Extending the OpenSHMEM Memory Model to Support User-Defined Spaces

Aaron Welch  
Computer Science  
Department  
University of Houston  
4800 Calhoun Rd.  
Houston, Texas USA  
dawelch@uh.edu

Swaroop Pophale  
Computer Science  
Department  
4800 Calhoun Rd.  
Houston, Texas USA  
spophale@uh.edu

Oscar Hernandez  
Computer Science and  
Mathematics Division  
Oak Ridge National Laboratory  
Oak Ridge, Tennessee USA  
oscar@ornl.gov

Stephen Poole  
Computer Science and  
Mathematics Division  
Oak Ridge National Laboratory  
Oak Ridge, Tennessee USA  
spoole@ornl.gov

Pavel Shamis  
Computer Science and  
Mathematics Division  
Oak Ridge National Laboratory  
Oak Ridge, Tennessee USA  
shamisp@ornl.gov

Barbara Chapman  
Computer Science  
Department  
4800 Calhoun Rd.  
Houston, Texas USA  
chapman@uh.edu

Stephen Poole  
Computer Science and  
Mathematics Division  
Oak Ridge National Laboratory  
Oak Ridge, Tennessee USA  
spoole@ornl.gov

ABSTRACT

OpenSHMEM is an open standard for SHMEM libraries. With the standardisation process complete, the community is looking towards extending the API for increasing programmer flexibility and extreme scalability. According to the current OpenSHMEM specification (revision 1.1), allocation of symmetric memory is collective across all PEs executing the application. For better work sharing and memory utilisation, we are proposing the concepts of teams and spaces for OpenSHMEM that together allow allocation of memory only across user-specified teams. Through our implementation we show that by using teams we can confine memory allocation and usage to only the PEs that actually communicate via symmetric memory. We provide our preliminary results that demonstrate creating spaces for teams allows for less consumption of memory resources than the current alternative. We also examine the impact of our extensions on Scalable Synthetic Compact Applications #3 (SSCA3), which is a sensor processing and knowledge formation kernel involving file I/O, and show that up to 30% of symmetric memory allocation can be eliminated without affecting the correctness of the benchmark.

1. INTRODUCTION

The OpenSHMEM library API follows the Partitioned Global Address Space (PGAS) programming model to support communication, synchronisation and other operations between processing elements (PEs) executing C, C++, or Fortran SPMD programs. Other useful operations provided by the OpenSHMEM library include calls for collective operations (symmetric memory allocation, broadcast, reduction, collection and synchronisation), atomic memory operations, distributed locks and data transfer ordering primitives (fence and quiet). All functionality provided by the current 1.1 OpenSHMEM specification is identical to the original SGI SHMEM library, which OpenSHMEM was based on.

OpenSHMEM provides collective calls for operations such as symmetric memory allocation, synchronisation, and data transfer operations. Most of these calls are collective over a subset of PEs that are defined by an active set which is defined by a triplet of parameters within each collective call.

As of OpenSHMEM Specification 1.1, allocation of symmetric memory using library calls such as shmalloc or shpalloc has the following limitations:

1. All PEs must participate in the call.
2. All PEs have the data object allocation even if they may not use it.
3. Implicit synchronisation at the end involving all PEs.

The above limitations can amount to a significant waste of time and space if only a fraction of the total number of PEs use a particular symmetric data object. It also means that shmalloc may not be called inside functions or methods that are not called by all PEs at the same point in execution.

Historically, there have been two primary elements of a particular communication operation in OpenSHMEM - the target PE (or multiple PEs in the case of collectives) and where the memory of interest is located. While these have been separate components in past iterations of the specification, they have both suffered from the same kinds of constraints imposed by the emphasis on a global view. However, while
the former issue is resolved by offering ways to work with well-defined subsets of the same operating set, the latter issue demands instead widening the scope of the current OpenSHMEM memory model to include any number of additional and disparate memory regions.

The most critical piece at the centre of all OpenSHMEM communication is memory and how it is handled. As such, it is important to carefully consider what features and constraints may be best suited for supporting memory management in OpenSHMEM applications. To facilitate a fine-grained work distribution, the requirement for the involvement of other potentially unrelated PEs in symmetric memory allocation must be removed.

