OpenSHMEM Extensions and a Vision for its Future Direction

Stephen Poole[1], Pavel Shamis[1], Aaron Welch[2], Swaroop Pophale[2], Manjunath Gorentla Venkata[1], Oscar Hernandez[1], Gregory Koenig[1], Tony Curtis[2], and Chung-Hsing Hsu[1]

[1] Extreme Scale Systems Center
Oak Ridge National Laboratory
{spoole, shamisp, manjugv, oscar, koenig, hsuc}@ornl.gov
[2] Computer Science Department
University of Houston
{dawelch, swaroop, arcurtis}@uh.edu

Abstract. The Extreme Scale Systems Center (ESSC) at Oak Ridge National Laboratory (ORNL), together with the University of Houston, led the effort to standardize the SHMEM API with input from the vendors and user community. In 2012, OpenSHMEM Specification 1.0 was finalized and released to the OpenSHMEM community for comments. As we move to future HPC systems, there are several shortcomings in the current specification that we need to address to ensure scalability, higher degrees of concurrency, locality, thread safety, fault-tolerance, I/O, etc. In this paper we discuss an immediate set of extensions that we propose to the current API and our vision for a future API, OpenSHMEM Next-Generation (NG), that targets future Exascale systems. We also explain our rational for the proposed extensions and highlight the lessons learned from other PGAS languages and communication libraries.

1 Introduction

OpenSHMEM [1] [2] [3], an API for the Partitioned Global Address Space (PGAS) programming model, is a derivative of SGI's SHMEM API, which was originally developed by Cray. The SHMEM API is widely used in parallel applications [4] and has been adopted by multiple system vendors including IBM, Quadrics, Hewlett Packard, QLogic, and Mellanox Technologies. Although these implementations are similar in functionality and semantics, they have minor differences that inhibit portability. The *OpenSHMEM* API is an open source community effort to standardize the SHMEM API used by the scientific community and hardware vendors, jointly led by the ESSC at ORNL, and the University of Houston.

The *OpenSHMEM* API provides a complete set of concise and powerful library calls to satisfy the communication needs of parallel applications. These include collective communication operations, remote memory access (RMA),

atomic memory operations, synchronization operations, distributed lock operations, and operations to check process and data accessibility. A complete set of operations, and semantics of the operations are detailed in [1].

Although the *OpenSHMEM* API provides an adequate and complete set of operations for implementing communication libraries for the time it was developed, it requires additional functionality for the exascale era. Particularly, as the needs of exascale applications change, so does system architecture - nodes have multiple CPU sockets and computing cores with varying instruction sets and power calibrating interfaces, network interfaces provide low-latency and high bandwidth communication, native support for RMA operations, collective operation functionality, etc. To better accommodate these technological shifts, the API needs to incorporate other useful concepts either born out of *OpenSHMEM* user experience or lessons from the evolution of other parallel programming languages and communication libraries (e.g. CAF 2.0, Titanium, UPC, Chapel, MPI 3.0).

The most important changes that are critical for *OpenSHMEM* are the following:

- Non-blocking Operations: The invocation of a library call returns before the operation is complete, providing the application with the opportunity to hide the latency of the operation with additional computation.
- Fault Tolerance: The API should enable implementation of fault-tolerant and fault-aware communication libraries. In addition, it should provide adequate support for building fault resilient applications.
- Hybrid Programming: The API should support, or at least not prohibit, interoperability with other programming models. This provides the application an opportunity to use multiple programming models simultaneously to better suit architectural needs, including heteregenous systems.
- Isolation: This provides private communication contexts for groups of PEs, an important attribute for enabling the construction of libraries.
- Locality: This will help to define processing elements (PEs) and symmetric memory areas that are "next" to each other and that can be mapped to multiple devices/accelerators within a node or to nodes close to each other.

