Designing, Implementing, and Evaluating the Upcoming OpenSHMEM Teams API

David Ozog  
Intel Corporation  
Hudson, Massachusetts USA  
david.m.ozog@intel.com

MD. Wasi-ur- Rahman  
Intel Corporation  
Austin, Texas USA  
md.rahman@intel.com

Gerard Taylor  
Intel Corporation  
Hudson, Massachusetts USA  
gerard1.taylor@intel.com

James Dinan  
Intel Corporation  
Hudson, Massachusetts USA  
jamies.dinan@intel.com

Abstract—For many years, the OpenSHMEM parallel programming interface has provided a high-performance alternative to MPI that emphasizes one-sided messaging, simplifies communication across a global memory space, and bolsters the capabilities of rapidly evolving fabric interconnect technologies. The OpenSHMEM specification standardizes the library interfaces, prioritizing a performant and portable API. The specification continues to mature with vigorous support from several authoritative vendors and researchers. For example, the OpenSHMEM specification committee is actively standardizing a unique teams API that enables user-defined subsets of application processes to efficiently and productively perform communication operations, such as collective routines, remote memory accesses, and remote atomic operations.

This paper describes the OpenSHMEM teams interface and several interesting aspects and challenges in implementing the API, as well as possible extensions that could improve the programmability and/or performance. We evaluate the performance of a preliminary implementation and show that using teams effectively can facilitate impressive improvements of collective operations at scale (2-16x speedup, depending on the algorithm, node count, and buffer size), even while simplifying the underlying programming model.

Index Terms—OpenSHMEM, PGAS, HPC, parallel computing, distributed systems

I. INTRODUCTION

For over a quarter of a century, the SHMEM interface has provided an effective alternative to MPI (the Message Passing Interface) by emphasizing Partitioned Global Address Space (PGAS) programming, simplifying one-sided messaging semantics, and efficiently utilizing modern high-speed interconnection network capabilities [1], [2]. For almost a decade, the OpenSHMEM community has worked to standardize SHMEM by defining an open specification that provides a common, portable, and consistent interface regardless of vendor implementation.

It is well known that High Performance Computing (HPC) systems are evolving towards extremely heterogeneous configurations, where compute nodes potentially have multiple network interface devices, specialized accelerator devices, non-uniform memory hierarchies, various memory kinds, and much more. With such heterogeneity becoming the new norm, programming models such as OpenSHMEM must adapt to this challenging programming environment. Accordingly, the OpenSHMEM community is exploring several new proposals, such as teams and user-defined memory spaces [3], memory partitions [4], and other extensions for hybrid and heterogeneous computing [5], [6].

This paper focuses on the proposed OpenSHMEM teams API, which is being incorporated into the next release of the OpenSHMEM specification. Teams provide applications with a method for creating groups of OpenSHMEM processes that can communicate with each other using point-to-point and collective communication operations. In this regard, OpenSHMEM teams are similar to MPI communicators and can be used to capture hierarchical and spatial parallel decomposition, similar to how MPI communicators are often used. In addition, teams are also being considered as a method for exposing system topology and hybrid system resources to applications, allowing programmers to optimize for these aspects of modern system architectures.

In this paper, we motivate and present the design of the teams API. We describe an implementation of the proposed teams API in the open source Sandia OpenSHMEM [7] library, which supports the Portals 4 and libfabric communication layers. We further analyze the performance characteristics and resource consumption of our implementation and discuss opportunities for future expansion of the proposed API.

II. BACKGROUND AND MOTIVATION

Like MPI, OpenSHMEM deploys a single program multiple data (SPMD) model in which all processing elements (PEs) execute the same program executable, but operate on separate data. In particular, every PE has a unique identifier number, which is used to differentiate between PEs and to aid in specifying different data objects. Unlike MPI, OpenSHMEM requires designating a memory region of remotely accessible symmetric data objects (symmetric objects, for short) so that each PE can perform remote memory accesses (RMAs) without the explicit participation of the remote PE. In particular, symmetric objects have the same name, type, and size on all PEs, and applications can dynamically allocate symmetric objects in a region that is called the symmetric heap. The primary advantage of this programming model is that performing one-sided operations is simple and efficient; it benefits many computational applications where data decomposition across PEs is straightforward, and when it is prudent for communication to occur without requiring target PEs to post receive operations. One disadvantage of this programming model is
that some operations, such as the allocation of symmetric objects (e.g. via the \texttt{shmem_malloc} routine), must occur on all PEs, requiring expensive synchronization. Additionally, while one-sided messaging semantics are generally convenient, great care must be taken by application developers to avoid subtle race conditions that may occur from PEs simultaneously interacting with symmetric data regions.