Our main contribution is in changes to the current memory model of OpenSHMEM to add memory context to teams called spaces and to provide a simple concise API for this to the OpenSHMEM specification. The main advantage of spaces is that memory allocation does not need to be collective across all PEs but only by the PEs included in a particular team, which makes it a scalable alternative by eliminating synchronisation across all PEs. Spaces enable PEs in a team to communicate via symmetric variables allocated only by the PEs in the team and can be looked at as a mechanism for allocation of data objects that are symmetric only with respect to the team.

Providing spaces to teams of OpenSHMEM PEs is also a way of providing isolation for different work-sharing teams. This is a step towards making the OpenSHMEM memory model compatible with other libraries. Use of spaces may allow memory allocation within other libraries for communication using OpenSHMEM without the risk of interfering with user applications.

Going ahead, building the concept of teams with associated spaces may simplify debugging memory related errors; since spaces are associated with specific teams of PEs, errors that occur within particular spaces benefit from the more narrowly defined scope that the teams associated with them provide. This also means that problems within the context of one team/space are relatively isolated from the rest of the application, minimising their impact as well. Other significant implications could be in multi-threading environments where teams could be groups of threads. This could allow for a more natural way of providing the extra context that such threading models require, but is out of the scope of this paper and so is left for future work.

Heterogenous computing using accelerators is another venue which maps well with our concept of teams and spaces. Keeping in context the amount of memory available for the CPU and the accelerators and the types of computations that are off-loaded to the accelerators, applications could provide fine-tuned memory allocations that enable efficient implementation of the OpenSHMEM specification on hybrid platforms.

The rest of the paper is organised as follows. Section 3 describes existing programming models and libraries that have evolved to incorporate non-collective memory allocation for remote memory operations. For better understanding of the OpenSHMEM environment we provide background information in Section 2 and describe the design of the API in Section 4. In Section 5 we evaluate the proposed framework using micro-benchmarks and the SSCA3 kernel. In Section 6 we summarise our approach and highlight its key contributions as well as its drawbacks and look at the possible avenues of future development.

2. BACKGROUND

There are two categories of variables that can be used in an OpenSHMEM program, namely, symmetric and non-symmetric or local data. In OpenSHMEM, communication is possible only via symmetric data objects. Since the OpenSHMEM library provides bindings for the C, C++ and Fortran languages, certain variable classes within each language are defined as symmetric. The library implementation has to ensure that these variables are accessible to other PEs executing the same OpenSHMEM application.

By definition, symmetric data consists of variables that are allocated by the OpenSHMEM library (using special memory allocation calls) or defined as global or static in C/C++ or in common or save blocks in Fortran. These variables have the same name, size, type, and offset address on all PEs. The library calls shmalloc (C/C++) and shcalloc (Fortran) allocate symmetric data objects on the symmetric heap, which is remotely accessible by every other PE. Symmetric data allocation is a collective process and the OpenSHMEM specification requires that it occurs at the same point in the execution path of all PEs with identical size values. The code in Listing 1 illustrates the different symmetric variable categories in a C program using OpenSHMEM.

Listing 1: Variable Categories in OpenSHMEM

```
1 int aglobal; /*symmetric variable*/
2
3 void main(void)
4 {
5     int *x;
6     int me, npes, y;/*local variables*/
7     static int astatic;/*symmetric variable*/
8     start_pes(0);
9     /*dynamic allocation of symmetric data object*/
10    x = (int *)shmalloc(sizeof (int));
11    ...
12    shmem_int_put(&astatic, x, 1,
13                   (me+1)%npes);
14    ...
15    shmem_int_get(&y, &aglobal, 1,
16                   (me+1)%npes);
17    ...
18    shfree(x);
19    ...
20    return 0;
21 }
```

In line 1 of Listing 1 the variable aglobal is a global variable and hence symmetric. At line 7 the OpenSHMEM library is initialised and all OpenSHMEM processes must execute this call. The variable astatic declared in line 8 is also symmetric as per the OpenSHMEM specification. A symmetric variable x is allocated in line 10 using the dynamic memory allocation call shmalloc. Conceptually, the symmetric variable x is allocated the requested amount of memory at the same memory
offset on the symmetric heap for each PE. This mechanism facilitates fast remote address calculation at the source PE as the remote symmetric variable’s address can be computed by adding the base address of the symmetric heap on the target PE and the local offset corresponding to the same symmetric variable. This is managed internally by the OpenSHMEM library implementation. An OpenSHMEM library implementation may choose to have symmetric variables at the same address to speed up target destination address calculation. Symmetric variables may contain different values on all PEs.