In this paper, we propose a series of extensions to the *OpenSHMEM* API that work toward adding some of the above functionality. Our work in this paper can be classified into two different categories:

1. A series of extensions that strive to maintain backward compatibility with the current *OpenSHMEM* API, while aiming to improve programmer productivity and the performance and scalability of *OpenSHMEM* applications. The extensions include 1) Explicit active sets 2) Non-blocking operations 3) Library Shutdown, and 4) Multi-threading support.
2. We make a case for a series of extensions that is far-reaching and geared more towards the needs of exascale era applications and hardware. The extensions in this category include 1) Isolation 2) Locality 3) Error Model, and 4) I/O.

The rest of the paper is organized as follows: In Section 2, we describe the motivation behind adding the proposed incremental extensions to OpenSHMEM and the potential benefit that can be gained by adding these useful features. In Section 3, we describe in detail the different extensions and their semantics with concrete examples and prototypes. We also provide a broader view of the different aspects that may greatly impact OpenSHMEM in the march towards Exascale and discuss a few nascent concepts for OpenSHMEM-NG in Section 4. Section 5 throws light on the different features that were found lacking and then added into other PGAS languages and libraries (namely Chapel, UPC, Co-array Fortran, Titanium, and MPI-3). We conclude in Section 6 by summarizing our work and highlighting the key contributions.

2 Motivation for the Incremental Extensions

In OpenSHMEM 1.0, collective operations are performed on implicitly defined active sets using a strided pattern that is restricted to powers of two. These operations are expected to be called by all members of the defined set of PEs, but for any PEs that are not in the set, calling the same operation results in undefined behavior. There are a number of limitations to this approach, including the inability to specify an arbitrary stride or to select participating PEs through other methods. It can also become cumbersome and redundant to provide the full details of the active set with each successive call, and could lend itself to error and inconsistencies. Furthermore, active sets are currently designed to exist only during the life of a particular collective call. Allowing the user to explicitly define an active set provides opportunity for reuse, which will increase programmer productivity. This is possible while still maintaining backward compability with OpenSHMEM 1.0 since active sets may be reused throughout the entire application without significantly changing the memory, execution, and synchronization models of OpenSHMEM 1.0.

Overlapping computation with communication allows for better concurrency and utilization of resources when using hardware that supports the operations. There are very tangible benefits that may be realized by making collectives, atomics, and RMA communication calls non-blocking. The time spent waiting on the results of a computation or a RMA operation may be better utilized by doing other useful work. Non-blocking calls expose potential completion latency that may be utilized by the application to execute other units of work or by hybrid programming models (i.e. OpenSHMEM plus multithreading and tasking).

For verifying the potential for overlap in the context of OpenSHMEM, we developed a working prototype for a non-blocking version of the atomic call *shmem_longlong_fadd()* called *shmem_longlong_fadd_nb()*. The prototype was implemented using the OpenSHMEM reference implementation [5] with the Universal Common Communication Substrate communication middleware (UCCS) [6] [7]. UCCS is a high-performance communication middleware that provides a broad range of semantics useful for the PGAS programming model. This prototype was put to the test using a custom version of the Communication Offload

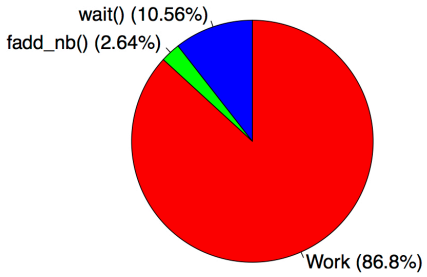

Fig. 1: Potential Overlap using Non-blocking Fetch-and-Add

MPI-based Benchmark (COMB) [8], modified to work using SHMEM instead of MPI code. This test first times the execution of *shmem_longlong_fadd_nb()* immediately followed by a wait call on the operation (effectively making it a blocking atomic operation). Next, it times the execution of the non-blocking atomic followed by increasingly large amounts of work before waiting on the operation. When the total time spent starts to exceed the "blocking" time, all the latency that can be exploited for additional computation has been used, so it records the time breakdown for the point at which it crossed the latency boundary.