The current OpenSHMEM 1.4 specification supports a variety of collective operations including the barrier, broadcast, reduction, gather, and all-to-all communication patterns. However, because of the ingrained asynchrony of one-sided remote and local accesses to the symmetric regions, most OpenSHMEM collectives APIs require passing a symmetric synchronization array called a \texttt{pSync} that is used to synchronize the one-sided communication operations performed in the implementation of the collective operation. OpenSHMEM 1.4 collective operations are performed on a subset of PEs referred to as the active set, that are specified to the collective operation using a starting PE index, log base two stride, and an active set size. PEs that participate in a collective must pass a properly initialized \texttt{pSync} array of the appropriate size, and users must take care to ensure that the \texttt{pSync} memory is not in use by any other collective operation (e.g. a prior or concurrent operation) at any PE in the active set. A common pattern seen in optimized OpenSHMEM applications is to initiate multiple \texttt{pSync} arrays to enable back-to-back collectives, as in Algorithm 1. Furthermore, the OpenSHMEM reduction operations must collectively operate on a symmetric data region, so these routines also include a \texttt{p\_rk} array that must be properly initialized and sized, similar to the \texttt{pSync} array.

The OpenSHMEM teams API, proposed for inclusion in the OpenSHMEM 1.5 specification, provides mechanisms for grouping together sets of processing elements (PEs) that will communicate with each other. A key goal of the teams API is to eliminate \texttt{pSync} and \texttt{p\_rk} and to improve the usability of the collectives API by relaxing the current active set model. A forward-looking goal beyond the OpenSHMEM 1.5 teams API is to create structures for future versions of OpenSHMEM that reduce the cost of global synchronization operations, such as \texttt{shmem_malloc}, by only including active team members. For example, if each team manages a separate symmetric heap (as suggested in [3]) then PE subsets that operate on particular symmetric data only need to synchronize within that subset for instances such as symmetric heap allocations. Furthermore, the OpenSHMEM 1.5 teams API lays groundwork towards offering better support for programming on heterogenous systems by exposing separate symmetric heaps across different memory kinds [4]. Such an interface would provide a useful abstraction for locality-aware optimizations, which until now has been under the purview of application and/or library developers [8].

OpenSHMEM 1.5 teams also allow for more flexible and efficient collective operations as these operations require synchronizations only across the members of the team. Underlying fabric interconnect resources that support and/or accelerate collective operations are inherently limited. Teams provide an abstraction that allows OpenSHMEM implementations to more efficiently manage these resources and applications to have implicit control over such resources.

Finally, the recent introduction of a communication management interface in OpenSHMEM 1.4 [2], [9] complements the teams interface nicely. As currently defined, these communication contexts are associated with teams and enable point-to-point operations to occur within the team’s PE indexing space. This provides OpenSHMEM implementations the opportunity to manage and coordinate communication resources in a way that isolates multi-threading environments across different contexts within a team.

While many of the possible teams-associated features mentioned in this section are somewhat ambitious and still remain as proof-of-concepts, the next section describes the officially proposed OpenSHMEM teams API to delineate the research topics from features that are expected to be standardized in the OpenSHMEM 1.5 specification.

A. The OpenSHMEM Teams API Proposal

This section provides a brief summary of the OpenSHMEM teams API proposal as it is currently defined. We focus on features that are most relevant to the implementation in Sandia OpenSHMEM as described in Section IV, but there are many other characteristics not covered here. If the reader is interested in more explicit details, such as the API function signatures, please refer to the official proposal document managed by the OpenSHMEM Teams and Collectives subcommittee [10].

Teams are created via a split operation (currently, by calling either the \texttt{shmem_team_split_strided()} or the \texttt{shmem_team_split_2d()} routine), which forms a child team consisting of a subset of PEs from within an existing parent team. The \texttt{default} team is the team consisting of all PEs launched by the application, and is referred to by the handle, \texttt{SHMEM\_TEAM\_WORLD}. Unless another team is specified, all point-to-point and collective communication operations are performed on the default team.

During a split operation, a child team subset is specified by a PE triplet: \texttt{(start, stride, size)} where \texttt{start} is the first PE identifier in the subset, \texttt{stride} is the number of PEs between subsequent members, and \texttt{size} is the total number of PEs in the team. For example, if the default team consists of PE identifiers \{0, 1, \ldots, 10\}, then the child team specified by triplet \{3, 2, 4\} consists of PEs \{3, 5, 7, 9\}. This design choice has an interesting consequence on the OpenSHMEM library implementation’s internal representation of a team: it can always be expressed by a \texttt{(start, stride, size)} triplet. Even with the 2D split routine, which splits a parent team into \texttt{x} and \texttt{y} teams by specifying an \texttt{xrange} (See Figure 1), the child teams always have well-defined \texttt{start}, \texttt{stride}, and \texttt{size} values.

