of communication statuses, `WAIT` and `READY`, are introduced to mark the status of the receiver's *scratch* buffer. A new array `plan->local_remote_status` stores, at the sender, the statuses of target receive buffers at all remote PEs. Before issuing a remote write on a particular PE's *scratch* buffer, the origin checks the availability status of the buffer in its local array. If the status is still set to `WAIT`, the `shmem_put` is delayed to a later date when, hopefully, the target PE will have unpacked the *scratch* buffer, thereby making it safe for reuse. When the target finishes unpacking the *scratch* buffer, it remotely toggles the status at the sender to `READY` (with a `shmem_int_p` operation), to inform the sender that it may start issuing `shmem_put` operations safely.

These synchronizations mimic the establishment of rendezvous between two-sided matching send-recv pairs. However, the MPI two-sided rendezvous can start at the earliest when both the sender and the receiver have posted the operations. In contrast, in the one-sided version, the receiver can toggle the availability status much earlier at the sender, as soon as the *unpack* operation is completed, which makes for a very high probability that the status is already set to `READY` when the sender checks for the availability of the target buffer. Unfortunately, we found that on many OpenSHMEM implementations, the delivery of `shmem_int_p` may be lazy, and delayed, negating the aforementioned advantage. The simple addition of a `shmem_quiet` call after all `shmem_int_p` have been issued is sufficient to force the immediate delivery of these state changes, and henceforth improves the probability that the state is already set to `READY` when a sender checks the status.

Opportunistic Unpacking: In the *ChkBuff* implementation, a PE unpacks its incoming data only after it completed the puts for all the outgoing data. The slightly modified *AdvChkBuff* version embraces a more dynamic, opportunistic ordering of the *unpack* operations. After issuing the `shmem_put` targeting the ready peers, the sender skips the peers which are still marked as in state `WAIT`, and instead switches to the task of unpacking the available incoming data. Before the next iteration, the sender then checks if it still has un-issued `shmem_put` operations from the current *plan* and satisfies them at this point. A major advantage of this method is that it overlaps the rendezvous synchronization wait time at the sender with data unpacking.

A design feature in the LAMMPS communication pattern, however, adds some complexity to the *AdvChkBuff* strategy. In the same `remap_3d`, the *in* and *out* buffers could point to the same memory blocks, in certain cases. This is not an issue when a PE sends out (or packs) all the outgoing data from the *in* buffer before it starts unpacking the received data from the *scratch* buffer. However, in the *AdvChkBuff* strategy, the unpack operation can happen before `shmem_put`. A memory block from the *in* buffer, where the data is waiting to be transferred to a slow PE, could thereby be mistakenly overwritten by the unpacking of the *scratch* buffer, touching that same block through the aliased pointer in the *out* buffer. This unexpected memory sharing between the two data buffers obviously threatens program correctness. To resolve this cumbersome sharing, in the conditional case where that behavior manifests, a temporary duplicate of the *out* buffer is allocated, and the unpack happens in that copy instead. The next iteration of the 3D-FFT will then consider that buffer as the input memory space, and the original buffer is discarded when the iteration completes (hence the `shmem_put` operations have all completed as well).

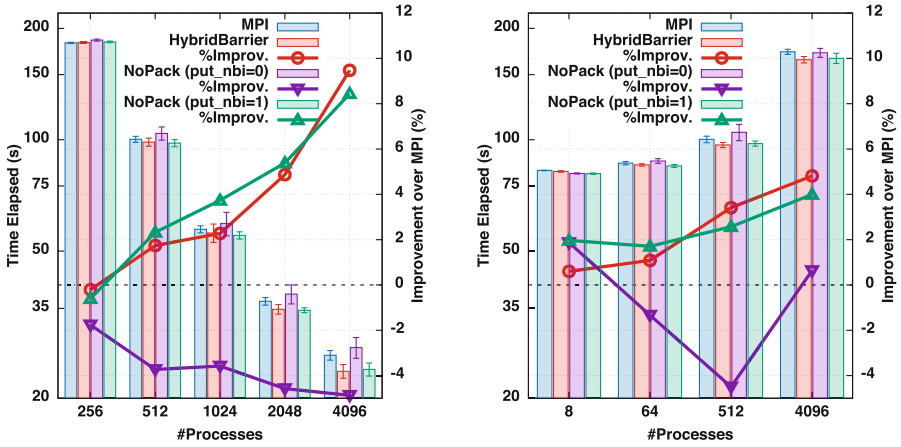
5 Evaluation

5.1 Experimental Setup

During the evaluation, we perform both strong and weak scaling experiments. In both cases we consider the *rhodopsin protein simulation* input problem. For the