Using this benchmark, we show that for atomic memory updates, 86% of the total time taken by the call can be employed doing other useful work (see Figure 1). This is significant and insightful especially since atomics are low latency calls - the overlap opportunity afforded by other non-blocking calls like collective operations and data transfer will be much greater. Additionally, the non-blocking call was shown to not add any noticeable overhead, so there is no disadvantage to using the non-blocking calls over their blocking counterparts.

3 Proposed Extensions

In this section, we describe and present our proposed set of extensions to OpenSHMEM 1.0. These incremental extensions are aimed at addressing some of the scalability bottlenecks of OpenSHMEM interfaces, and providing communication interfaces that enable overlapping communication with computation in applications.

Our set of extensions consist of:

- *Explicit active sets*, which allow the OpenSHMEM users to define the criterion for creation of the active set as well as the lifetime of the active set.
- *Non-blocking Collective* operations include a method for invoking the collective operation, and a method for learning the progress of the call. It enables the OpenSHMEM users to overlap collective communication with computation.

– *Non-blocking put, get, and atomic operations*, like the non-blocking collective operations extension, have a method for invoking the call, and a method for learning the progress of the call.
– *Abort and Exit Support* allows OpenSHMEM programs to terminate at any point during their execution. It enables applications to free the resources before exiting, or facilitate sanity checks before exiting the OpenSHMEM environment.

3.1 Explicit Active Sets

Our proposed extension to active sets will help the programmer define and reuse active sets explicitly via a proposed API. The active set construct will help the user group sets of processes and reuse active sets across collective operations using the OpenSHMEM API. Our proposed active set extension is an incremental extension to the active sets in OpenSHMEM 1.0, where it implicitly created sets of processes used for executing collective operations. A set of API calls will be used to define a set of PEs that comprise an active set, each using a different selection strategy, and will return an opaque handle that can be used thereafter to identify this set of PEs. This handle will only be guaranteed to be usable within collectives for PEs within the set, though the associated creation function may be called by any superset of the PEs contained in the set. Whether a particular calling PE is in the active set defined by the handle may be checked by calling the *shmem_in_aset()* function, which will return a non-zero value if it is in the set, or zero if it is not. The size of a particular set can be queried through the *shmem_aset_size()* function, and the *shmem_aset_delete()* function destroys the active set object. The full list of supported call signatures for active sets is included below in Figure 2.

Create a strided active set.	shmem_aset *shmem_create_strided_aset(int PE_start, int PE_stride, int PE_size)
Create a log-strided active set.	shmem_aset *shmem_create_log_strided_aset(int PE_start, int PE_log_stride, int PE_size, int stride_base)
Create a user-defined active set.	shmem_aset *shmem_create_custom_aset(int PE_start, shmem_offset_fn offset, int PE_size, void *const_params)
Create a generic active set.	shmem_aset *shmem_create_generic_aset(int *PE_list, int PE_size)
Check if a PE is in an active set.	int shmem_in_aset(shmem_aset *aset)
Query the size of an active set.	int shmem_aset_size(shmem_aset *aset)
Delete an active set.	void shmem_delete_aset(shmem_aset *aset);

Fig. 2: Examples of Proposed active set operations and their APIs

In the case of concurrent collective operations that involve overlapping active sets the user has to ensure that they do not work on the same pSync or pWrk

arrays. With these new extensions, the user will be able to create active sets using one of the four active set creation calls that can be used to select the member PEs of the active set. These calls allow the user to create active sets using strided sets based on both powers of two and arbitrary strides, generic arrays of participating PE ids, and via user-defined functions. For the latter case, a user may define an active set with a function that when called on a set of size n, will produce the ith PE id in the set for $0 \leq i < n$. All PE ids generated by the function must be both valid and unique in the set, and the id generated for a particular input must always be the same regardless of how many times it is called. The user can pass an arbitrary amount of data as actual parameters to the function specified at creation time of the active set. After being passed to the creation function, these parameters should remain constant for the lifetime of the active set. As an example, a custom function may be thought of as being very similar to a mathematical function such as $f(i) = i^2 + \text{PE\_start}$, for $0 \leq i < n$, for which $n = 4$ and $\text{PE\_start} = 3$ would produce the set $\{3, 4, 7, 12\}$.

