Portable, MPI-Interoperable Coarray Fortran

Chaoran Yang
Department of Computer Science
Rice University
chaoran@rice.edu

Wesley Bland
Math. and Comp. Sci. Division
Argonne National Laboratory
wbland@mcs.anl.gov

John Mellor-Crummey
Department of Computer Science
Rice University
johnmc@rice.edu

Pavan Balaji
Mathematics and Computer Science Division
Argonne National Laboratory
balaji@mcs.anl.gov

Abstract

The past decade has seen the advent of a number of parallel programming models such as Coarray Fortran (CAF), Unified Parallel C, X10, and Chapel. Despite the productivity gains promised by these models, most parallel scientific applications still rely on MPI as their data movement model. One reason for this trend is that it is hard for users to incrementally adopt these new programming models in existing MPI applications. Because each model use its own runtime system, they duplicate resources and are potentially error-prone. Such independent runtime systems were deemed necessary because MPI was considered insufficient in the past to play this role for these languages.

The recently released MPI-3, however, adds several new capabilities that now provide all of the functionality needed to act as a runtime, including a much more comprehensive one-sided communication framework. In this paper, we investigate how MPI-3 can form a runtime system for one example programming model, CAF, with a broader goal of enabling a single application to use both MPI and CAF with the highest level of interoperability.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed, and parallel languages; D.3.4 [*Programming Languages*]: Processors—Compilers, Runtime environments

Keywords Coarray Fortran; MPI; PGAS; Interoperability

PPOPP '14, February 15–19, 2014, Orlando, Florida, USA.
Copyright © 2014 ACM 978-1-4503-2656-8/14/02…$15.00.
http://dx.doi.org/10.1145/2555243.2555270

1. Introduction

Message Passing Interface (MPI) is the *de facto* standard for programming large-scale parallel programs today. There are several reasons for its success including a rich and standardized interface coupled with heavily optimized implementations on virtually every platform in the world. MPI's philosophy is simple: it's primary purpose is not to make simple programs easy to implement; rather, it is to make complex programs possible to implement.

In recent years a number of new programming models such as Coarray Fortran (CAF) [24], Unified Parallel C (UPC) [29], X10 [25], and Chapel [7] have emerged. These programming models feature a Partitioned Global Address Space (PGAS) where data is accessed through language load/store constructs that eventually translate to one-sided communication routines at the runtime layer. Together with the obvious productivity benefits of having direct language constructs for moving data, these languages also provide the potential for improved performance by utilizing the native hardware communication features on each platform.

Despite the productivity promises of these newer programming models, most parallel scientific applications are slow to adopt them and continue to use MPI as their data movement model of choice. One of the reasons for this trend is that it is not easy for programmers to incrementally adopt these new programming models in existing MPI applications. Because most of these new programming models have their own runtime systems, to incrementally adopt them, the user would need to initialize and use separate runtime libraries for each model within a single application. Not only does this duplicate the runtime resources, e.g. temporary memory buffers and metadata for memory segments, but it is also error-prone with a possibility for deadlock if not used in a careful and potentially in a platform-specific manner (more details to follow).

Why do we need Interoperable Programming Models? Because of MPI's vast success in the parallel programming arena, a large ecosystem of tools and libraries has developed around it. There are high-level libraries using MPI for complex mathematical computations, I/O, visualization and data analytics, and almost every other paradigm used by scientific applications. One of the drawbacks of other programming models is that they do not enjoy such support, making it hard for applications to utilize them. For example, a new CAF application cannot directly utilize a math library (such as PETSc or Trillinos) that is written with MPI. Similarly, if one developed a high-performance FFT library in CAF, an MPI application cannot directly plug it in if the underlying runtime systems are not interoperable.

Such requirements are already seen in a number of scientific applications today. For example, in [23], Preissl et. al. identified that the Gyrokinetic Tokamak Simulation code that is based on MPI+OpenMP can naturally benefit from the language constructs in CAF that enable direct remote data accesses. Consequently, they modified their application to further hybridize it with MPI+CAF+OpenMP, and demonstrated performance improvements with such a model. QMCPACK [16], an MPI-based quantum montecarlo package developed by Oak Ridge National Laboratory, and GFMC [17, 22], an MPI-based nuclear physics montecarlo simulation developed by Argonne National Laboratory, both demonstrate such requirements as well. Specifically, both of these applications rely on large arrays that reside on each node for their core sequential computations, and they use MPI to communicate data between processes. As problem sizes grow, however, these arrays are becoming too large to reside on a single node, thus requiring the memory of multiple nodes to accommodate them. Hybridizing with MPI+CAF provides a natural extension for these MPI applications where they can simply define these arrays as CAF coarrays, allowing the runtime system to distribute them across nodes and convert load/store accesses of these arrays to remote data access operations.