Non-symmetric or local data is available to individual PEs only and is not visible to or directly accessible by a remote PE. Some OpenSHMEM library routines like shmem_put and shmem_get may use local variables as the source and the destination respectively.

3. RELATED WORK
Communication mechanisms for subgroups of processes have been explored by other programming models and libraries. The Message Passing Interface-2 (MPI-2) specification allows for remote memory access (RMA) within a group of MPI processes represented by a communicator through the mechanism of window creation [2]. This is a collective call executed by all processes in the group of the communicator and the window object can be used by these processes to perform RMA operations.

An initialisation operation allows each process in an intra-communicator group to collectively specify a particular window in its memory. When each process specifies this window of existing memory, it exposes it to RMA accesses by remote processes specified by the communicator. Once such a window is created it returns an opaque object that represents the group of processes that own and may access the set of windows along with the attributes of each window. Additionally, improvements to one-sided communication in MPI-3 have made it possible to implement OpenSHMEM in MPI [12]. With MPI’s strong history behind groups and communicators, this makes it particularly relevant when considering the design qualities of memory management across subsets of processes.

Co-array Fortran (CAF) 2.0 [14], which evolved and then diversified from co-array features in Fortran 2008, allows for dynamic allocation/deallocation of co-arrays and other shared data and local variables within subroutines. They also introduced the concept of CAF teams as a way to facilitate subgrouping of CAF images. CAF 2.0 allows team based declaration and allocation of co-arrays within a procedure. This allows for asymmetric allocation of co-arrays and removes the need for global synchronisation across all CAF images, thus providing an isolated domain for team members to communicate and synchronise [26].

Unified Parallel C (UPC) [3] also allows for dynamic allocation of shared memory, and the function can either be collective or not. Functions like upc_all_alloc and upc_all_dealloc are not required to be called by all threads. If upc_all_alloc is called by multiple threads, all threads making the call get different allocations, while upc_all_dealloc returns a pointer to shared space where the space has affinity to the calling thread. In contrast, upc_all_alloc provides a mechanism for collective allocation of shared space. The arguments passed to this function must be identical on all threads. At the end of its use, shared memory allocated via upc_all_alloc must be deallocated by calling upc_all_free.

X10 [6] introduced features that were motivated by high-productivity, high-performance parallel programming for non-uniform cluster systems. Places and activities are two such concepts. Places in X10 are a virtual collection of resident non-migrating data objects, while activities operate on the resident/local data [19]. Every X10 activity runs in a place. Objects and activities are bound to a particular place, but places may migrate across physical locations. This is motivated by affinity and load balance considerations. Communication using remote data is different than the other PGAS programming languages. In X10, an activity may read/write remote variables only by spawning an activity at the remote place. This allows for different activities to perform accesses to remote variables. Unless it is explicitly enforced by the programmer, inter-place data accesses have weak ordering semantics (to eliminate synchronisation overhead).

Chapel [4] has the concept of domains and locales [5] that represent ways to map data and units of the target system architecture respectively. Chapel’s PGAS memory model allows tasks executing within a given locale to access lexically visible variables whether they are allocated locally or on a remote locale. Locales also facilitate the creation of global-view, distributed arrays. Chapel defines domains to drive loops and to declare and operate on arrays. This is done through a domain map. In the absence of a domain map, an array’s elements are mapped to the current locale. Domain maps can target a single local locale (layout) or multiple locales to store distributed index sets and arrays (distributions).

Global Address Space Programming Interface (GASPI) [11] is an API specification for PGAS applications. GASPI defines groups as subsets of processes/ranks, and collective operations are restricted to the ranks forming the group. Segments are defined for RMA and can be allocated in hierarchical memory regions. Segments are globally accessible from every thread of every GASPI process and represent the partitions of the global address space [8]. Within a segment, memory addresses are specified by the triple consisting of the rank, segment identifier, and offset.