One disadvantage of defining teams in terms of the PE triplet is that it is not possible to split a parent team into any arbitrary subgroup. For example, if a parent teams consists of PE identifiers \{0, 1, 2, 3, 4, 5, 6\}, then one cannot represent a child
which consists of all PEs that share a memory domain. As

\texttt{SHMEM\_TEAM\_SHARED} calls, but that is beyond the scope of this paper.

no need for application synchronization between the creation

back-to-back team creation with overlapping membership and

believe it is possible to devise an algorithm that would support

such complications by simply requiring that team creation be

interprocess shared memory programming, similar to the model supported

via the existing \texttt{shmem\_ptr} routine.

The notion of grouping distributed processes and/or threads in parallel programming models is by no means entirely new. This section considers existing approaches in other programming models and juxtaposes them with OpenSHMEM teams.

Firstly, OpenSHMEM teams are analogous to MPI communicators, which are inherent to the MPI programming model [12]. Several analogues exist between these APIs: \texttt{SHMEM\_TEAM\_WORLD} to \texttt{MPI\_COMM\_WORLD} and \texttt{shmem\_team\_split\_2d} to \texttt{MPI\_Cart\_create} to name a couple; but there are several differences as well. For instance, \texttt{MPI\_Comm\_split} is a color-based split operation that allows the user to specify an arbitrary assignment of processes to communicators, whereas OpenSHMEM teams currently limit users to teams that can be defined by the simple triplet as described in Section II-A. \texttt{MPI\_Groups} are analogous to the OpenSHMEM Sets/Groups concept that was proposed for, but not included in the final OpenSHMEM teams API [11]. As a result, there is no direct analogue to performing intersection and/or union operations with OpenSHMEM teams.

As of MPI 3.0, \texttt{MPI\_Comm\_Split\_Type} with \texttt{MPI\_COMM\_TYPE\_SHARED} enables the partitioning of processes that share memory, similar to \texttt{SHMEM\_TEAM\_SHARED} as described in Section II-A. The resulting MPI communicator can be used in conjunction with \texttt{MPI\_Win\_allocate\_shared} to provide interprocess shared memory programming, similar to the model supported via the existing \texttt{shmem\_ptr} operation.

The UPC programming language has explored the utility of teams for collectives [13], and the UPC++ library defines team objects [14]. In UPC++, all collective operations occur on a team, \texttt{world()} is the default team, split operations are color-based, and a unique \texttt{team\_id} serves as the team handle. Similar to OpenSHMEM, UPC++ has a shared segment, which leads to an interesting analogue with \texttt{SHMEM\_TEAM\_SHARED}: the UPC++ \texttt{local\_team} object. The \texttt{local\_team} is an ordered set of processes where heap storage in the shared segment allocated by any process in the team is local to all members [15].

Many other SPMD-based PGAS programming models support teams. For instance, a recent Fortran technical specification introduces extensive support for teams of images (or processes) with support for coarray collective intrinsics [16]. The X10 language [17] supports teams with a split operation. DASH, a C++ based PGAS runtime also has teams with
A split operation, as well as mechanisms to detect locality domains and hierarchies using PAPI and hwloc [18]. DART, a DASH runtime over MPI-3, has support for memory allocation within a specified team [19]. Even some non-SPMD models, such as HPX and Chapel, also refer to notions grouped processes, but tend towards “locales” [20] or “localities” [21] that more directly reflect underlying physical memory domains and/or compute node topologies. Such abstractions are more conducive to these programming models.

IV. DESIGN AND IMPLEMENTATION

This section summarizes the software design and architecture of OpenSHMEM teams within the Sandia OpenSHMEM (SOS) library [7], which is an implementation of the OpenSHMEM specification with support for the Open Fabrics Interface (OFI), Portals 4.0, and XPMEM [22]–[24].

A. Software Considerations

A team in SOS consists of a simple C-struct containing a unique team ID, a pSync index, a (start, stride, size) triplet, a configuration structure, and a list of contexts. During initialization of the SOS runtime, these fields are set for the default team, SHMEM_TEAM_WORLD, and for the shared memory team, SHMEM_TEAM_SHARED. In addition, SOS defines SHMEMX_TEAM_HOST, which is a team comprising all PEs that share the same host name; as well as SHMEMX_TEAM_LEADER, which selects a single PE from each unique host to make up a team of leader PEs. The “x” in SHMEMX signifies that these constants are extensions of the OpenSHMEM specification.

While it may seem like SHMEM_TEAM_SHARED and SHMEMX_TEAM_HOST should consist of the same PE members, this is generally not the case. Section II-A stated that the OpenSHMEM shared memory team is defined in terms of memory accessibility via the shmem_ptr routine. SOS only returns a non-null pointer via shmem_ptr when using an on-node communication library, such as XPMEM. If neither of these transports are enabled, then shmem_ptr returns a non-null value for the self PE only. In other words, SHMEM_TEAM_SHARED only consists of the self PE, unless XPMEM is enabled. However, we will see in Section V-B that significant performance improvements can be attained by using SHMEMX_TEAM_HOST together with SHMEMX_TEAM_LEADERS to optimize for system topology even when the on-node transports are disabled.

