OpenSHMEM as a Portable Communication Layer for PGAS Models: A Case Study with Coarray Fortran

Naveen Namashivayam, Deepak Eachempati, Dounia Khaldi and Barbara Chapman
Department of Computer Science
University of Houston
Houston, Texas
{*nravi, dreachem, dounia, chapman*}*@cs.uh.edu*

***Abstract*—Languages and libraries based on the Partitioned Global Address Space (PGAS) programming model have emerged in recent years with a focus on addressing the programming challenges for scalable parallel systems. Among these, Coarray Fortran (CAF) is unique in that as it has been incorporated into an existing standard (Fortran 2008), and therefore it is of particular importance that implementations supporting it are both portable and deliver sufficient levels of performance. OpenSHMEM is a library which is the culmination of a standardization effort among many implementers and users of SHMEM, and it provides a means to develop light-weight, portable, scalable applications based on the PGAS programming model. As such, we propose here that OpenSHMEM is well situated to serve as a runtime substrate for CAF implementations.**

In this paper, we demonstrate how OpenSHMEM can be exploited as a runtime layer upon which CAF may be implemented. Specifically, we re-targeted the CAF implementation provided in the OpenUH compiler to OpenSHMEM, and show how parallel language features provided by CAF may be directly mapped to OpenSHMEM, including allocation of remotely accessible objects, one-sided communication, and various types of synchronization. Moreover, we present and evaluate various algorithms we developed for implementing remote access of non-contiguous array sections and acquisition and release of remote locks using the OpenSHMEM interface.

***Keywords*-PGAS; OpenSHMEM; Coarray Fortran; Strided communications; Parallel Programming Model designs**

I. Introduction

In recent years, a class of languages and libraries referred to as *Partitioned Global Address Space* (PGAS) [1] has emerged as an alternative to MPI for programming large-scale parallel systems. The PGAS programming model is characterized by a logically partitioned global memory space, where partitions have affinity to the processes/threads executing the program. This property allows PGAS-based applications to specify an explicit data decomposition that reduces the number of remote accesses with longer latencies. This model attempts to combine both the advantages of the SPMD model for distributed systems with the data referencing semantics of shared memory systems.

Several languages and libraries provide a PGAS programming model, some with extended capabilities such as asynchronous spawning of computations at remote locations in addition to facilities for expressing data movement and synchronization. Unified Parallel C (UPC) [2], Coarray Fortran (CAF) [3], X10 [4], Chapel [5], and CAF 2.0 [6] are examples of language-based PGAS models, while OpenSHMEM [7] and Global Arrays [8] are examples of library-based PGAS models. Library-based models assume the use of a base language (C/C++ and Fortran being typical) which does not require or generally benefit from any special compiler support. This essentially means the library-based PGAS implementations rely completely on the programmer to use the library calls to implement the correct semantics of its programming model.

Over the years there have been numerous system-level interfaces which have been used for both language- and library-based PGAS implementations, including GASNet [9], Infiniband Verbs API [10], MVAPICH2-X [11], ARMCI [12], and Cray's uGNI [13] and DMAPP [14]. Several OpenSHMEM implementations are available over a broad range of parallel platforms, including implementations based on GASNet, ARMCI, MPI-3.0 [15], and MVAPICH2-X [16]. The Berkeley UPC project uses GASNet, and the University of Houston's CAF implementation [17] works with either GASNet or ARMCI. The SHMEM, CAF, and UPC implementations [18] from Cray use DMAPP.

OpenSHMEM, referred to above as a library-based PGAS model, is a library interface specification which is the culmination of a unification effort among many vendors and users in the SHMEM programming community. Some important OpenSHMEM implementations are discussed in Section II-B. OpenSHMEM was designed to be implemented as a light-weight and portable library offering the facilities required by PGAS-based applications. In this paper, we will examine the suitability of OpenSHMEM as a runtime layer for other PGAS languages. In particular, we focus on how Coarray Fortran may be implemented on top of OpenSHMEM. The motivation for such a mapping is that OpenSHMEM is widely available, and it provides operations with very similar semantics to the parallel processing features in CAF – thus the mapping can be achieved with very little overhead which we show. Furthermore, such an implementation allows us to incorporate OpenSHMEM calls directly into CAF applications (i.e. Fortran 2008 applications using coarrays and related features) and explore the ramifications of such a hybrid model.

We found that not all operations in CAF have an efficient 1-to-1 mapping to OpenSHMEM. In particular, we developed in this work new translation strategies for handling remote access involving multi-dimensional strided data and acquisition and release of remote locks. Strided communications for multi-dimensional arrays are fundamental in many scientific applications and warrant careful consideration into how they may be efficiently handled in PGAS implementations. The OpenSHMEM API supports *indexed* remote memory access operations which work on only 1-dimensional strided data, and its interface for locks is unsuitable for CAF. Therefore, in this paper we introduce a new algorithm for the general case of strided data communications in multi-dimensional arrays and also an adaptation of the MCS lock algorithm [19] for the case of coarray locks on a specified image.

The contributions of this paper are thus:

- a description of how strided data communications for multi-dimensional arrays may be efficiently implemented over OpenSHMEM;
- extensions to the MCS lock algorithm for supporting CAF locks over OpenSHMEM;
- a comprehensive evaluation of this CAF-OpenSHMEM implementation using our PGAS microbenchmarks [20], a coarray version of a distributed hash table code [21] and Himeno Benchmark [22].

This paper is organized as follows. We discuss CAF features and its various available implementations as well as various OpenSHMEM implementations in Section II. We motivate the choice of OpenSHMEM as a communication layer for CAF in Section III. We show the translation of the features of CAF to OpenSHMEM in Section IV. Experimental results using PGAS microbenchmarks, a distributed hash table benchmark, and the Himeno benchmark are presented in Section V. We survey different related designs of PGAS models in Section VI. Finally, we discuss future work and conclude in Section VII.

II. Background

In this section we given an overview of the Coarray Fortran and OpenSHMEM programming models.