PGAS as well as the message passing paradigm has explored the challenges of allocation, placement and access to data by subgroups of processes. Although the approaches discussed use different ways to express subgrouping and memory accessibility, the main objective is to provide a better and more flexible memory abstraction for subsets of processes. There is an obvious need for these concepts in the current OpenSHMEM specification and we hope to provide a solution in terms of the new features and a minimalistic API to use these features. We look at all the advantages and disadvantages of the concepts and their implementations provided by other PGAS languages/libraries and MPI to design our proposal for teams and spaces for OpenSHMEM.
4. DESIGN
To address the concerns raised in Section[1] we are proposing additions to the current OpenSHMEM programming model, with a special focus on diversifying the memory model to include symmetric memory regions that exist only with respect to a particular team through the use of spaces. This represents a big change in the memory model of the OpenSHMEM library. Figure[1] shows the conceptual memory model when using the OpenSHMEM library (as defined by Specification 1.1 and earlier). The new model depicted in Figure[2] visually demonstrates how allocation with spaces allows for the creation and use of memory that is symmetric only with respect to particular teams.

![Figure 1: Memory Model as per OpenSHMEM Specification 1.1](image1)

![Figure 2: New Memory Model with Spaces](image2)

Prototypes of these extensions were implemented in the OpenSHMEM reference implementation[17] using Universal Common Communication Substrate (UCCS)[15] as seen in[20].

4.1 Teams
We introduced the teams extension in[16] and now expand the concept by providing teams within teams, explicit indexing of members of a team and providing a team split functionality. Traditional active sets in OpenSHMEM were implicitly defined as part of collective calls and did not conceptually live outside of individual calls. Contrary to this, teams are defined externally and are reusable objects. In an abstract sense, teams are simply a way to select a subset of PEs to operate with respect to.

4.1.1 Design of Teams
Creation of teams is performed through separate calls that specify which PEs to include in the definition, and return an opaque handle to the team that can be passed through to later functions that will use it. All teams are created from others as subsets of the PEs in their parent. There are two predefined teams available throughout an OpenSHMEM application’s execution. SHMEMX_WORLD_TEAM is a team which includes all PEs available to the OpenSHMEM application in the system. If the criteria for team creation can not be met or will not result in the successful selection of any PEs, then the predefined SHMEMX_NULL_TEAM is returned instead.

Upon creation, all valid members of a team are given new PE index values from 0 to n − 1 with respect to the new team. The creation process is not collective; it is a completely local operation. However, all PEs included in the selection must create the team before being able to successfully complete any collective calls on that team. It is not necessary to ensure that only the PEs in the definition of a team attempt to create it - membership in a team can and should be queried through the use of an API call on that team.

Different types of teams can be created by calling creation functions specific to that type, while all subsequent operations employ a shared syntax for all types. We focus on two such kinds of functions in this paper - strided sets and the creation of axial splits. The design of other types of teams is beyond the scope of this paper and is considered in future work.

Strided teams are defined similarly to active sets in OpenSHMEM 1.1, except that here the stride may be any generic value instead of being logarithmic (base 2). The axial splits are unique in that they produce multiple teams describing portions of the larger parent, rather than just one describing all desired PEs. These work by defining how to arrange the parent’s PEs in a dimensional space so that it may be split up along each axis, using the calling PE as the origin. This effectively allows developers to operate on particular rows or columns of a computational space. There are three functions for accomplishing this - two, three, and n-dimensional splits. The former two are simply instances of the n-dimensional split created for convenience, the latter of which results in the creation of n teams each consisting of all PEs along one of the n dimensions (again, using the current PE’s position in the dimensional space as the origin of the split). While the presence of the n-dimensional split technically obviates the need for the other two functions, it can be slightly more difficult to understand and use, while the other convenience functions provide a quick and easy interface that will be enough to satisfy most use cases. Since all new extensions are required to use the shmemx namespace, we design our proposed API accordingly. Table[1] lists the proposed creation functions.

The code shown in Listing[3] shows the grouping created with a 2-D axial split, which is represented visually in Figure[2]. Here, each PE receives two teams as a result of the call:
Create a strided team. shmemx_create_strided_team(long PE_start, long PE_stride, long PE_size, shmemx_team parent, shmemx_team *subteam);

Create a 2-D split. shmemx_team_2d_split(long xrange, long yrange, shmemx_team parent, shmemx_team *xaxis, shmemx_team *yaxis);

Create a 3-D split. shmemx_team_3d_split(long xrange, long yrange, long zrange, shmemx_team parent, shmemx_team *xaxis, shmemx_team *yaxis, shmemx_team *zaxis);

Create a n-D split. shmemx_team_axial_split(long dimensions, long *range, shmemx_team parent, shmemx_team **axes);