SOS checks that the shared memory and host memory teams are representable with a (start, stride, size) triplet by searching for on-node PEs. For SHMEM_TEAM_SHARED, this is done via shmem_ptr; and for SHMEMX_TEAM_HOST, this is done by either querying the PMIx runtime [25] for PMIX_LOCAL_PEERS, or by calling gethostname() when PMIx is not available. SOS then assures that there is a constant stride-value between all local PEs. In practice, the job launcher usually assigns either sequential IDs or round-robin IDs to on-node processes. In both circumstances, a valid triplet exists for both shared memory and host memory teams. However, it is certainly possible for the job launcher (for example, via a user’s custom host file) to assign arbitrary PE values, potentially resulting in a PE subset that is not representable as a valid triplet. Since all PEs in the team must have the same PE subset and each instance of SHMEM_TEAM_SHARED must also include the local PE, a single team cannot be created that includes all PEs on the node. In such cases, SOS simply resorts to including only the self PE in both SHMEM_TEAM_SHARED and SHMEMX_TEAM_HOST.

B. Team-Based Point-to-Point Operations

All point-to-point operations, such as puts and gets as well as all atomic memory operations (AMOs), operate on an OpenSHMEM context [9], [26], [27]. With the addition of the teams API, all contexts are associated with a particular team. In SOS, the internal context type simply contains a pointer to the appropriate team object. Accordingly, supporting team-based point-to-point communication involves relatively few changes to SOS internals: translating a team PE index to global PE index (start + stride*team_pe_num) before all point-to-point operations, associating the default context to the default team at initialization, and freeing all associated contexts when destroying a team.

C. Team Creation

In addition to setting up the default and shared memory teams, SOS initialization also allocates designated pSync regions for the pre-defined teams and user-defined teams. Figure 2 shows a graphic that represents the memory layout for these pSync buffers. An environment variable, SHMEMX_TEAMS_MAX, indicates the total number of pSync arrays to allocate on the symmetric heap for all teams (the default is 10). To support back-to-back collectives, SOS manages two separate pSync arrays for each team: the first set is for the standard application-level collectives, and the second set is for internal synchronizations, barriers, syncc, etc. Since synchronization operations can be performed back-to-back, the latter pSync array can be used to synchronize the former to support arbitrary back-to-back collective operations.

It is critical that each PE agrees on a particular pSync index before conducting team collective operations. For the pre-defined teams, this agreement is straightforward: at initialization, SHMEM_TEAM_WORLD, SHMEM_TEAM_SHARED, SHMEMX_TEAM_HOST, and SHMEMX_TEAM_LEADERS claim pSync indices 0, 1, 2, and 3, respectively. For the user-defined teams, selecting a common pSync index requires collective...
agreement. To accomplish this, SOS maintains an availability bitmask where each bit represents whether or not a particular pSync region is available at the time of team creation. During SOS initialization, all bits across the bitmask are set to 1 (except the 0th, 1st, 2nd, and 3rd bits, corresponding to the pre-defined teams), indicating that the corresponding pSync buffers are available. During team creation, each PE in the active set participates in an AND reduction across the availability bitmask, selecting the lowest-order non-zero bit. This bit slot is set to 0, and then corresponds to the pSync index for the newly created team until the team is destroyed, when the bit is again set to 1.

D. Collectives

SOS supports several different algorithms for performing OpenSHMEM collectives. For instance, SOS may deploy either a linear, tree, recursive doubling, ring, and/or dissemination algorithm, depending on the collective operation, the number of PEs in the active set, and the operand buffer size(s). Section V below presents results for the recursive doubling and ring algorithms applied to reductions, so we very briefly describe those two specific algorithms here. More explicit details can be found in other seminal literature on optimizing collective operations [28], [29].

The recursive doubling reduction algorithm involves pairwise exchanges of data between PEs whose IDs differ by increasing factors of 2. For example, in a reduction starting at PE 0 with unit stride, the first exchange occurs between neighboring PEs (PE 0 exchanges with 1, 2 exchanges with 3, etc.). The second exchange occurs between every 2nd PE (0 exchanges with 2, 1 with 3, etc.). The third exchange occurs between every 4th PE, and so on. Every round accumulates the result of the previous round. Recursive doubling is beneficial for small message sizes, where the dominant term is due to the number of rounds, which is logarithmic with the total number of PEs.