A. Overview of Coarray Fortran (CAF)

CAF is an explicitly parallel subset of the Fortran language, first standardized in Fortran 2008, which adheres to the PGAS programming model. CAF programs follow an SPMD execution model, where all execution units called images are launched at the beginning of the program; each image executes the same code and the number of images remains unchanged during execution. Each image can access data belonging to other images through remote memory accesses without the actual involvement of the source image. Several additional features, not presently in the Fortran standard, are expected in a future revision and are available in the CAF implementation in OpenUH [17] compiler. Data objects belonging to an image may be remotely-accessible or purely local (only accessible by the local image). Remotely accessible data objects in CAF may be further divided into two categories – symmetric and non-symmetric. Symmetric, remotely accessible objects include coarrays and subobjects of coarrays. Non-symmetric, remotely accessible objects include allocatable components of coarrays of a derived type, and objects with the target attribute which are associated with a pointer component of a coarray of derived type. Remotely accessible objects in CAF can be accessed by a non-local image without the local image's explicit participation. CAF consists of various mechanisms for facilitating parallel processing, including: declaration of or dynamic allocation and deallocation of remotely-accessible data objects, one-sided communication, remote atomic operations, barriers, point-to-point synchronization, broadcasts, and reductions. Any remote memory access in CAF must involve a reference to a coarray, and it is achieved through the use of square brackets [] (a "co-indexed coarray reference").

The left-hand side of Figure 1 depicts an example of a CAF program. In this example code, `coarray_x` is a statically declared coarray (symmetric, remotely-accessible variable) and `coarray_y` is a dynamically allocated coarray. The intrinsics `num_images()` and `this_image()` return the total number of images and the local image index (a unique index between 1 and *num_images*, inclusive), respectively. Finally, we can see how remote memory accesses are achieved in CAF, using the co-subscript notation.

Table I summarizes various available implementations of CAF and their respective communication layer details. GFortran [23] has added CAF support relatively recently,

Implementation	Compiler	Communication Layer
UHCAF	OpenUH	GASNet, ARMCI
CAF 2.0	Rice	GASNet, MPI
Cray-CAF	Cray	DMAPP
Intel-CAF	Intel	MPI
GFortran-CAF	GCC	GASNet, MPI

Table I: Implementation details for CAF

and uses the OpenCoarrays CAF runtime which can work over MPI and GASNet. The Cray CAF implementation uses Cray's DMAPP API, while CAF 2.0 from Rice has both GASNet [6] and (recently) MPI [24] layers. The Intel implementation uses Intel MPI implementation. UHCAF, the CAF implementation provided by the OpenUH [17] compiler, can execute over either GASNet or ARMCI. In this paper, we describe our approach in targeting CAF to OpenSHMEM using our UHCAF implementation.

```
integer :: coarray_x(4)[*]
integer, allocatable :: coarray_y(:)[:]

allocate(coarray_y(4)[*])

integer :: num_image
integer :: my_image
num_image = num_images()
my_image = this_image()

coarray_x = my_image
coarray_y = 0

coarray_y(2) = coarray_x(3)[4]
coarray_x(1)[4] = coarray_y(2)

sync all
```

```
start_pes(0);
int *coarray_x, *coarray_y;

coarray_x = shmalloc(4 * sizeof(int));
coarray_y = shmalloc(4 * sizeof(int));

int num_image = my_pe();
int my_image = num_pes();

for (int i = 0; i < 4; i++) {
  coarray_x[i] = my_image;
  coarray_y[i] = 0;
}
shmem_int_get(*coarray_y+(2*4),*coarray_x+(3*4),1,3);
shmem_int_put(*coarray_x+4,*coarray_y+(2*4),1,3);

shmem_barrier_all();
```

Figure 1: An OpenSHMEM variant of a CAF code

B. *Overview of OpenSHMEM*

OpenSHMEM which offers many additional features for supporting PGAS-based applications. The features encompass management of symmetric objects, remote read and write operations, barrier and point-to-point synchronization, atomic memory operations, and collective operations. OpenSHMEM programs follows an SPMD-like style of programming where all processing elements (PEs) are launched at the beginning of the program. One general objective of OpenSHMEM libraries is to expose explicit control over data transfer with low latency over communications. The right hand side of Figure 1 depicts an example of the SHMEM variant for the same program shown on the left.

Historically, there have been multiple vendor library implementations of SHMEM. OpenSHMEM [25] is an effort to create a standardized specification for the SHMEM interface. [25] provides a list of various available SHMEM libraries from different vendors including Cray, SGI, Quadrics, and other implementations based on GASNet and ARMCI. With various optimized vendor specific implementations, it has the potential to be used as an effective communication layer for other PGAS models such as CAF.

III. Why OpenSHMEM?

In this section, we motivate the use of OpenSHMEM as a portable communication layer for PGAS models. The goal in this section is to identify the performance of OpenSHMEM implementations, namely Cray SHMEM and MVAPICH2-X SHMEM, against two other one-sided communication libraries which are candidates for supporting PGAS languages, GASNet and MPI-3.0. For these measures, we used two platforms, Stampede and Titan III. For both systems, each node allows execution with up to 16 cores per node. We used microbenchmarks from the PGAS Microbenchmark Suite [20] to motivate our use of CAF over OpenSHMEM. We benchmarks point-to-point communication tests using `put` and `get` operations in SHMEM, MPI-3.0 and GASNet. These tests are between pairs of PEs, where members of the same pair are always on two different nodes. The tests are performed using 1 and 16 pairs across two compute nodes, to assessment communication cost with and without inter-node contention.

The latency performance results of `put` tests are shown in Figure 2. In Figure 2(a) and Figure 2(b), we have compared the OpenSHMEM implementation over MVAPICH2-X [16] against the GASNet and MPI-3.0 implementation over MVAPICH2-X on Stampede. In Figure 2(c) and Figure 2(d), we have compared the Cray SHMEM implementation against Cray MPICH and GASNet on the Cray XC30 platform.