Table 1: Team Creation API

<table>
<thead>
<tr>
<th>Team Creation API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a strided team. shmemx_create_strided_team(PE_start, PE_stride, PE_size, shmemx_team parent, shmemx_team *subteam);</td>
</tr>
<tr>
<td>Create a 2-D split. shmemx_team_2d_split(xrange, yrange, shmemx_team parent, shmemx_team *xaxis, shmemx_team *yaxis);</td>
</tr>
<tr>
<td>Create a 3-D split. shmemx_team_3d_split(xrange, yrange, zrange, shmemx_team parent, shmemx_team *xaxis, shmemx_team *yaxis, shmemx_team *zaxis);</td>
</tr>
<tr>
<td>Create a n-D split. shmemx_team_axial_split(dimensions, *range, shmemx_team parent, shmemx_team **axes);</td>
</tr>
</tbody>
</table>

Table 2: Team Query and Destruction API

<table>
<thead>
<tr>
<th>Team Query and Destruction API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find the index in PE src. shmemx_team_translate_index(int dst for PE in src, PE, shmemx_team src, shmemx_team dst);</td>
</tr>
<tr>
<td>Get the size of a team. shmemx_team_size(shmemx_team team);</td>
</tr>
<tr>
<td>Destroy a team. shmemx_destroy_team(shmemx_team team);</td>
</tr>
</tbody>
</table>

4.1.2 Prototype Implementation

The SHMEMX_WORLD_TEAM has a lifetime of the entire OpenSHMEM application, hence a new team may be created from the world team at any point. Since PE ids within this team are globally consistent across all processes, a new team simply needs to have a mechanism for reindexing PE ids with respect to itself to ids in this world view. These ids can be calculated in a deterministic fashion with a translation function, obviating the need to store more information than a few input values to this function, and recalculating these translations when necessary. Thus, storage requirements for a team are constant for a particular type, incurring only minimal memory overhead.

For strided teams, this is especially easy - for a start id of a and a stride of s, a PE in the team with an index of x has a world index value of y = sx + a. Handling of the axial splits does not require a new translation function, as the results of an n-dimensional split can be produced by simply finding the appropriate deltas between adjacent PEs along a given dimension and using those to create n strided teams. This otherwise internal translation function is exposed through the shmemx_team_translate_index() function, with the added requirement that the destination team may need to be searched when not equivalent to the world team. The code in Listing 3 illustrates the use of the proposed features:

Listing 2: 2-D Axial Split

```
Create a strided team. shmemx_create_strided_team(long PE_start, long PE_stride, long PE_size, shmemx_team parent, shmemx_team *subteam);

Create a 2-D split. shmemx_team_2d_split(long xrange, long yrange, shmemx_team parent, shmemx_team *xaxis, shmemx_team *yaxis);

Create a 3-D split. shmemx_team_3d_split(long xrange, long yrange, long zrange, shmemx_team parent, shmemx_team *xaxis, shmemx_team *yaxis, shmemx_team *zaxis);

Create a n-D split. shmemx_team_axial_split(long dimensions, long *range, shmemx_team parent, shmemx_team **axes);
```

Figure 3: Visual Representation of 2-D Axial Split in Listing 2

Query functions are provided for use with teams. These new API calls are listed in Table 2. The most important of these is shmemx_team_translate_index(), which translates the index of a PE in a source team to a destination team. If there is no valid translation, it returns a negative value (error). This can be used to get a PE's own id within a team as well as determine membership in a team, both of which can be seen demonstrated in Listing 3. Additionally, shmemx_team_size() returns the size of a team. When finished with a team, it can be destroyed via the shmemx_destroy_team() call. This call is also not collective, but is required for all PEs that created it. In addition to these new functions, all point to point and collective calls can be used with teams, and be made to make use of the indexing provided by them.