In addition, we will use explicit active sets as arguments to create the proposed non-blocking collective calls as described in Section 3.2.1. This is to encourage the use of explicit active sets, together with non-blocking collective operations, while minimizing changes to the existing OpenSHMEM 1.0 API. The behavior for the original collective operations will remain the same, including `shmem_barrier_all()`, which will always use all PEs in the system.

An example is shown below demonstrating the process of creating and using a strided active set:

```
1  int main(int argc, char **argv) {
2     shmem_aset *aset;
3     ...
4     /* creates an active set containing PEs 0, 3, 6, 9, ... */
5     aset = shmem_create_strided_aset(0, 3, npes / 3);
6     /* equivalent to the OpenSHMEM 1.0-style (me % 3 == 0) */
7     if (shmem_in_aset(aset)) {
8        shmem_barrier_aset(aset, pSync);
9     }
10 }
```

Similarly, creating a custom function for selecting the PEs in an active set involves little more than creating the custom function itself:

```
1  int my_custom_fn(int PE_index, int PE_start, int PE_size, void
       *const_params) {
2     return PE_index * PE_index + PE_start;
3  }
4  int main(int argc, char **argv) {
5     shmem_aset *aset;
6     ...
7     /* creates an active set containing PEs 2, 3, 6, 11 */
8     aset = shmem_create_custom_aset(2, &my_custom_index_fn, 4, NULL);
9     if (shmem_in_aset(aset)) {
10       shmem_barrier_aset(aset, pSync);
11    }
12 }
```

The placement and use of the *shmem_aset_create_func()* and *shmem_in_aset()* functions here may seem unusual for some SHMEM developers. Being placed outside of the conditional means that the create function can be called by PEs

that are not in the defined active set. This is due to the updated syntax for active set creation, and will still result in valid code.

The performance of each of the four methods is compared using barrier operations to that of the implicitly defined active sets from OpenSHMEM 1.0 in Figure 3. All the tests were performed using the same OpenSHMEM reference implemention modified to use UCCS as described in Section 2. Each of the results represents the time spent performing a barrier on four PEs, using either an implicitly defined logarithmically strided active set or one of the four methods for creating explicit active sets as previously described. It can be seen that not only is there no additional overhead for defining active sets explicitly compared to the original implicit definitions, but there is also no performance penalty dependent on which method for defining an explicit active set is chosen.

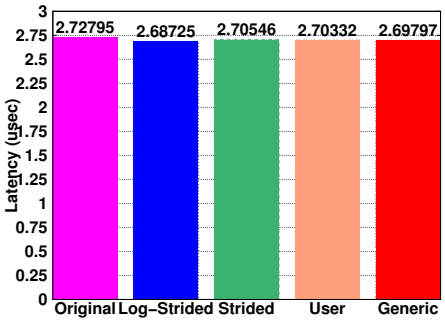

Fig. 3: Performance of Barriers on Active Sets

3.2 Non-blocking Operations

All proposed non-blocking operations require a mechanism to check for completion of the request. For this purpose we introduce an opaque request object of type *shmem_request_handle_t* and two query functions defined on this request object, namely, *shmem_wait_req()* and *shmem_test_req()*. Each non-blocking operation will produce exactly one such handle, which will be unique to that particular outstanding operation. All calls to non-blocking operations return immediately and all participating PEs must check for completion by using a *wait* or a *test* before reusing any resources involved in the operation. A call to *wait* will return only once the operation is completed, while *test* returns immediately with information on the status of the request, regardless of whether or not it has completed. There are no specific requirements in terms of the progress model for these operations and an OpenSHMEM library implementation is free to choose when and how the operations progress.