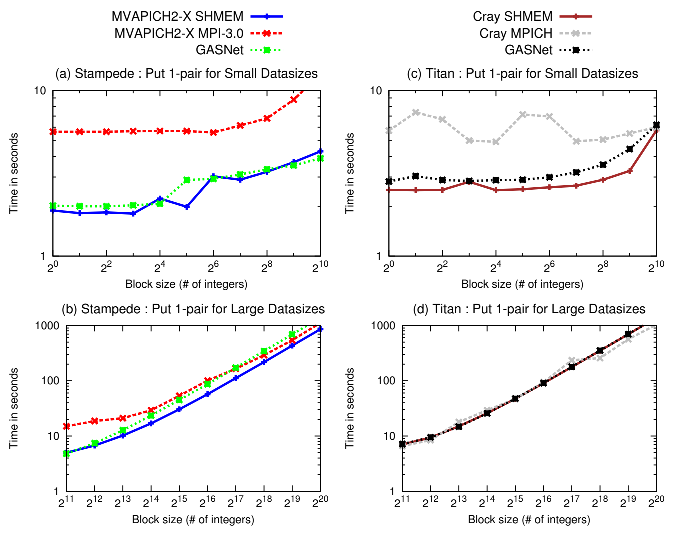


Figure 2: Put Latency comparison using two nodes for SHMEM, MPI-3.0 and GASNet

In general, the latency of both GASNet and OpenSHMEM is less than the tested MPI-3.0 implementations when there is no contention (1 pair). For small data sizes until 2^{10} integers, both the performance of OpenSHMEM and GASNet are almost similar on Stampede, but Cray SHMEM performs better than GASNet on Titan. For large message sizes OpenSHMEM performs better than GASNet. With inter-node contention (16 pairs), SHMEM performs better than both GASNet and MPI-3.0 on Stampede, and has latency similar to GASNet on Titan.

The bandwidth performance results for `put` tests are shown in Figure 3. In Figure 3(a) and Figure 3(b), we have compared the MVAPICH2-X OpenSHMEM implementation [16] against the GASNet and MVAPICH2-X MPI-3.0 implementations on Stampedek. In Figure 3(c) and Figure 3(d), we have compared the Cray SHMEM implementations against Cray MPICH and GASNet on Titan.

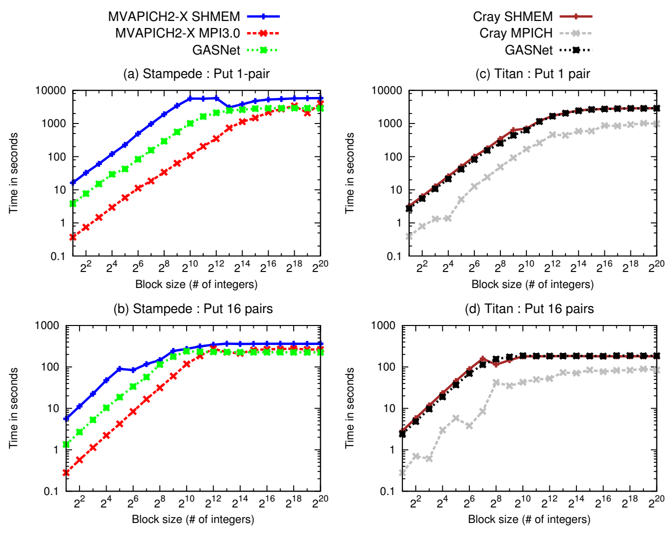


Figure 3: Put Bandwidth comparison using two nodes for SHMEM, MPI-3.0 and GASNet

The bandwidth of SHMEM is better than GASNet and MPI-3.0 on both the Stampede and Titan experimental setups. This is because MVAPICH2-X SHMEM is optimized using IB verbs on Stampede, while on Titan Cray SHMEM is optimized using DMAPP. Availability of certain features like remote atomics in OpenSHMEM also provides an edge over GASNet. These features are exploited in our CAF over OpenSHMEM implementation for implementing support for coarray locks.

From this, we can observe that OpenSHMEM implementations well tuned when compared to other available one-sided libraries. This provides OpenSHMEM an edge over other libraries for being used as a communication layer for other PGAS programming models. Apart from the performance factors, ease of programming and the wide availability makes OpenSHMEM more highly portable and easy to use as a communication layer for PGAS implementations.

IV. CAF Runtime Design over OpenSHMEM

Table II presents a comparative study of various available features in CAF and OpenSHMEM. Symmetric data allocations, remote memory data accesses, atomic operations, and collectives [1] are available in both models. So, these features available in OpenSHMEM may be used directly to implement the corresponding CAF operations. However, features which are unavailable in OpenSHMEM, such as multi-dimensional strided data communications and remote locks, have to be effectively implemented for CAF. In this section, we present the design and implementation of our translation.

A. *Symmetric Data Allocation*

Language features in CAF for supporting parallel execution are similar to the features provided by OpenSHMEM. Consider Figure 1 which shows the correspondence between CAF and OpenSHMEM constructs. Coarrays (which may be `save` or `allocatable`) are the mechanism for making data remotely accessible in a CAF program; the corresponding mechanism in OpenSHMEM is to declare the data either as *static* or global or to explicitly allocate it using the *shmalloc* routine. Hence, remotely accessible data in CAF programs may be easily handled using OpenSHMEM. A `save` coarray will be automatically remotely accessible in OpenSHMEM, and we can implement the `allocate` and `deallocate` operations using `shmalloc` and `shfree` routines, respectively. Additionally, coarrays of derived type may be used to allow non-symmetric data to be remotely accessible. To support this with OpenSHMEM, we *shmalloc* a buffer of equal size on all PEs at the beginning of the program, and explicitly manage non-symmetric, but remotely accessible, data allocations out of this buffer. The CAF `this_image` and `num_images` intrinsics map directly to the `my_pe` and `num_pes` routines in OpenSHMEM.

B. *Remote Memory Accesses*

Remote memory accesses of contiguous data in OpenSHMEM can be performed using the `shmem_putmem` and `shmem_getmem` function calls, and they can be used to implement the remote memory accesses in CAF. Though the properties of RMA in CAF and OpenSHMEM are relatively similar, they are not exactly matching. CAF ensures that accesses to the same location and from the same image will be completed in order; OpenSHMEM has stronger semantics, allowing remote writes to complete out of order with respect to other remote accesses.