For strided teams, this is especially easy - for a start id of a and a stride of s, a PE in the team with an index of x has a world index value of y = sx + a. Handling of the axial splits does not require a new translation function, as the results of an n-dimensional split can be produced by simply finding the appropriate deltas between adjacent PEs along a given dimension and using those to create n strided teams. This otherwise internal translation function is exposed through the shmemx_team_translate_index() function, with the added requirement that the destination team may need to be searched when not equivalent to the world team. The code in Listing 3 illustrates the use of the proposed features:

```
int main(int argc, char **argv) {
    int me, team_index;
    shmemx_team strided, *axis;
    ... me = shmem_my_pe();
    /* creates a team containing */
    /* PEs 1, 4, 7, ..., 82 */
    shmemx_create_strided_team(1, 3, 27, SHMEMX_WORLD_TEAM, &strided);
    team_index = shmemx_team_translate_index(me, SHMEMX_WORLD_TEAM, strided);
    if (team_index > 0) {
        long range[3] = {3, 3, 3}
        shmemx_team_barrier(strided, pSync);
        if (team_index == 0)
            printf("strided team completed barrier!\n");
        shmemx_team_axial_split(3, range, strided, &axis);
        for (i = 0; i < 3; i++) {
            team_index = shmemx_team_translate_index(me, SHMEMX_WORLD_TEAM, axis[i]);
            for (j = 0; j < shmemx_team_size(axis[i]); j++)
                /*add 1 to symm_var on all*/
                /*PEs along axis i*/
                shmem_int_add(symm_var, 1,
                                shmemx_team_translate_index
```
(j, axis[i], SHMEMX_WORLD_TEAM));
shmemx_destroy_team(axis[i]);
}
shmemx_destroy_team(strided);

Listing 3: Creation and Use of OpenSHMEM Teams

4.2 Spaces
Spaces are regions of memory allocated only by a chosen team of PEs. They represent a shift in the underlying memory model and assumptions about memory allocation and management from those found in OpenSHMEM 1.1. Past iterations of OpenSHMEM have relied on a single globally accessible and symmetric heap for all dynamic memory management for communication operations. To make this less expensive and involving for operations specific to only particular subsets of PEs and to increase the flexibility and programmability of OpenSHMEM, spaces were designed to create and provide access to memory exclusive to only those subsets of PEs. The primary such use of this in this paper is to generate additional symmetric heaps for use within the PEs of particular teams of interest.

4.2.1 Design of Spaces
The most obvious use of user-defined spaces would be to mimic the global symmetric heap, but extend its design to work with individual teams. Similar to how the traditional global symmetric heap is designed, this results in equal portions of the overall memory being located within each of the members of the assigned team in the form of n individual spaces across a team of size n. Together, these spaces and the interface used to manipulate and incorporate them into the rest of the API give the same illusion of a unified shared memory region historically provided by the global symmetric heap. To this end, we introduce a new call to create such a space and assign a team to it. All allocation and freeing of memory within the space is subject to the same rules as the global symmetric heap traditionally has been, with the only difference being that those rules apply strictly to the members of the team assigned to it. The complete list of proposed functions for spaces is listed in Table 3. Creating a space and assigning a team to it can be accomplished through the use of the shmemx_create_symmetric_space() call. Assigning a team to a space is a collective operation across all PEs in the team. Once a team is assigned, the space is immediately available for use upon return of the call. Furthermore, memory allocated within a space is available to all PEs in the team that was assigned to it, meaning that subteams or other such groupings that also contain any of the same PEs will likewise have access to the memory within for those specific PEs. Additionally, destroying a space can be accomplished with shmemx_destroy_space() and is a collective operation across the team originally assigned to it and results in freeing all the allocated memory associated with it. The space is guaranteed to be remotely accessible until all pending communication using it is complete.

After a space is destroyed, use of any such memory or the space itself is invalid and results in undefined behaviour. Destruction of a space is an explicit operation - destroying a team will not result in the destruction of any memory that team may have had access to. This is partly a result of the fact that other teams may still be using the space, as well as to allow the user to control when a potentially costly operation should occur, as destruction of a team is cheap but destruction of a space may be expensive due to incomplete operations and synchronisation. Memory can be allocated and freed within a space using the shmemx_malloc_space(), shmemx_realloc_space(), and shmemx_free_space() calls.