strong scaling case, the input problem size remains constant at $8 \times 8 \times 8 \times 32\,\mathrm{K}$ atoms. For the weak scaling tests, the input problem size is set proportionally to the number of PEs, so that each processor handles a load of 32 K atoms.

The evaluations are performed on Titan, a Cray XK7 supercomputer located at ORNL (with Cray-MPICH 6.3.0 and Cray-shmem 6.3.0 software stacks). On this machine, even when two different allocations request the same number of nodes, they may be deployed on different physical machines, connected by a different physical network topology. This has been known to cause undesired performance variability that prevents directly comparing the performance obtained from different allocations. We eliminate this effect by comparing MPI versus OpenSHMEM on the same allocation, and by taking averages over 10 samples of each experiment. We present the total time of the LAMMPS application with error bars to illustrate any increase (or lack thereof) in the performance variability, and we then present the speedup of the considered OpenSHMEM enhanced variants over the MPI implementation.



(a) Strong Scaling. (b) Weak Scaling.

Fig. 3. Total LAMMPS execution time comparison between the following versions: original MPI, *HybridBarrier*, and *NoPack* (with and w/o non-blocking `shmem_put`).

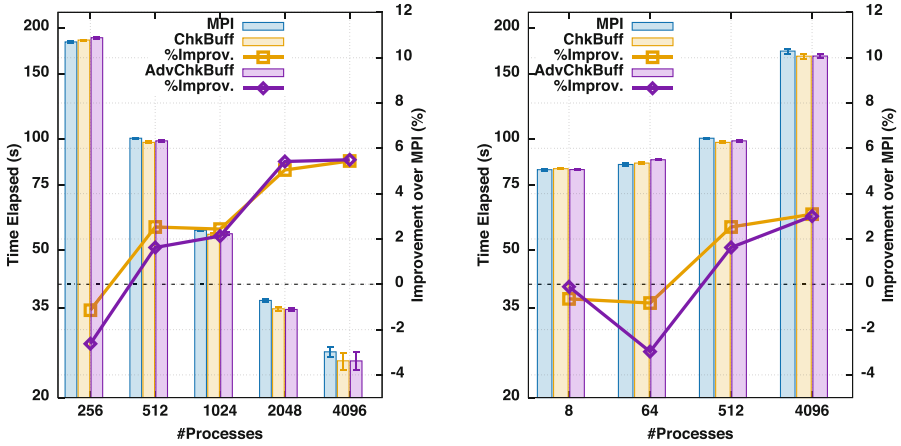
5.2 Comparison Between the *HybridBarrier* and MPI Versions

Figure 3 presents the strong scaling and weak scaling of LAMMPS for the original version where the `remap_3d` function is purely implemented with MPI, and the two hybrid versions (*HybridBarrier* and *NoPack*) where the `remap_3d` function features `shmem_put` communication and OpenSHMEM based communication completion signaling; yet the protection against premature inter-iteration data overwrite relies on an `MPI_Barrier` synchronization. The first observation