In the example shown in Figure 4, the initial values of *coarray_x* (which is 3) will be written to *coarray_y* at image 2, achieved using a `shmem_putmem` call. The subsequent modification of *coarray_x* to the value 0 does not require

[1]In UHCAF, we implement CAF reductions and broadcasts using 1-sided communication and remote atomics available in OpenSHMEM.

Properties	CAF	OpenSHMEM
Symmetric data allocation	`allocate`	`shmalloc`
Total image count	`num_images()`	`num_pes()`
Current image ID	`this_image()`	`my_pe()`
Collectives - reduction	co_*operator*	shmem_*operator*_to_all...
Collectives - broadcast	`co_broadcast`	`shmem_broadcast`
Barrier Synchronization	`sync all`	`shmem_barrier_all`
Atomic swapping	`atomic_cas`	`shmem_swap`
Atomic addition	`atomic_fetch_add`	`shmem_add`
Atomic AND operation	`atomic_fetch_and`	`shmem_and`
Atomic OR operation	`atomic_or`	`shmem_or`
Atomic XOR operation	`atomic_xor`	`shmem_xor`
Remote memory put operation	[]	`shmem_put()`
Remote memory get operation	[]	`shmem_get()`
Single dimensional strided put	[] to strided data	`shmem_put(,stride,)`
Single dimensional strided get	[] from strided data	`shmem_get(,stride,)`
Multi dimensional strided put	[]	×
Multi dimensional strided get	[]	×
Remote Locks	*lock*	×

Table II: Summary of features for supporting parallel execution in Coarray Fortran (CAF) and OpenSHMEM

additional synchronization, since shmem_putmem ensures local completion. However, the put operation from data from *coarray_b* to *coarray_a* needs an additional shmem_quiet function to ensure the remote completion of this transfer. Otherwise, the next transfer statement from *coarray_a* to *coarray_c* would become erroneous in OpenSHMEM.

```
coarray_x(:) = 3
coarray_y(:)[2] = coarray_x(:)
coarray_x(:) = 0

coarray_a(:)[2] = coarray_b(:)
coarray_c(:) = coarray_a(:)[2]
```

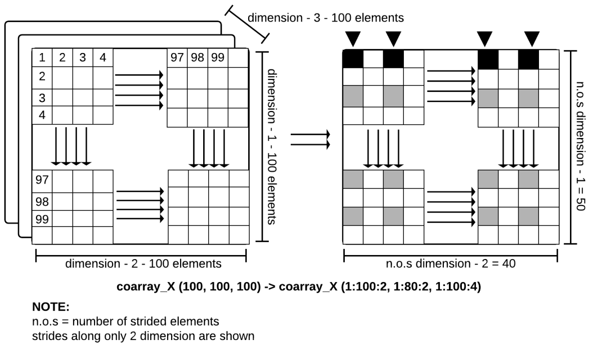
Figure 4: CAF RMA local and remote completion

Therefore, in our translation, we insert shmem_quiet after each call to shmem_put and before each call to shmem_get to handle remote memory transfers in CAF.

C. Strided Remote Memory Access

Strided remote memory accesses (RMA) in OpenSHMEM are available; however, the API presumes accesses of 1-dimensional array buffers with a single specified stride. Hammond [26] proposes a solution for two-dimensional (matrix oriented) strided data communication as an extension to OpenSHMEM using the DMAPP interface; however, to the best of our knowledge, there are no efficient proposals for multi-dimensional strided communications. In this section, we introduce an efficient algorithm for performing multi-dimensional strided remote accesses using the existing 1-dimensional strided routines (i.e. shmem_iput or shmem_iget).

The naïve way to implement multi-dimensional strides using OpenSHMEM is to use multiple shmem_putmem or shmem_getmem functions for each strided element on each dimension. For instance, consider a three dimensional array *coarray_X(100, 100, 100)*. When we perform a strided operation as *corray_X(1:100:2, 1:80:2, 1:100:4)*, we have 50, 40 and 25 strided elements in dimension 1, 2 and 3 respectively. Hence, we have a total of 50 * 40 * 25 shmem_putmem or shmem_getmem calls.



dimension - 3 - 100 elements
1
2
3
4
97
98
99
2
3
4
97
98
99
dimension - 1 - 100 elements
n.o.s dimension - 1 = 50
dimension - 2 - 100 elements
n.o.s dimension - 2 = 40
coarray_X (100, 100, 100) -> coarray_X (1:100:2, 1:80:2, 1:100:4)
NOTE:
n.o.s = number of strided elements
strides along only 2 dimension are shown

Figure 5: Multi-dimensional Stride Example

In our solution, we apply two optimizations: the first optimization is to identify the most suitable dimension for using the shmem_iput or shmem_iget strided calls and use it as the base dimension base_dim in order to reduce the number of strided calls. Consider the same example as above where we have dimension 1 with 50 strided elements, dimension 2 with 40 and dimension 3 with 25 strided elements. Here, the base dimension would be dimension 1, because it has more strided elements than the other dimensions; the base dimension can be called through `shmem_iput` or `shmem_iget`. Hence, the number of calls will be reduced to 1 * 40 * 25 which is much better than the naïve solution. Besides reducing the number of strided calls, we should consider the locality of data. If we consider the above example and we use the second dimension to be a base dimension, we have the strided length for each element to be of size 100 since the size of dimension 1 is 100. In this case, we will obtain data from different cache levels. The tradeoff between reducing the number of calls and considering the data locality is considered in our approach by applying the `base_dim` calculation to only the two first dimensions. The dark spots on Figure 5 shows the locations where `shmem_iput` operations occur. We implemented both these two optimizations and call this algorithm `2dim_strided`.

D. Support for Locks

CAF includes support for locks in order for an application to manage exclusive access to remotely-accessible data objects. Locks are required to be coarrays, and an image may acquire and release a lock at any other image. For example, for a lock that is declared with `type(lock_type) :: lck[*]`, an image may acquire the lock `lck` at image *j* using the statement `lock(lck[j])`, and it may release it using the statement `unlock(lock[j])`. Note that another image may simultaneously acquire the corresponding `lck` lock at another image. OpenSHMEM also provides support for locks, which must be symmetric variables. However, in OpenSHMEM symmetric locks are treated as a single, logically *global* entity, meaning that the library does not permit acquiring or releasing a lock at a specific PE. Consequentially, implementing CAF locks using OpenSHMEM's lock support would not be feasible. For a program running with N images, each image would need to allocate a symmetric array of size N for each declaration of a lock. A more space-efficient approach is necessary.