For reductions, the ring algorithm performs a bandwidth optimized reduce-scatter followed by an all-gather. Unlike the recursive doubling algorithm, the communication latency incurred is linear, rather than logarithmic, with respect to the total number of PEs. However, the bandwidth consumed is proportional to the message size, whereas the recursive doubling algorithm consumes bandwidth that is proportional to the product of the message size and the logarithm of the number of PEs. Therefore, we expect to see the ring algorithm perform better for large-buffer (bandwidth-sensitive) reductions, whereas the recursive doubling algorithm to perform better for small-buffer (latency-sensitive) reductions.

V. EXPERIMENTS

We evaluate our OpenSHMEM teams implementation on two different systems: an internal cluster named Diamond and the Cori supercomputer at the National Energy Research Scientific Computing Center (NERSC). Diamond’s compute nodes contain 2 CPU sockets, each with an Intel® Xeon® Platinum 8170 (Skylake) CPU at 2.1 GHz and 192 GB of DDR4 2666 MHz RAM. Each CPU has 2-way Intel® Hyper-Threading Technology with 26 cores, providing 52 physical cores and 104 hardware thread contexts per compute node. Nodes are connected via the Intel® Omni-Path 100 series fabric (Intel® OPA) with a fat-tree topology. Intel® OPA supports 100 Gigabits per second in each link direction, corresponding to 12.5 GB/s of uni-directional bandwidth and 25 GB/s of bi-directional bandwidth.

The operating system on Diamond’s compute nodes is Red Hat® Enterprise Linux Server release 7.5 (Maipo) with Linux® kernel 3.10.0-862.el7.x86_64, and all binaries are built using the wip/teams branch of Sandia OpenSHMEM (SOS) based on version 1.4.4, GNU GCC version 4.8.5, and libfabric version 1.7.0 with the PSM2 provider, manual progress enabled, and a progress interval of 1 microsecond. We use the MPICH Hydra process launcher version 3.2.1 to execute all jobs and restrict processes to be bound to a particular CPU socket (--bind-to=socket).

As noted in a separate work, enabling manual progress can substantially improve the performance of latency-sensitive OpenSHMEM applications on Intel® OPA platforms [30]. With manual progress, each target PE reads the local completion queue for events associated with data transfers at opportunistic moments, such as during shmem_quiet, shmem_wait, and shmem_test operations. With automatic progress, a separate progress thread makes runtime progress at regular intervals. Throughout our experiments below, SOS was configured to enable manual progress (--enable-manual-progress) as well as automatic progress. We reduce the automatic progress polling interval to the lowest possible value (FI_PSM2_PROG_INTERVAL=1), because this marginally improves the latency of collective operations.

NERSC’s Cori supercomputer is a Cray® XC40 sytem [31]. Cori contains 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. Our measurements focus on the Haswell nodes, each having 128 GB of DDR4 2133 MHz RAM. Each CPU has 2-way Intel® Hyper-Threading Technology with 32 cores, providing 32 physical cores and 64 hardware thread contexts per compute node. All the compute nodes run a lightweight Cray® Linux Environment based on the SuSE® Linux Enterprise Server distribution. Cori deploys the Aries interconnect with a dragonfly topology [32]. The Aries NIC bandwidth on symmetric bulk data transfers is approximately 8 GB/s in each direction [32]. Similar to the Diamond setup, all binaries are built using a modified branch of Sandia OpenSHMEM (SOS) based on version 1.4.4, GNU GCC version 8.2.0, and libfabric version 1.8.x with the GNI provider.

The experiments in Section V-B utilize the OSU Micro-Benchmark collection [33], [34] version 5.6.2. In particular, measurements are taken from the OpenSHMEM sum reduction benchmark, comparing the original algorithm with a slightly modified version that uses the pre-defined shared host memory team (SHMEMX_TEAM_HOST) and the leader team (SHMEMX_TEAM_LEADERS). We configure the OSU bench-
marks with default parameters unless otherwise noted and the same SOS setups described above.

A. Team Creation Latency

As discussed in Section IV-C, the OpenSHMEM teams API currently requires team creation to be collective on the parent team, making it an inherently expensive synchronization operation. For user-defined teams, the Sandia OpenSHMEM design includes a bitwise AND reduction across the availability bitmask (shown in Figure 2) to determine an appropriate pSync slot in which to perform subsequent team-based collective operations. The goal of our first experiment is to measure the performance cost of doing team creation operations and to quantify the relative cost of the bitwise reduction compared to any other overheads.