<table>
<thead>
<tr>
<th>Space Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creates a symmetric space and assigns team to it.</td>
<td>shmemx_create_symmetric_space( space and assigns team to it.</td>
</tr>
<tr>
<td>Allocate a symmetric data object within space.</td>
<td>shmemx_malloc_space(size_t, size, shmemx space space);</td>
</tr>
<tr>
<td>Changes the size of memory at ptr within space.</td>
<td>shmemx_realloc_space(void *ptr, size_t size, shmemx space space);</td>
</tr>
<tr>
<td>Frees the memory at ptr within space.</td>
<td>shmemx_free_space(void *ptr, shmemx space space);</td>
</tr>
<tr>
<td>Destroys a space and all memory allocated in it.</td>
<td>shmemx_destroy_space( shmemx space space);</td>
</tr>
</tbody>
</table>

Table 3: Spaces API

4.2.2 Implementation using UCCS
We implemented the concepts of teams and spaces using the OpenSHMEM reference implementation [17] using UCCS. UCCS is a communication middleware which aims to provide a high performing low-level communication interface for implementing parallel programming models. It is designed to minimise software overheads and provide direct access to network hardware capabilities without sacrificing productivity. It has already been integrated into the OpenSHMEM reference implementation in [20].

The performance characteristics of different implementations of spaces have the potential to vary quite widely. The primary implementation characteristic of the chosen implementation strategy is that of imitating the necessary steps for creation of the original global symmetric heap. This requires the creation of completely separate and independent memory regions whose setups result in self sufficient management. Some RDMA networks may require memory regions to be registered and access keys to be generated and used before remote access may be granted. The UCCS middleware abstracts these networks into a single interface that may be used to access them through the use of opaque registration handles. However, these registration handles must still be manually managed within OpenSHMEM in order to access the associated memory.

While these regions that must be registered to the network and the memory within spaces do not need to be directly related, it is far simpler to handle management in this manner and issue memory regions in static chunks. Before communication with respect to the new space can be performed, these registrations need to be shared with other PEs in the team.
The code shown in Listing 4 demonstrates the use of spaces with a PE, then an access to a specific data object may lead to a search across all such spaces to find which one the data object belongs to. While this could be done efficiently, it may not be necessary, and other methods to avoid this search are left for future work.

In addition to the user portion of the space, we allocate a small amount of extra memory within the same registered region for use as a synchronisation array (pSync). This is necessary in order to satisfy the requirements of symmetric memory allocation, as it is required for each allocation to perform a barrier at the end to ensure completion. All allocations on a particular space will use this pSync for performing synchronisation across the assigned team, but this is neither exported nor a general replacement for pSync in user collectives.

It is not strictly necessary to include spaces as an extra parameter to communication calls, as a particular data object will only belong to a single space. However, omitting them creates the possibility that if multiple spaces are associated with a PE, then an access to a specific data object may lead to a search across all such spaces to find which one the data object belongs to. While this could be done efficiently, it may not be necessary, and other methods to avoid this search are left for future work.

The code shown in Listing 4 demonstrates the use of spaces to allocate a symmetric data object specific to a particular team.

```c
shmemx_team team;
shmemx_space space;
shmemx_create_strided_team(from, to - from, 2, SHMEMX_WORLD_TEAM, &team);
```

5. EVALUATION

The evaluation of this implementation was conducted on an SGI Altix XE1300 system located at the Oak Ridge National Laboratory’s Extreme Scale System Center. The system consists of 12 compute nodes, each with two Intel Xeon X5660 CPUs for a total of 12 CPU cores and 24 threads. Compute nodes are interconnected with Mellanox’s ConnectX-2 QDR HCA with one port. We installed the OpenSHMEM reference implementation from [20] with the new extensions included, using UCCS version 0.5. All tests were run with increasing numbers of PEs up to 128.

5.1 Testing with Micro-benchmarks

We first evaluated our work with a couple of micro-benchmarks to show the basic costs of creating and using spaces with teams. As discussed before, to use spaces for communication we need to first create them, communicate the relevant information across all PEs in a team, and then allocate symmetric data within the new space. Each of these tests were repeated 5000 times before recording the median values, and each iteration allocated new memory without releasing any old memory to ensure that the same memory locations didn’t get used repeatedly.

Figure 4 shows the time taken to communicate the different parameters (registration handles, addresses, etc.) associated with the new space. At first glance these numbers seem high but there are two factors to consider: the first is that this is a one time cost and multiple symmetric data object allocations are possible from the same space, and second is that being only an initial implementation, there is likely to be plenty of room for improvement. This is effectively the only new cost associated with allocating memory with spaces compared to the global heap, as the `shmemx_malloc()` function works the same as `shmalloc()`, with the exception that the synchronisation will only occur across a subset of processes.