In our implementation, we adapted the well-known MCS lock algorithm [19] for scalable shared memory systems, designed to avoid spinning on non-local memory locations, to efficiently handle locks in CAF programs. In this algorithm, a distributed FIFO queue is used to keep track of images which are contending for lock `lck`, where each image holds a node within the queue called a *qnode* and `lck` contains an internal *tail* field which points to the last qnode in the queue. A *locked* field within the qnode indicates whether the lock is still being held by a predecessor image. A *next* field refers to a successor image which should be notified after the local image acquires and releases the lock, by resetting its qnode's *locked* field.

At any moment, an image may have up to M+1 allocated qnodes, corresponding to M currently held locks and potentially one additional lock it is currently waiting to acquire. We use a hash table lookup, with the hash key being the tuple (`lck, j`) for the lock `lck[j]`, to check if a lock is currently held during execution of the `lock` statement or to obtain the local qnode for a held lock during execution of the `unlock` statement. When an image executes the `lock` statement, it will allocate a local qnode for the lock, and then it will use an atomic fetch-and-store operation to update the lock variable's tail pointer to point to its qnode. The fetched value will point to the prior tail, i.e. its predecessor's qnode. If this value is not ∅, it uses this to set its predecessor's *next* field. It will then locally spin on its qnode's *locked* field until it is reset to 0 by its predecessor. When an image executes the `unlock` statement, it uses an atomic compare-and-swap operation on the lock variable to conditionally set the *tail* field to ∅ if its qnode is still the tail (no successor yet). If there is a detected successor, then the image will reset the *locked* field of the successor's qnode. Finally, it will deallocate its qnode and remove it from the hash table.

Since the qnode must be remotely accessible to other images, it is allocated out of the pre-allocated buffer space for non-symmetric remote-accessible data. OpenSHMEM provides both atomic fetch-and-store and compare-and-swap for implementing the updates to the lock variable. The *tail* and *next* fields, functioning as pointers to qnodes belong to a remote image, are represented using 20 bits for the image index, 36 bits for the offset of the qnode within the remote-accessible buffer space, and the final 8 bits reserved for other flags. By packing this "remote pointer" within a 64-bit representation, we can utilize support for 8-byte remote atomics provided by OpenSHMEM.

V. EXPERIMENTS

This section discusses experimental results for our implementation of UHCAF using OpenSHMEM as a communication layer on three sets of benchmarks: (1) the PGAS Microbenchmark suite [20], which contains code designed to test the performance and correctness for put/get operations and locks written in CAF, (2) a distributed hash table benchmark which makes use of locks, and (3) a CAF version of the Himeno Benchmark.

A. Experimental Setup

We used three different machines for our experimental analysis. The configurations of these machines are given in Table III. For all the experiments involving the UHCAF implementation, we used the OpenUH compiler version 3.40 with CAF runtime implemented over either GASNet or the native SHMEM implementations (Cray SHMEM or MVAPICH2-X SHMEM).

Stampede We used the Stampede supercomputing system at Texas Advanced Computing Center (TACC). We made use of the SHMEM and MPI-3.0 implementations available in MVAPICH2-X version 2.0b in Stampede for our analysis. We also used GASNet version 1.24.0 with IBV conduit for our relative comparisons.

Cluster	Nodes	Processor Type	Cores/Node	Interconnect
Stampede (TACC)	6,400	Intel Xeon E5 (Sandy Bridge)	16	InfiniBand Mellanox Switches/HCAs
Cray XC30	64	Intel Xeon E5 (Sandy Bridge)	16	Dragonfly interconnect with Aries
Titan (OLCF)	18,688	AMD Opteron	16	Cray Gemini interconnect

Table III: Experimental Setup and Machine configuration details

Cray XC30 On the Cray XC30 machine, we used Cray SHMEM version 6.3.1, Cray Fortran compiler 8.2.6, and GASNet version 1.24.0 with Aries conduit for our relative analysis.

Titan Titan is a Cray XK7 supercomputing system at Oak Ridge Leadership Computing Facility (OLCF). We have used Cray SHMEM 6.3.1 and Cray Fortran compiler 8.2.6 and used GASNet version 1.24.0 with Gemini conduit for our analysis.

B. Microbenchmarks

We used the PGAS Microbenchmark suite [20] for analyzing the performance of CAF over OpenSHMEM in various environments described above to measure contiguous and multi-dimensional strided `put` bandwidth as well as performance of locks.

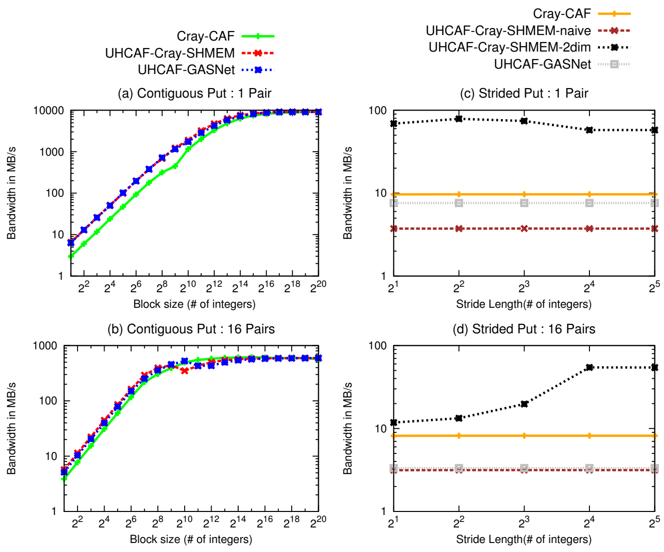


Figure 6: PGAS Microbenchmark Tests on Cray XC30: Put Bandwidth and 2-Dimensional Strided Put bandwidth