Figure 3 shows the execution time of a microbenchmark that performs a series of team creation operations (via the shmem_team_split_strided routine). The red curve shows the time taken by repeatedly duplicating SHMEM_TEAM_WORLD. The green curve shows the corresponding execution time consumed by the AND reduction across the availability bitmask, which comprises almost all of the team creation time. The green curve shows a similar measurement, except that each split operations includes only half of the PEs in the parent team. The parent team always starts at PE 0, and the stride increments by $2^i$ for the $i$th team creation, resulting in the teams $\{0, 1, \ldots, N\}, \{0, 2, 4, \ldots, N/2\}, \{0, 4, 8, \ldots, N/4\}$, and so on, repeating the process when necessary. The yellow curve shows the time spent in the availability bitmask reduction when halving the size of each new team. We see that the reduction consumes relatively less time in this case, because only the PEs in the active set perform the reduction, whereas all PEs in the parent team perform a barrier synchronization.

At the time of this writing, SOS supports up to 64 simultaneous teams (corresponding to a uint64_t value used for the availability bitmask). Accordingly, the team creation microbenchmark used to generate Figure 3, measures the latency to create a maximum of 64 teams. Additionally, to reduce measurement variation, we measure the latency of 64 team creations over 1000 iterations and calculate the mean. After each iteration, all 64 teams must be destroyed to free slots in the availability bitmask.

Algorithm 1: Original OSU Sum Reduce Microbenchmark

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    int size = 1;
    int iterations = 1000;
    double timer = 0;
    float *recvbuf, *sendbuf;
    long *pSync1, *pSync2;
    float *pWrk1, *pWrk2;

    // Initialize 2 pSync arrays and 2 pWrk arrays:
    static long pSync1[SHMEM_REDUCE_SYNC_SIZE];
    static long pSync2[SHMEM_REDUCE_SYNC_SIZE];
    for (int t = 0; t < SHMEM_REDUCE_SYNC_SIZE; t++)
        pSync1[t] = SHMEM_SYNC_VALUE;
    for (int t = 0; t < SHMEM_REDUCE_SYNC_SIZE; t++)
        pSync2[t] = SHMEM_SYNC_VALUE;

    float *pWrk1 = (float*)shmem_malloc(sizeof(float) * MAX(nreduce/2+1, SHMEM_REDUCE_MIN_WRKDATA_SIZE));
    float *pWrk2 = (float*)shmem_malloc(sizeof(float) * MAX(nreduce/2+1, SHMEM_REDUCE_MIN_WRKDATA_SIZE));

    // Loop over buffer sizes:
    for(int size=1; size <= max_msg_size; size *= 2) {
        // shmem_barrier_all();
        timer = 0;

        // Timed loop:
        for(int i = 0; i < iterations; i++) {
            t_start = TIME();
            shmem_float_sum_to_all(recvbuf, sendbuf, size, 0, 0, numprocs, pWrkF1, pSync1);
            if (SHMEMX_TEAM_LEADERS != SHMEMX_TEAM_INVALID)
                shmem_float_sum_to_all(recvbuf, recvbuf, size, 0, 0, numprocs, pWrkF2, pSync2);
            t_stop = TIME();
            timer += t_stop - t_start;
            // shmem_barrier_all();
        }
        latency = (double)(timer * 1.0) / iterations;
    }
    return 0;
}
```

Algorithm 2: Teams-based Modification of Algorithm 1

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    int size = 1;
    int iterations = 1000;
    double timer = 0;
    float *recvbuf, *sendbuf;
    long *pSync1, *pSync2;
    float *pWrk1, *pWrk2;

    // Loop over buffer sizes:
    for(int size=1; size <= max_msg_size; size *= 2) {
        // shmem_barrier_all();
        timer = 0;

        // Timed loop:
        for(int i = 0; i < iterations; i++) {
            t_start = TIME();
            shmemx_float_sum_reduce(SHMEMX_TEAM_HOST, recvbuf, sendbuf, size);
            if (SHMEMX_TEAM_LEADERS != SHMEMX_TEAM_INVALID)
                shmemx_float_sum_reduce(SHMEMX_TEAM_LEADERS, recvbuf, recvbuf, size);
            t_stop = TIME();
            timer += t_stop - t_start;
            // shmem_barrier_all();
        }
        latency = (double)(timer * 1.0) / iterations;
    }
    return 0;
}
```
B. Shared-Memory Team Performance

Because a majority of HPC applications utilize collective operations [35], many real-world applications would directly benefit from performance enhancements to collective communication algorithms. This section describes a teams-based optimization of OpenSHMEM reduction collectives that exploits memory locality, then quantifies the performance improvement with a series of micro-benchmarks measurements.

Algorithms 1 and 2 juxtapose two different implementations of the OSU sum reduce benchmark. The first algorithm shows the most relevant snippet of the original code, which must initialize two separate pSync and pWrk arrays to avoid synchronization issues when issuing back-to-back collectives. The second algorithm uses the predefined SHMEMX_TEAM_HOST and SHMEMX_TEAM_LEADERS teams to first conduct the reduction in shared memory, then to reduce that result across all host leader PEs. Note that when using the teams API, there is no need to specify the pSync or pWrk arrays, because they are managed by the implementation, as exhibited by Figure 2 and described in Section IV-C.