3.2.1 Non-blocking Collective Operations We extend the OpenSHMEM API for collective operations such as collection, reduction, barrier and broadcast by adding their non-blocking variants. With the new non-blocking collective operations a single collective call is replaced by a call to the non-blocking collective (which would return a request handle) followed by a call to *shmem_wait_request()* which accepts the handle as a parameter and returns when the collective call has completed on the calling PE. All non-blocking collective operations are defined on explicit active set handles as discussed in Section 3.1, instead of the triplet *PE_start, logPE_stride and PE_size*. Worth noting is that pSync remains a required part of the function. This is due to the fact that the active sets handles themselves do not have any dedicated memory assigned to them, separating the handling of memory from the selection of PEs for any particular active set definition. This allows a developer to maintain the same high degree of control over lower level management of the program and its associated memory as has been traditionally possible, while still receiving the benefits of the new active set definitions. A complete example showing a non-blocking collective operation in an OpenSHMEM program is illustrated below:

```
1  int main(int argc, char *argv[]){
2     shmem_aset *aset1;
3     shmem_request_handle_t request1;
4     ...
5     start_pes(0);
6     ...
7     if(me%2 == 0){
8        // aset1 contains 0,2,4,6
9        aset1 = create_aset_strided(0,2,4,&err);
10       shmem_broadcast32_nb(&y,&x,1,aset1,pSync, &request1);
11       ...
12       //some useful work
13       ...
14       shmem_wait_req(request1);
15    }
16    ...
17    return 0;
18 }
```

The non-blocking collective semantics allow multiple outstanding collective operations to be in progress on a given active set at any particular point in time. Each outstanding non-blocking and blocking collective requires its own symmetric array. A high quality implementation would not require a call to *shmem_test_request()* to progress the outstanding collective operation. However, a semantically correct implementation can progress asynchronously, or during a call to the OpenSHMEM library.

3.2.2 Non-blocking Atomic Operations In the same manner, we introduce non-blocking variants for the OpenSHMEM atomic library calls (swap, compare-and-swap, increment, fetch-and-increment, add, fetch-and-add). A call to a non-blocking atomic returns a handle which can be passed as a parameter to *shmem_wait_request()*. The *wait* returns when the atomic operation has completed on all the local as well as target PE. An example of non-blocking atomic operation in an OpenSHMEM program is illustrated below:

```
int main(int argc, char *argv[]){
   shmem_request_handle_t request1;
   ...
   start_pes(0);
   ...
   shmem_longlong_fadd_nb(target, 10, 1, &oldval, &request1);
   ...
   //some useful work
   ...
   shmem_wait_request(request1);
   ...
   return 0;
}
```

3.2.3 Non-blocking Data Transfer Operations According to OpenSHMEM Specification 1.0 the *put* operation returns only after the local buffer is available for reuse. As a non-blocking extension to the *put* operation, the call will return immediately and the local buffer will not be available for reuse until the operation has achieved local completion. These non-blocking semantics can be especially advantageous for communication patterns that involve communicating parts of a buffer to different PEs without immediate buffer reuse. The *get* operation returns only after the value is updated at the local PE. For programs that do not need the value updated by the *get* call immediately, waiting for local completion is an unnecessary burden. The non-blocking *get* operation will allow the local PE to execute other calls and operations while waiting for the remote data to be communicated. An example of a non-blocking *get* operation in an OpenSHMEM program is illustrated below:

```
int main(int argc, char *argv[]){
   shmem_request_handle_t request1;
   ...
   start_pes(0);
   ...
   shmem_int_get_nb(target, source, 1, me+1, &request1);
   //some useful work
   //call wait before the value is required by local PE
   shmem_wait_request(request1);
   x = target + 0.25 * y;
   ...
   return 0;
}
```

Non-blocking data transfer is not a novel concept and SHMEM implementations like Quadrics [9] have included it in its library API. We differ in our approach as we define the non-blocking operations on explicit active set handles. By creating a explicit active set we not only simplify the API but make OpenSHMEM programs more suitable for analysis via compiler based tools.