1) Contiguous Communications

Based on the RMA design implementations described in Section IV-B, we analyzed the performance of CAF `put/get` tests using Cray Fortran (Cray CAF) vs OpenUH with CAF runtime (UHCAF) executing over Cray SHMEM on the Cray XC30 machine. The same tests are also performed using UHCAF over MVAPICH2-X SHMEM vs UHCAF over GASNet on Stampede. Here, the bandwidth for the `put/get` operations performed between image pairs

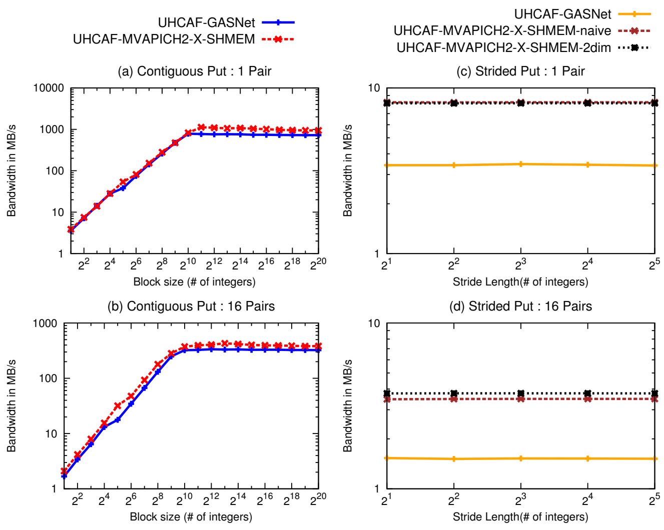


Figure 7: PGAS Microbenchmark Tests on Stampede: Put Bandwidth and 2-Dimensional Strided Put bandwidth

with each image on a different node are captured. We already showed in the previous performance tests described in Section III that the performance of OpenSHMEM is more suitable for short message sizes. These results are confirmed in plots (a) and (b) in Figure 6 Figure 7. We obtain an average of 18% improvement in UHCAF implementation over OpenSHMEM in both the Cray XC30 and Stampede environment.

2) Strided Communications

Results from both Cray XC30 and Stampede to measure the bandwidth of strided `put` are presented in plots (c) and (d) in Figure 6 and Figure 7, respectively. The `2dim_strided` algorithm described in Section IV-C is based on the assumption that the single dimensional strided routines `shmem_iput` or `shmem_iget` are optimized. In this microbenchmark test, we capture the bandwidth for 1-sided strided `put` on an image pair with each image of the pair on a separate node.

Results shown in plots (c) and (d) for Figure 7 show that the 1-dimensional strided routines `shmem_iput` or `shmem_iget` for the MVAPICH2-X SHMEM implementation is not optimized. They also show that UHCAF over MVAPICH2-X SHMEM for the naïve and the `2dim_strided` implementations are the same, because the `shmem_iput` or `shmem_iget` routines are performing multiple `shmem_putmem` or `shmem_getmem` calls underneath. However, OpenSHMEM performs better than the naïve GASNet implementation using multiple `shmem_putmem` or `shmem_getmem` routines.

Results shown in plots (c) and (d) for Figure 6 show the results on the Cray XC30 system. We observe that the `shmem_iput`/`shmem_iget` routines are optimized for Cray SHMEM using DMAPP and thus are more efficient than the naïve implementation using multiple `shmem_putmem` or `shmem_getmem` routines. Using the `2dim_strided` algorithm, we can get better results than with UHCAF using the naïve algorithm or with the Cray CAF implementation. The results show around 3x improvement in bandwidth using UHCAF implementation over Cray SHMEM compared to Cray CAF, and 9x improvement compared to the naïve implementation.

3) Locks

We used a simple microbenchmark where all the images try to repeatedly acquire and then release a lock on image 1. Execution time results on Titan using up to 1024 images (over 64 nodes) are shown in Figure 8 where we compare Cray CAF, UHCAF over GASNet, and UHCAF over Cray SHMEM. Locking with the UHCAF over Cray SHMEM implementation performs better than with the UHCAF over GASNet implementation, especially for number of images $\geq$ 128 images. Moreover, the implementation of locks using UHCAF over Cray SHMEM is more efficient than the one in Cray CAF. On average, using UHCAF over Cray SHMEM is 22% faster than using Cray CAF and 10% faster than using UHCAF over GASNet.

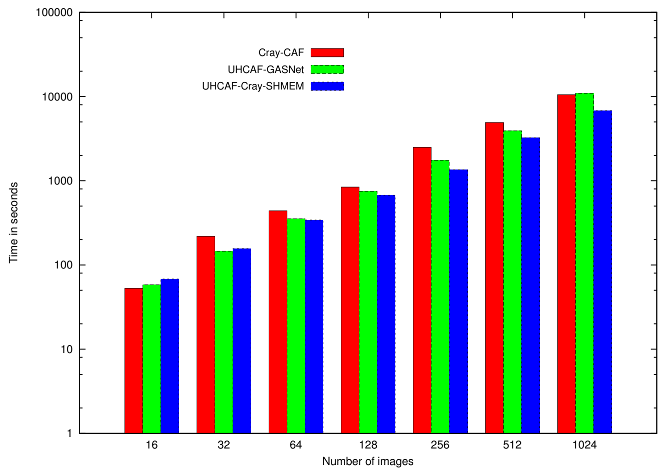


Figure 8: Microbenchmark test for locks on Titan. All images trying to attain and release lock on image 1.

C. Distributed Hash Table (DHT)

In the Distributed Hash Table (DHT) benchmark [21], each image will randomly access and update a sequence of entries in a distributed hash table. In order to prevent simultaneous updates to the same entry, some form of atomicity must be employed; this is achieved using coarray locks. Comparison of results on Titan until 1024 cores (over 64 nodes) are presented in Figure 9. We observe that the DHT benchmark using the UHCAF over Cray SHMEM implementation is 28% faster than the Cray CAF implementation and 18% faster than the UHCAF over GASNet implementation.