is that the standard deviation of results is very similar between all versions. The conversion to OpenSHMEM has not increased the performance variability. In terms of speedup, the *HybridBarrier* version enjoys better performance than the MPI version in all cases. The strong scaling experiments demonstrate an improvement from 2 % at 512 PEs, to over 9 % at 4096 PEs, which indicates that the OpenSHMEM APIs indeed accelerate the whole communication speed by amortizing the cost of multiple communication in each synchronization. This is mainly due to the per-communication cost of synchronization; in the original loop over `MPI_Send`, each individual communication synchronizes with the matching receive, in order; in contrast, and despite the coarse granularity of the `MPI_Barrier` synchronization, which may impose unnecessary wait times between imperfectly load balanced iterations, the cost of synchronizing in bulk numerous `shmem_put` operations to multiple targets is amortized over more communications and permits a relaxed ordering between the *puts*. Even in the weak scaling case, in which the communication speed improvement is mechanically proportionally diminished, the *HybridBarrier* version still outperforms the MPI version by 4 % at 4096 PEs, and as the system scale grows, the performance benefit of OpenSHMEM increases, which indicates that the OpenSHMEM programming approach has the potential to exhibit better scalability. One has to remember that these improvements are pertaining to the modification of a single function in LAMMPS, which represents approximately 30 % of the total communication time, meanwhile the communication time is only a fraction of the total time. Such levels of communication performance improvements are indeed significant, and would be even more pronounced should all MPI operations, which are still accounting for the most communication time in the hybrid version of LAMMPS, were ported to OpenSHMEM.

5.3 Consequences of Avoiding Packing

Figure 3 also compares the *NoPack* version to the MPI version. When the default environment respects the full, blocking specification for `shmem_put` operations (`DMAPP_PUT_NBI=0`), the performance of the *NoPack* version is reduced compared to both the MPI and *HybridBarrier* versions, which both pack the data. With this strict semantic, the injection rate for small messages is limited by the wait time to guarantee that the source buffer can be reused immediately. On the Cray system, the parameter (`DMAPP_PUT_NBI=1`) permits relaxing the `shmem_put` operation semantic: the `shmem_put` operations are allowed to complete immediately, but the source buffer can be modified only after a subsequent synchronization operation (like a `shmem_quiet`) has been issued. With this option, neither the MPI nor the *NoPack* versions performance are modified (an expected result, that parameter has no effect on MPI, and the *HybridBarrier* version exchanges large messages, and is therefore essentially bandwidth limited). However, the *NoPack* version's performance improves tremendously, as the injection rate for small messages is greatly increased. This observation provides strong evidence that the addition of a non-blocking `shmem_put` could result in significant benefits for some application patterns with a large number of small size message exchanges.



(a) Strong Scaling. (b) Weak Scaling.

Fig. 4. Total LAMMPS execution time comparison between the following versions: original MPI, *ChkBuff*, and *AdvChkBuff*.

However, as is illustrated in the `remap_3d` function, the packing strategy proves to be an effective workaround: even with non-blocking puts, the *NoPack* version closely matches the performance of the *HybridBarrier* version.

5.4 Performance When Eliminating Group Barriers

The goal of the *ChkBuff* and *AdvChkBuff* versions is to eliminate the group synchronization, employed in the *HybridBarrier* version, that avoid the premature overwrite of the *scratch* buffer by a process that has advanced to the next iteration. Their performance is compared to the MPI version in Fig. 4. The strong scaling of these versions (Fig. 4a) generally perform better than the MPI version, with a 2 % improvement from 512 PEs, and upto 5 % for 4096 PEs. In the weak scaling case, the benefit manifests only for larger PEs counts, with a 3 % improvement.

However, when comparing to the *HybridBarrier* version, presented in the previous Fig. 3, no further performance improvement is achieved. Although the barrier has been removed, it is likely that the introduction of synchronization between pairs of peers still transitively synchronize these same processes. In addition, the busy waiting loop on the peer's buffer status scans the memory and increases the memory bus pressure. It finally appears that, in the OpenSHMEM model, trying to optimize the scope of synchronizations can result in decreasing performance, by synchronizing at too fine a granularity, which actually mimics the traditional implicit synchronization pattern found in MPI two-sided applications.

5.5 Discussion

This porting effort has led to a few observations on the entire process. Today, MPI is considered as the de-facto programming paradigm for parallel applications, and PGAS type languages are still only challengers. Shifting from one programming model to another is not an easy task: it takes time and effort to educate the developers about the new concepts, and then correctly translate the application from one programming paradigm to another.