3.3 Thread Safety

In OpenSHMEM 1.0 there is no mention of thread safety and what one might expect when trying to execute an OpenSHMEM program in a multi-threaded environment. Providing basic thread safety support may promote interoperability between OpenSHMEM and other programming models and allow data transfers

to be broken down into smaller units to facilitate communication latency hiding via overlap. Since a PE maps to a process in a multi threaded environment, as the current specification stands, multiple threads executing OpenSHMEM calls such as *start_pes()*, *shmem_finalize()*, *shmalloc()*, *shfree()*, and *shrealloc()* can lead to unpredictable execution patterns and results. Collective operations are a major OpenSHMEM functionality group that may be executed by only one thread per PE. Since the concept of an active set is a purely logical one in OpenSHMEM Specification 1.0 there is no way to define operations on it. With explicit active set handles there is a programmable object on which we can now define thread-safe operations and constraints. Although prototypes to these functions are beyond the scope of this paper, four hierarchical execution model modes for threading support proposed by Cray (*thread single, thread funneled, thread serialized* and *thread multiple*) [10] could be well suited for providing thread safety semantics with the new extensions. Each of the four levels of threading support specifies the number of threads per PE that may participate, and this is set through a thread safety initialization call *shmem_init_thread()* which accepts one on the four modes as an argument.

3.4 Abort and Exit Support

The OpenSHMEM Specification 1.0 does not have any support for an abort or exit call. As an extension to OpenSHMEM Specification 1.0, we propose a *shmem_finalize()* function to shut down the library, and *shmem_abort()* to signal the abortion of the OpenSHMEM program. The primary difference in the semantics of *shmem_finalize()* and *shmem_abort()* is that *shmem_finalize()* indicates normal completion of the program and the OpenSHMEM environment can be re-initialized after *finalize* by calling *start_pes()*. *shmem_abort()*, however, signals the run-time that OpenSHMEM library operations cannot continue after the call returns. Semantics for both the calls are as follows:

1. ***shmem_finalize()***
   (a) It is the last OpenSHMEM call by any PE.
   (b) All pending OpenSHMEM operations will have completed when the call returns.
   (c) The OpenSHMEM environment can be re-initialized by calling *start_pes()* after *finalize*.
   (d) Any OpenSHMEM calls after *shmem_finalize* and before the *start_pes()* will lead to undefined behavior.
2. ***shmem_abort()***
   (a) Any PE can call *shmem_abort()* to the program execution. After any PE calls the *shmem_abort()* call, the behavior of the program is undefined.
   (b) Any OpenSHMEM operation after any PE calls *shmem_abort()* will lead to undefined behavior.
   (c) After exiting the OpenSHMEM environment, a process may or may not terminate depending on the library implementation.

4 A Vision for OpenSHMEM's Future

OpenSHMEM-NG is a vision for the next big leap in the evolution of OpenSHMEM. Here we propose new ideas: changes are not incremental, and not necessarily backward compatible with OpenSHMEM 1.0. In order to address the need for backward compatibility, we plan to develop source-to-source translation tools that will help to update legacy applications to a new standard.

4.1 Adding Memory Context to Active Sets

In the current OpenSHMEM Specification 1.0, all symmetric memory allocations have to be made across all the PEs in an application. However, with the introduction of explicit active sets (Section 3), the active set opaque handle may be reused for multiple collective operations. This provides more control and flexibility to the user when decomposing the work within an OpenSHMEM application. Adding a memory context to the active set, where memory context is a symmetric memory space available only to the members of the active set, is the next logical step. The memory context will provide an efficient medium for memory management without incurring the cost of memory allocation across all PEs. Moreover, this will also help address the issue of isolation, where applications and multiple libraries using OpenSHMEM can safely co-exist independently of each other. Introducing memory context to active sets can also address the issue of *locality*, where logical sets of PEs and memory spaces can be used to define locality and be mapped to cores and memory that are close to each other.