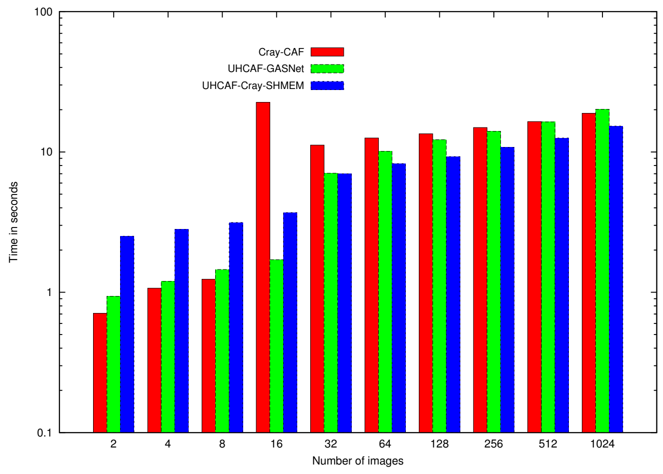


Figure 9: Distributed Hash Table (Titan)

D. Himeno

The Himeno parallel benchmark is used to evaluate the performance of an incompressible fluid analysis code, by using the Jacobi iteration method to solve Poisson's equation. The CAF version of this benchmark is used to measure the performance of the CAF implementation over GASNet versus CAF over MVAPICH2-X SHMEM on Stampede. Figure 10 shows the results of this comparison for up to 2048 total cores (across 128 nodes). We see that the performance of UHCAF over MVAPICH2-X SHMEM is better than UHCAF over GASNet, when the number of images $\geq$ 16. This is because MVAPICH2-X SHMEM is highly optimized for inter-node communication.

The most important part of the Himeno benchmark is the usage of matrix oriented strided data for the halo communication. We found the best implementation for this benchmark is CAF over MVAPICH2-X SHMEM using the naïve algorithm. Note that there was no benefit in using our `2dim_strided` algorithm on the Stampede system, because MVAPICH2-X SHMEM implements the `shmem_iput`/`shmem_iget` routines as a series of contiguous put/get operations, and also the `2dim_strided` does not help for the matrix oriented multi-dimensional strides.

The major difference between the regular multi-dimensional strides and the matrix oriented multi-dimensional strides is the stride in the base dimension. For matrix-oriented strides, the base dimension is a contiguous block. It would be better to perform a single `shmem_putmem` than a single `shmem_iput` in the base dimension for this case. This is reason why the Himeno benchmark was not performed on Titan or Cray XC30 machine. On an average, we obtain around 6% better performance and to the maximum we obtain 22% better performance with UHCAF over MVAPICH2-X SHMEM compared to the UHCAF over GASNet implementation.

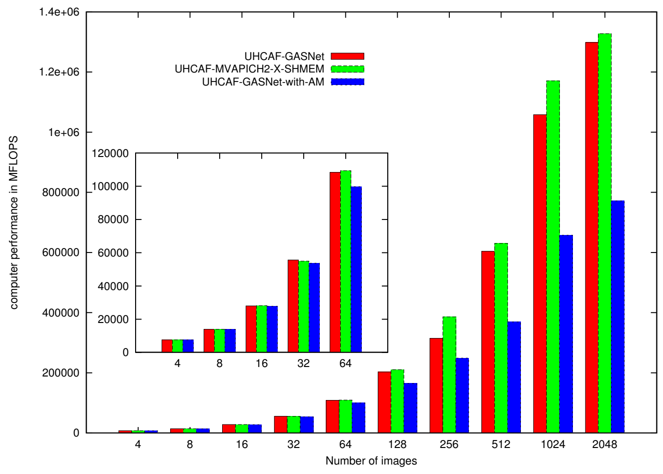


Figure 10: CAF Himeno Benchmark Performance Tests in Stampede

VI. Related Work

Our optimizations on the multi-dimensional strided algorithm relies on the indexed, 1-dimensional strided put/get routines being optimized in OpenSHMEM. Though we cannot expect this optimization to be available in all the OpenSHMEM implementations, there are works [27] [28] related to these optimizations for non-contiguous derived data types which can be applied for OpenSHMEM 1-dimensional strided optimization as well.

[14] stresses on the importance of a generalized communication layer for PGAS programming models. There they emphasize the importance of using DMAPP as a unified communication layer. A common unified communication layer also eases the interoperability between different programming models.

In [29], PGAS models are combined with MPI implementations to obtain better performance results for Gyrokinetic Fusion Applications. Various one-sided algorithms have been developed using CAF and integrated into the best performing algorithm using the preexisting MPI codes to obtain better performance. Hence, a hybrid application using PGAS and MPI is not uncommon. Yang et al. [24] have shown the importance of interoperability between CAF and MPI using their CAF 2.0 [6] implementation over MPI-3.0.

A comparative study on different parallel programming models has been presented in [30]. It is shown that PGAS models, in particular OpenSHMEM, has the potential to obtain better performance results than MPI implementations. Also, developing specific and highly optimized PGAS models for different architecture would be a critical task. The solution to this issue is to use a common base for all the different PGAS implementations which can handle all the low-level, architecture-specific optimizations. For example, Cray has implemented all of their PGAS models (i.e. UPC, CAF and OpenSHMEM) with DMAPP [18]. Libraries like DMAPP are vendor specific; creating a standard among different vendor libraries would be highly improbable. OpenSHMEM may be considered as a potential candidate for this purpose. This will make it possible to increase the potential for developing many interoperable hybrid applications with OpenSHMEM and other PGAS models.

VII. Conclusion and Future Work

In this paper, we demonstrated how OpenSHMEM can function as a communication layer for CAF. The addition of OpenSHMEM to the runtime implementation provided in the OpenUH CAF compiler (UHCAF) was done via two steps: (1) direct mapping of CAF language features to OpenSHMEM, including allocation of remotely accessible objects, one-sided communication, and various types of synchronization, and (2) evaluation of various algorithms we developed for implementing non-contiguous communications and acquisition and release of remote locks using the OpenSHMEM interface. We obtain an average of 18% of improvement of bandwidth for the put/get operations performed between image pairs using UHCAF implemented over OpenSHMEM in both the Cray XC30 and Stampede environments. For strided communications, we achieved around 3x improvement of bandwidth using UHCAF implementation over Cray SHMEM compared to performance with the Cray Fortran implementation. The result of our lock microbenchmark was that UHCAF over Cray SHMEM executed 22% faster than Cray Fortran implementation and 10% faster than the UHCAF over GASNet implementation. Performance improvement relative to the original UHCAF over GASNet implementation was also observed for the Himeno CAF benchmark.