The Good: Although MPI is a more stable, well-defined API, PGAS based approaches have clear advantages for some types of usage patterns. Until OpenSHMEM reaches the same degree of flexibility and adaptability, being able to compose the two programming paradigms in the context of the same application is a clear necessity. Our experience confirmed that it is indeed possible to mix the programming models, and to upgrade an application in incremental steps, focusing on performance critical routines independently in order to take advantage of the strong features of each model, and improve the performance and scalability of a highly-optimized MPI application.

An area where the PGAS model exhibit a clear advantage over MPI is the handling of unexpected messages. when MPI applications get out of sync due to imbalance in the workload or system noise, most messages become unexpected, forcing the MPI library to buffer some data internally to delay the delivery until the posting of the corresponding receive call. This temporary buffering has an impact on the memory accesses as it implies additional memory copies and thus extra overhead on the memory bus. As the memory is always exposed in OpenSHMEM, even when the target is passive, this pathological usage pattern cannot be triggered. In exchange, some of the implicit synchronization semantic carried by the send-recv model is lost, and explicit synchronization becomes necessary.

The Bad: One of the most unsettling missing features from the OpenSHMEM standard is the capability to work with irregular process groups. The current group collective operations are limited to well defined process topologies, mostly multi-dimensional cubes. As a consequence, they lack the flexibility of their MPI counterparts which operate on communicators and can express collective behaviors across arbitrary groups of processes. This leads to less natural synchronization points, especially in non-symmetric cases.

Another missing feature is a standardized support of the non-blocking *put* capabilities of the network. Forcing the `shmem_put` operation to ignore its strict specification and relaxing the total completion of the operation to the next synchronization does improve tremendously the injection rate for small messages. However, today, one has no portable way to achieve this result in the OpenSHMEM specification, and relying on packing still remains the best bet to achieve the maximum bandwidth when the input data are scattered.

The Ugly: One of the selling points of OpenSHMEM is the exposure at the user level of *bare metal* network primitives, with a promise for a more direct access

to the hardware and higher performance compared with more abstract programming paradigms, such as MPI. However, the restricted API of OpenSHMEM, and especially the lack of high level primitives such as collective communication with neighborhood communication patterns, forces the application developers to design and implement their own synchronization operations, making the applications sensitive to hardware and network topological feature variations. Furthermore, the explicit management of synchronization from the user code (instead of the implicit synchronization carried by the two-sided operations) can result in notable performance differences, depending on the strategy one employs to restore the required synchronization for a correct execution. We found that some theoretically promising optimizations can actually yield a negative effect on overall performance, and that the reasons for these detrimental consequences are often subtle and hard to predict. Overall, the balance between portability and performance is weaker than in MPI.

6 Conclusions and Future Work

In this work, we explore the process of converting the communication-intensive part of an MPI-based application, LAMMPS, to OpenSHMEM. We reveal some major programming challenges introduced by the semantics and syntax differences between the two programing models. We demonstrate how to transform a common communication pattern based on MPI point-to-point communication into the corresponding OpenSHMEM version. We evaluate our work on the Titan supercomputer looking at both strong scaling and weak scaling tests up to 4096 processes. While our work was successful in demonstrating a clear advantage in terms of performance for the OpenSHMEM hybridized version, choosing between different approaches and optimizing is not straightforward. Looking at more details and differences between different versions employing OpenSHMEM reveal counter-intuitive and significant performance differences between the possible explicit synchronization strategies that are not always matching expectations. It becomes apparent that a mechanical conversion from MPI to OpenSHMEM is not the most suitable approach. Instead, extracting the most performance from a programming paradigm requires adapting the underlying algorithm to the concepts exposed by the paradigm, rather than a simple one-to-one translation.

Our future work will address this last point, making deeper changes in the algorithm to avoid the additional synchronizations between processes. This includes investigating the opportunities to reduce the number and scope of synchronizations in the OpenSHMEM-based LAMMPS. We can also combine the presented optimizations, for example investigate if it is possible to remove the packing in the *AdvChkBuff* version. Last, with only a subset of the communication routines upgraded to OpenSHMEM, we have observed a significant reduction in runtime for the entire application. We will therefore identify other communication-intensive functions in LAMMPS and start the process of replacing them with their OpenSHMEM counterparts.