4.2 Error Model

In OpenSHMEM-NG, error reporting will be an important aspect. As the complexity of the hardware and programming environment increases, it becomes increasingly important to be able to identify errors and provide meaningful information to the programmer. This also paves the path towards fault tolerance and resilience. Other than programming errors there exist a plethora of error conditions related to memory, network, and communication failures. Extending the OpenSHMEM API with error handlers and defined error states will enable proper error handling on the application level.

4.3 OpenSHMEM I/O

The OpenSHMEM I/O extensions will be aimed at providing interfaces with parallel I/O capabilities for OpenSHMEM applications. The interfaces will be geared towards facilitating and co-ordinating concurrent I/O access among the OpenSHMEM PEs, and abstracting the semantics provided by parallel file systems to match the needs of applications.

5 Related Work

Process groups is a concept that has been around in programming models such as MPI. It has also been proposed as an extension to existing PGAS languages such as CAF 2.0 and Titanium. The concept of *locales* in Chapel and *places* in X10 is tied to process affinity and locality, where the programmer can map computations or data to specific *locales* or *places*. The concept of groups has been used to define communication contexts that can be used in coupled applications that perform communication and computation in subsets and independently from each other, such as an application and library using the same communication library independently of each other.

Co-array Fortran 2.0 introduced the concept of teams [11], ordered sequences of process images that represent a subset of an existing team. All process images start as members of a global team known as `team_world`. New teams can be created from existing teams by splitting based on a common "color" or merging two teams to produce the union of them. Additionally, a topology can also be applied to a team to abstract the layout and access patterns of the processes involved in an operation. Titanium also uses a similar concept where teams of threads are defined as objects that have methods to split into sub teams.

Similarly, communicators in MPI are arbitrary sets of processes used for performing communication on them selectively as independent functional units. Initially, all processes are part of a single global communicator, after which sub-communicators may be created from it either by splitting or by specifying a subset of parent processes to either include or exclude [12]. While this approach can provide a lot of flexibility, the communicator creation process imposes implicit synchronization, which might be undesirable for other programming models.

The incremental change to OpenSHMEM Specification 1.0 will include explicit handles to active sets. At this juncture, however, there is no communication context associated with it. This is due to the desired separation between active set definitions and memory spaces and handling, as described in Section 3.2.1. For OpenSHMEM-NG, having a concept similar to communicators in MPI with isolated communication contexts may be useful.

CAF 2.0 incorporated asynchronous point-to-point and collective operations [13] and detailed the benefits that such operations could provide by providing a higher degree of communication to computation overlap. Hiding the latency of communication has been employed in other places before CAF 2.0. For example, MPI showed that adding non-blocking collectives to MPI-2 allowed them to provide a 99% overlap between communication and computation along with an approximate performance gain between 13 to 15% (depending on the underlying network) for the 3D-FFT application [14]. Even before that, asynchronous collective operations like broadcast in MPI [15] [16] and barrier [17] have been studied to enable hiding of communication latency and barrier duration of barrier regions. Other PGAS languages like X10 support asynchronous activities via the *async* statement, and use *phaser accumulators* for increasing communication-computation overlap for reduction operations [18]. Both UPC [19] and MPI have varying degrees of support for a global exit of all threads and processes

respectively. This allows for the executing process elements to do a collective exit when some explicit execution requirement is not met (*abort*) or do a clean exit at the end of the program (release resources and do other sanity checks). The obvious benefit of these concepts has prompted these proposed extensions to OpenSHMEM 1.0.

6 Conclusions

In this paper, we described a series of extensions to the *OpenSHMEM* API that strive to maintain backward compatibility with the current *OpenSHMEM* API, and aim to improve programmer productivity as well as the performance and scalability of *OpenSHMEM* applications. The extensions include 1) Explicit active sets 2) Non-blocking operations 3) Library Shutdown, and 4) Multi-threading support.

We made a case for a series of extensions that is far-reaching and geared more towards the needs of exascale era applications and hardware. The extensions in this category include 1) Isolation 2) Locality 3) Error Model, and 4) I/O.