Figure 4 (left) compares the measured latency of the original OSU implementation in Algorithm 1 with the teams implementation in Algorithm 2 using the recursive doubling algorithm. The most salient feature of this graph is that the latency of the original implementation worsens as the total number of PEs increases. On the other hand, the host-memory with leader teams implementation shows excellent scaling. The reason for this behavior is simply that the teams implementation performs as much local communication and computation as possible before reducing remote data. This lowers the number of PEs participating in remote communication to the number of leaders, or equivalently, to the number of compute nodes in the execution, thereby improving the overall latency.

Figure 4 (right) shows measurements from the same experiment as Figure 4 (left), but instead invokes the ring reduction algorithm (described in Section IV-D). This algorithm also exhibits relatively poor scaling in the original implementation when compared to the host/leader teams variant. As predicted in Section IV-D, we see that the ring algorithm incurs a much higher latency for small buffer sizes when compared to the recursive doubling algorithm, whereas the ring algorithm performs far better than the recursive doubling algorithm at large buffer sizes. This effect is exacerbated in the original OSU implementation, but is still present when compared to the host/leader teams variant.

Figure 5 shows the same experiments as Figure 4, but instead on the Cori system. Unlike the Diamond measurements, we took care on Cori to reduce execution time variability, which has been observed to be quite high on Cray XC systems [36]. We increased the number of iterations to be 10 times larger than the default values of the OSU microbenchmark. Additionally, we launched only a single 10-node job and gathered all 1-10 node curves within that single job. The overall behavior exhibited by these measurements is consistent with the Diamond data: the original implementation shows relatively poor scaling compared to the host/leader teams implementation, and the ring algorithm shows relatively better latency with larger buffer sizes.

VI. Conclusion and Future Work

This work motivates the need for a programming abstraction that groups processes into teams using the OpenSHMEM model. We presented the latest rendition of the OpenSHMEM teams API and described a preliminary implementation of teams within the Sandia OpenSHMEM library. Using an OSU reduction microbenchmark, we show that considerable
application-level optimizations (2-16x improvement, depending on the node count and buffer size) are possible with simple changes to the program, and without requiring the user to initialize and specify synchronization and work buffers. We compare the ring and recursive doubling reduction algorithms using this microbenchmark, and see that recursive doubling outperforms ring with smaller buffer sizes, whereas ring outperforms recursive doubling with larger buffer sizes. Both algorithms result in significantly lower latency when reductions on a shared host memory team are followed by reductions across a leader team. Because a majority of HPC applications utilize collective operations [35], many real-world applications would directly benefit from such teams-based collective optimizations.

Throughout this work, several observations suggest possible improvements to be considered in future work. For instance, our implementation has several shortcomings that could be easily addressed with more development time. As an example, our implementation can support more simultaneous teams by supplying a longer availability bitmask during team creation. Our implementation also allocates a single pSync array to each team for non-barrier collective operations. Increasing the number of pSync arrays allocated to a team and keeping track of when a pSync array is no longer in use at remote PEs can eliminate barrier operations used to synchronize these arrays when performing collective operations.

Further, we have only presented results for two simple reduction algorithms (recursive doubling and a ring), but there are other more subtle features of these algorithms that are not fully exploited by our implementations. In fact, the ring algorithm does not require synchronization between back-to-back collectives, but our implementation performs a synchronization to avoid race conditions pertinent with the other collective algorithms. Additionally, several more sophisticated algorithms exist that should provide better overall scaling properties (such as Rabenseifner’s algorithm [28]). The result of our OSU sum reduction experiment also strongly suggests that the internal SOS reductions would benefit from a similar host/leader style reduction - incorporating this optimization into SOS itself is a clear next step.

Finally, the teams API proposal is not yet ratified by the OpenSHMEM specification committee, so details about the interface may change in the near term. We refer the interested reader to the teams and collectives working group within the OpenSHMEM specification committee for the latest developments on the teams API proposal [10].

ACKNOWLEDGMENT

The authors of this paper cannot take much credit for the design of the proposed OpenSHMEM teams API. We would like to sincerely thank the leaders of the teams proposal, Megan Grodowitz and Pavel Shamis, for their dedication in designing the API and driving the documentation through the OpenSHMEM standards process. We would also like to thank Nicholas Park, who has dedicated a considerable amount of assistance in designing the teams API and preparing the proposal document. Finally, a big thanks to the all members of the OpenSHMEM Specification Committee, especially to the committee secretary, Manju Gorentla Venkata, and to the committee chair, Steve Poole.