Acknowledgements. This material is based upon work supported by the U.S. Department of Energy, under contract #DE-AC05-00OR22725, through UT Battelle subcontract #4000123323. The work at Oak Ridge National Laboratory (ORNL) is supported by the United States Department of Defense and used the resources of the Extreme Scale Systems Center located at the ORNL.

References

1. Using the GNI and DMAPP APIs. Technical Report S-2446-3103, Cray Inc. (2011). http://docs.cray.com/books/S-2446-3103/S-2446-3103.pdf
2. OpenSHMEM application programming interface (version 1.2). Technical report, Open Source Software Solutions, Inc. (OSSS) (2015). http://www.openshmem.org
3. Barriuso, R., Knies, A.: SHMEM's user's guide for C. Technical report, Cray Research Inc. (1994)
4. Gerstenberger, R., Besta, M., Hoefler, T.: Enabling highly-scalable remote memory access programming with MPI-3 one sided. Sci. Program. **22**(2), 75–91 (2014). doi:10.3233/SPR-140383
5. Jose, J., Potluri, S., Subramoni, H., Lu, X., Hamidouche, K., Schulz, K., Sundar, H., Panda, D.K.: Designing scalable out-of-core sorting with hybrid MPI+PGAS programming models. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, PGAS 2014, pp. 7:1–7:9. ACM, New York (2014). doi:10.1145/2676870.2676880
6. Jose, J., Potluri, S., Tomko, K., Panda, D.K.: Designing scalable graph500 benchmark with hybrid MPI+OpenSHMEM programming models. In: Kunkel, J.M., Ludwig, T., Meuer, H.W. (eds.) ISC 2013. LNCS, vol. 7905, pp. 109–124. Springer, Heidelberg (2013)
7. Li, M., Lin, J., Lu, X., Hamidouche, K., Tomko, K., Panda, D.K.: Scalable MiniMD design with hybrid MPI and OpenSHMEM. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, PGAS 2014, pp. 24:1–24:4. ACM, New York (2014). doi:10.1145/2676870.2676893
8. Li, M., Lu, X., Potluri, S., Hamidouche, K., Jose, J., Tomko, K., Panda, D.: Scalable graph500 design with MPI-3 RMA. In: 2014 IEEE International Conference on Cluster Computing (CLUSTER), pp. 230–238, September 2014. doi:10.1109/CLUSTER.2014.6968755
9. MPI Forum. MPI: A Message-Passing Interface Standard (Version 2.2). High Performance Computing Center Stuttgart (HLRS), September 2009
10. Plimpton, S.: Parallel FFT package. Technical report, Sandia National Labs. http://www.sandia.gov/~sjplimp/docs/fft/README.html
11. Plimpton, S.: Fast parallel algorithms for short-range molecular dynamics. J. Comput. Phys. **117**(1), 1–19 (1995). doi:10.1006/jcph.1995.1039
12. Poole, S.W., Hernandez, O.R., Kuehn, J.A., Shipman, G.M., Curtis, A., Feind, K.: OpenSHMEM - toward a unified RMA model. In: Padua, D.A. (ed.) Encyclopedia of Parallel Computing, pp. 1379–1391. Springer, Heidelberg (2011). doi:10.1007/978-0-387-09766-4_490
13. Pophale, S., Nanjegowda, R., Curtis, T., Chapman, B., Jin, H., Poole, S., Kuehn, J.: OpenSHMEM Performance and Potential: An NPB Experimental Study. In: Proceedings of the 6th Conference on Partitioned Global Address Space Programming Model, PGAS 2012. ACM, New York (2012)
14. Vetter, J.S., McCracken, M.O.: Statistical scalability analysis of communication operations in distributed applications. SIGPLAN Not. **36**(7), 123–132 (2001). doi:10.1145/568014.379590