Figure 5 shows the time required for allocating an integer in the new space. At present we follow similar semantics as that of `shmalloc` where all PEs in the team have to synchronise before any PE returns from the call. If this requirement is eliminated, better performance could be obtained. The reason that only a single int is allocated is due to the fact that the only difference in performance will come from the smaller synchronisation cost; the number of PEs involved has no impact on the performance of different allocation sizes. For especially large allocations, this performance difference will become less significant. We observe that a `shmalloc` takes 15.69 micro-seconds for allocating an integer across 128 PEs. Fewer PEs allocating the symmetric data object could translate to space savings as well as time savings, since
only a fraction of the total number of PEs will be involved in the synchronisation at the end. Using spaces for allocation rather than rely on the global heap already implies some amount of space savings dependent on the fraction of PEs that no longer need to be involved. However, if the total synchronisation time for all allocations on a space in addition to the creation time for that space is less than the time needed for synchronisation across all PEs for the equivalent number of allocations from the global heap, then time savings can be achieved as well.

Another aspect to note is that of any extra overhead incurred by a given implementation of spaces. The implementation we proposed does not have much overhead besides the need to store registration handles and start addresses for the various memory regions. The amount of memory required for this is architecture dependent but typically very small. Other implementation strategies may be more efficient in other ways, but may require a slightly higher overhead.

With our spaces approach we were able to identify and eliminate unnecessary use of symmetric variables. The Table 4 shows the symmetric memory required by the benchmark when using OpenSHMEM 1.1 and when using teams with spaces. We see an average saving of 30% of the symmetric space. For OpenSHMEM applications that are required to have a small memory footprint, using teams and spaces could yield significant benefits.

6. CONCLUSIONS AND FUTURE WORK

In this paper we presented two extensions to the OpenSHMEM specification that enable subsets of PEs and allocation of symmetric memory across these subsets, and provided details for a prototype implementation. Implementing spaces can be difficult, and the creation process can be potentially expensive due to overheads that may be imposed by network hardware and the information that needs to be exchanged between interested PEs. The advantages of this approach are that we remove the necessity of synchronising across all PEs for every symmetric object allocation, only the PEs involved in communication are required to allocate the memory needed for it, and it also becomes possible to handle more dynamic scenarios or interactions between OpenSHMEM libraries. From the kernels we see that up to 30% of the symmetric memory can be saved when we adopt the new extensions. Additionally, providing an abstraction for symmetric memory allocation makes adapting to heterogeneous architectures a simpler task for OpenSHMEM.

However, much work remains in order to achieve a more robust implementation of spaces and achieve better perfor-
Another critical piece of this problem is how to deal with a more long-term goal involves the complete reworking of the PGAS concepts of OpenSHMEM could be adapted to a new allocator and memory management strategies assist in a new selection strategies for teams are also being looked into, and may carry the potential to drastically increase the expressivity of the specification for its provided communication operations.

A more long-term goal involves the complete reworking of internal memory management strategies with the creation of a new allocator for distributed and symmetric memory. The symmetric memory allocator has traditionally been implemented as multiple individual allocators for each PE that deterministically generate the same allocations when called with the same allocation parameters in the same order. This has the benefit of being a local operation up to the barrier synchronisation, but may also have the potential for greater inefficiencies when faced with numerous instances of such allocators for each disjoint memory region. A concept that may prove useful in the creation of a new allocator is that of region-based memory. Region-based memory as seen in the works of Gay [19] and Tofte [24] offers an alternative approach to memory management within cohesive and contiguous blocks. This form of memory management also provides for efficient allocation and deallocation of entire regions at once [23], and has been demonstrated in the Myrmics allocator [13]. Furthermore, it has already been shown that this tactic can be applied successfully to PGAS settings through the use of DRASync, an allocator designed for region-based allocation and synchronisation [21].

Another critical piece of this problem is how to deal with heterogeneity in future computing systems. Not only could a new allocator and memory management strategies assist in better supporting these types of systems, but could naturally map very well to the asymmetry commonly found in them. The PGAS concepts of OpenSHMEM could be adapted to heterogeneity by using teams and spaces to reference and manipulate these separate components with their respective contexts preserved [22]. We may also be able to use some insight gained from the Asynchronous Partitioned Global Address Space model [18] in concordance with these extensions to exploit regional locality in such future systems.

7. ACKNOWLEDGMENTS

This work is supported by the United States Department of Defense and used resources of the Extreme Scale Systems Center located at the Oak Ridge National Laboratory.

8. REFERENCES