REFERENCES


A. Abstract

We explain how to obtain and build the software used in the performance measurements throughout the paper. The measurements were taken on two systems: Diamond, an internal cluster at Intel Corporation, and Cori, a Cray XC40 supercomputer at NERSC.

B. Description

1) Checklist (artifact meta information):

- Algorithm: Team creation, collective bitwise and sum reductions using ring and recursive doubling algorithms.
- Program: C/C++ code using the OpenSHMEM and libfabric communication libraries.
- Compilation: GNU GCC version 4.8.5 (Diamond), GNU GCC version 8.2.0 (Cori).
- Run-time environment: Red Hat Enterprise Linux Server release 7.5 (Maipo) with Linux kernel 3.10.0-862.el7.x86_64 (Diamond), Cray Linux Environment based on the SUSE Linux Enterprise Server Linux distribution (Cori).
- Hardware: Intel® Xeon® Scalable Processors (Skylake and Haswell), Intel® Omni-Path 100 Series Fabric with a fat-tree topology, Aries Interconnect with a dragonfly topology.
- Execution: Via shell scripts and the SLURM workload manager.
- Output: Execution time measured via gettimeofday() system time function, extracted from log files.
- Experiment workflow: see below.
- Experiment customization: see below.
- Publicly available?: Yes, this work is derived from Sandia OpenSHMEM (SOS), which is freely available via Github under a permissive software license.

2) How software can be obtained (if available):

The software can be obtained from Github here: https://github.com/davidozog/sandia-shmem/tree/wip/teams.

3) Hardware dependencies: Diamond contains a 2-socket Intel® Xeon® Platinum 8170 (Skylake) CPU at 2.1 GHz and 192 GB of DDR4 2666 MHz RAM and an Intel® Omni-Path 100 series fabric (Intel® OPA) with a fat-tree topology. Cori contains a 2-socket Intel® Xeon E5-2698 v3 (Haswell) processor at 2.3 GHz with 128 GB of DDR4 2133 MHz RAM and a Aries interconnect with a dragonfly topology.

4) Software dependencies:

- Libfabric (https://github.com/ofiwg/libfabric) version 1.7.0 (Diamond) and/or version 1.8.x (Cori)
- PSM2 (https://github.com/intel/opa-psm2) (git hash 816c0dbd)
- Hydra 3.2.1 launcher (http://www.mpich.org/static/downloads/3.2.1)
- OSU Micro-Benchmark collection version 5.6.2 (http://mvapich.cse.ohio-state.edu/benchmarks)
- Sandia OpenSHMEM wip/teams branch (git hash e26f0e95)

C. Installation

For the experiments over Intel® Omni-Path 100 series fabric, users should obtain and install the PSM2 library and libfabric. Users can follow the default build instructions provided in the README. To install libfabric over PSM2, users should add the --enable-psm2 flag during the configure step, while disabling all the other providers. Further documentation on these build instructions can be found on the SOS wiki page entitled, “Intel® Omni-Path 100 (PSM2) Build Instructions” (https://github.com/Sandia-OpenSHMEM/SOS/wiki/Intel%C2%AE-Omni-Path-100-(PSM2)-Build-Instructions). For experiments over the Aries interconnect, users should configure libfabric with the --enable-gni flag, while disabling the other flags. Instructions on this setup can be found on the SOS wiki page entitled, “Cray XC Build Instructions” (https://github.com/Sandia-OpenSHMEM/SOS/wiki/Cray-XC-Build-Instructions).

After the installation of libfabric with appropriate provider enabled, users can install the Sandia OpenSHMEM library. On Diamond, we enabled the flags --enable-pmi-simple, --enable-thread-completion, and --enable-manual-progress when running the Autotools-generated “configure” script; and on Cori, we enabled the --with-pmi=/opt/cray/pe/pmi/default, --with-xpmem, --enable-ofi-mr=pmi/default, --enable-completion-polling configuration flags. Users can follow the more specific SOS build instructions provided in the above links for the appropriate interconnect.

D. Experimental Workflow

All experiments in the paper were run using the SLURM workload manager, a single process per physical core, passing --bind-to=socket to the Hydra process launcher (on Diamond) or --cpu-bind=sockets to the srun launcher (on Cori), setting FI_PSM2_PROG_INTERVAL=1 (on Diamond only), setting SHMEM_REDUCE_ALGORITHM to either “ring” or “recdb”, and all other parameters set to their default values.

E. Experiment customization

When run on Cori, the OSU sum reduce experiment was modified to execute 10x more iterations than the default configuration to reduce variation in the timing measurements.

F. Notes

The modifications made to the OSU sum reduce benchmark to use the OpenSHMEM teams API are not available online, but they are shown verbatim in the paper (Algorithm 2). Also, the simple team creation benchmark (from Figure 3) is not publicly available on Github; but if need be, it can be provided upon request.