We plan as future work to further improve the multi-dimensional strided communications algorithm. We intend to account for more parameters to negotiate the tradeoff between locality and minimizing the number of single calls, such as cache line size and number of elements in each dimension. We plan also to utilize the `shmem_ptr` operation to convert intra-node accesses into direct load/store instructions and to develop hybrid codes with CAF and OpenSHMEM.

VIII. Acknowledgments

This work is supported through funding from Total, and the authors would like to thank Total, TACC, and OLCF for providing compute resources for the experiments.

References

[1] G. Almasi, in *Encyclopedia of Parallel Computing*, D. A. Padua, Ed., 2011.

[2] W. W. Carlson, J. M. Draper, and D. E. Culler, “S-246, 187 Introduction to UPC and Language Specification.”

[3] R. W. Numrich and J. Reid, “Co-array Fortran for Parallel Programming,” *SIGPLAN Fortran Forum*, vol. 17, no. 2, Aug. 1998.

[4] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, “X10: An Object-oriented Approach to Non-uniform Cluster Computing,” in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA’05, 2005.

[5] B. Chamberlain, D. Callahan, and H. Zima, “Parallel Programmability and the Chapel Language,” *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, Aug. 2007.

[6] J. Mellor-Crummey, L. Adhianto, W. N. Scherer, III, and G. Jin, “A new vision for coarray fortran,” in *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, ser. PGAS ’09, 2009.

[7] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, “Introducing openshmem: Shmem for the pgas community,” in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, ser. PGAS ’10, 2010.

[8] J. Nieplocha, R. Harrison, and R. Littlefield, “Global Arrays: a portable ldquo;shared-memory rdquo; programming model for distributed memory computers,” in *Supercomputing ’94., Proceedings*, Nov 1994.

[9] D. Bonachea, “GASNet Specification, V1.1,” Tech. Rep., 2002.

[10] Mellanox, “Specification: Mellanox InfiniBand Verbs API,” Tech. Rep.

[11] W. Huang, G. Santhanaraman, H. Jin, Q. Gao, and D. Panda, “Design and Implementation of High Performance MVAPICH2: MPI2 over InfiniBand,” 2007.

[12] J. Nieplocha and B. Carpenter, “ARMCI: A Portable Remote Memory Copy Libray for Ditributed Array Libraries and Compiler Run-Time Systems,” in *Proceedings of the 11 IPPS/SPDP’99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, 1999.

[13] Cray, “Using the GNI and DMAPP APIs,” Tech. Rep., February 2014.

[14] M. ten Bruggencate and D. Roweth, “DMAPP: An API for One-Sided Programming Model on Baker Systems,” Cray Users Group (CUG), Tech. Rep., August 2010.

[15] J. R. Hammond, S. Ghosh, and B. M. Chapman, “Implementing OpenSHMEM Using MPI-3 One-Sided Communication,” in *Proceedings of the First Workshop on OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools - Volume 8356*, ser. OpenSHMEM 2014, 2014.

[16] “OSU SHMEM implementation over MVAPICH2-X,” http://mvapich.cse.ohio-state.edu/overview/.

[17] B. Chapman, D. Eachempati, and O. Hernandez, “Experiences Developing the OpenUH Compiler and Runtime Infrastructure,” *Int. J. Parallel Program.*

[18] A. Vishnu, M. ten Bruggencate, and R. Olson, “Evaluating the Potential of Cray Gemini Interconnect for PGAS Communication Runtime Systems,” in *Proceedings of the 2011 IEEE 19th Annual Symposium on High Performance Interconnects*, ser. HOTI ’11, 2011.

[19] J. M. Mellor-Crummey and M. L. Scott, “Algorithms for Scalable Synchronization on Shared-memory Multiprocessors,” *ACM Trans. Comput. Syst.*, vol. 9, no. 1, Feb. 1991.

[20] “HPCTools PGAS-Microbenchmarks,” https://github.com/uhhpctools/pgas-microbench.

[21] C. Maynard, “Comparing One-Sided Communication With MPI, UPC and SHMEM,” Cray Users Group (CUG), Tech. Rep., 2012.

[22] “Himeno Parallel Benchmark,” http://accc.riken.jp/2444.htm.

[23] A. Fanfarillo, T. Burnus, V. Cardellini, S. Filippone, D. Nagle, and D. Rouson, “OpenCoarrays: Open-source Transport Layers Supporting Coarray Fortran Compilers,” in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS ’14, 2014.

[24] C. Yang, W. Bland, J. Mellor-Crummey, and P. Balaji, “Portable, mpi-interoperable coarray fortran,” *SIGPLAN Not.*

[25] S. Poole, O. Hernandez, J. Kuehn, G. Shipman, A. Curtis, and K. Feind, “Openshmem - toward a unified rma model,” in *Encyclopedia of Parallel Computing*, D. Padua, Ed., 2011.

[26] J. R. Hammond, “Towards a Matrix-oriented Strided Interface in OpenSHMEM,” in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS ’14, 2014.

[27] J. Nieplocha, V. Tipparaju, and M. Krishnan, “Optimizing Strided Remote Memory Access Operations on the Quadrics QsNetII Network Interconnect,” in *Proceedings of the Eighth International Conference on High-Performance Computing in Asia-Pacific Region*, ser. HPCASIA ’05, 2005.

[28] S. Byna, X.-H. Sun, R. Thakur, and W. Gropp, “Automatic Memory Optimizations for Improving MPI Derived Datatype Performance,” in *Proceedings of the 13th European PVM/MPI User’s Group Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. EuroPVM/MPI’06, 2006.

[29] R. Preissl, N. Wichmann, B. Long, J. Shalf, S. Ethier, and A. Koniges, “Multithreaded Global Address Space Communication Techniques for Gyrokinetic Fusion Applications on Ultra-Scale Platforms,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’11, 2011.

[30] H. Shan and J. P. Singh, “A comparison of mpi, shmem and cache-coherent shared address space programming models on a tightly-coupled multiprocessors,” *Int. J. Parallel Program.*, vol. 29.