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.

References

1. OpenSHMEM Org.: OpenSHMEM specification (2011)
2. Chapman, B., Curtis, T., Pophale, S., Poole, S., Kuehn, J., Koelbel, C., Smith, L.: Introducing OpenSHMEM: SHMEM for the PGAS community. In: Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model. PGAS '10, New York, NY, USA (2010)
3. Poole, S.W., Hernandez, O., Kuehn, J.A., Shipman, G.M., Curtis, A., Feind, K.: OpenSHMEM - Toward a Unified RMA Model. In: Encyclopedia of Parallel Computing. (2011) 1379–1391
4. Pophale, S., Nanjegowda, R., Curtis, T., Chapman, B., Jin, H., Poole, S., Kuehn, J.: Openshmem performance and potential: A npb experimental study (2012)
5. Pophale, S.S.: SRC: OpenSHMEM library development. In Lowenthal, D.K., de Supinski, B.R., McKee, S.A., eds.: ICS, ACM (2011) 374
6. Shamis, P., Venkata, M.G., Kuehn, J.A., Poole, S.W., Graham, R.L.: Universal common communication substrate (uccs) specification. version 0.1. Tech Report ORNL/TM-2012/339, Oak Ridge National Laboratory (ORNL) (2012)
7. Graham, R.L., Shamis, P., Kuehn, J.A., Poole, S.W.: Communication middleware overview. Tech Report ORNL/TM-2012/120, Oak Ridge National Laboratory (ORNL) (2012)
8. Lawry, W., Wilson, C., Maccabe, A.B., Brightwell, R.: Comb: A portable benchmark suite for assessing mpi overlap. In: IEEE Cluster. (2002) 23–26

9. Quadrics Supercomputers World Ltd.: SHMEM Programming Manual (2001)
10. CRAY: Thread-safe shmem extensions (2012)
11. Mellor-Crummey, J., Adhianto, L., Scherer, III, W.N., Jin, G.: A new vision for coarray fortran. In: Proceedings of the Third Conference on Partitioned Global Address Space Programing Models. PGAS '09, New York, NY, USA, ACM (2009) 5:1–5:9
12. Walker, D.W., Walker, D.W., Dongarra, J.J., Dongarra, J.J.: Mpi: A standard message passing interface. Supercomputer **12** (1996) 56–68
13. Scherer, III, W.N., Adhianto, L., Jin, G., Mellor-Crummey, J., Yang, C.: Hiding latency in coarray fortran 2.0. In: Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model. PGAS '10, New York, NY, USA, ACM (2010) 14:1–14:9
14. Hoefler, T., Kambadur, P., Graham, R., Shipman, G., Lumsdaine, A.: A case for standard non-blocking collective operations. In Cappello, F., Herault, T., Dongarra, J., eds.: Recent Advances in Parallel Virtual Machine and Message Passing Interface. Volume 4757 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2007) 125–134
15. Almási, G., Heidelberger, P., Archer, C.J., Martorell, X., Erway, C.C., Moreira, J.E., Steinmacher-Burow, B., Zheng, Y.: Optimization of mpi collective communication on bluegene/l systems. In: Proceedings of the 19th annual international conference on Supercomputing. ICS '05, New York, NY, USA, ACM (2005) 253–262
16. Cachin, C., Kursawe, K., Petzold, F., Shoup, V.: Secure and efficient asynchronous broadcast protocols. In Kilian, J., ed.: Advances in Cryptology CRYPTO 2001. Volume 2139 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2001) 524–541
17. Gupta, R.: The fuzzy barrier: a mechanism for high speed synchronization of processors. In: Proceedings of the third international conference on Architectural support for programming languages and operating systems. ASPLOS III, New York, NY, USA, ACM (1989) 54–63
18. Shirako, J., Peixotto, D.M., Sarkar, V., Scherer, W.: Phaser accumulators: A new reduction construct for dynamic parallelism. In: Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on. (2009) 1–12
19. UPC Consortium: Upc language specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab (2005)