Challenges in Interoperable Programming Models. There are several challenges in facilitating multiple programming models to interoperate with each other. Some of these aspects are related to the programming semantics and the execution model itself. For instance, for a programming model such as Chapel, which exposes a completely dynamic execution model where tasks can fork other tasks on demand, to interoperate with a more static model such as MPI, a clear definition of what the user is allowed to do needs to be specified. This part is not the focus of this paper. Instead we focus on interoperation of CAF and MPI, which have similar execution model semantics, with a number of images/processes that move data between each other.

Even for programming models with similar execution semantics, various challenges exist that make interoperability hard. For example, today, each programming model uses its own separate runtime system. For example, GASNet [5] is a common runtime system used by Berkeley UPC, CAF 2.0, and others. MPI, on the other hand, uses its own platform-specific runtime system. An application that uses multiple programming models would need to initialize and use multiple runtime systems in a single application, thus duplicating resources.

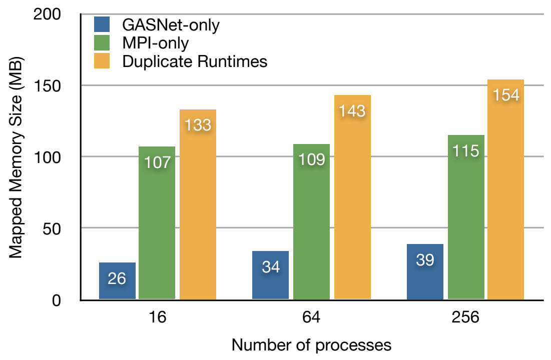

Figure 1. An example of memory usage when using both GASNet and MPI.

```
1  PROGRAM MAY_DEADLOCK
2  USE MPI
3
4  CALL MPI_INIT(IERR)
5  CALL MPI_COMM_RANK(MPI_COMM_WORLD, MY_RANK, IERR)
6
7  IF (MY_RANK .EQ. 0) THEN
8     A(:)[1] = A(:)
9  END IF
10
11 CALL MPI_BARRIER(MPI_COMM_WORLD, IERR)
12 CALL MPI_FINALIZE(IERR)
13 END PROGRAM
```

Figure 2. A CAF program that may deadlock because CAF cannot make progress when the process blocks in MPI.

Figure 1 shows an example of the per-process memory usage when initializing both GASNet and MPI in an application. [1] The memory usage of both libraries grows along with the number of processes. The duplicated runtime resources reduce the resources available to an application, which will eventually hurt performance or prevent the application from running at a larger scale.

Using multiple runtime systems within a single application also makes it hard to reason about the correctness of codes. The semantics of an operation are often well-defined with respect to its own runtime system but are unclear when multiple runtime systems are used. For example, Figure 2 lists a simple CAF program that has the process with rank 0 perform a coarray write (line 8), then has every process participate in an **MPI_BARRIER** (line 11). Depending on the im-

[1] GASNet uses less memory than MPI because it saves data such as metadata of memory segments in user-space buffers.

plementation of CAF, a coarray write operation may require the involvement of the target process to complete. Although such involvement is implicit, it requires the target process to make a runtime call to make progress internally. In this example, because the target process is likely to run into the **MPI_BARRIER** before seeing the write operation, the coarray write on process 0 may never complete. Some implementations of CAF may use a separate progress thread or have better support from the hardware allowing it to make asynchronous progress. This makes the scenario implementation-specific and even harder for users to debug.

Contributions of this paper. One possible solution to the interoperability issues described above is to use MPI as the runtime system for CAF, thus allowing all data movement to be funneled through a common runtime library. However, the use of MPI as a runtime system was previously deemed impossible for these PGAS programming models because MPI was considered inadequate to implement these models. Bonachea and Duell [6] put together a comprehensive list of reasons why the remote memory access (RMA) features introduced in the MPI-2 Standard fall short of the task of serving as a compilation target of PGAS languages. The recently released MPI-3 Standard, however, adds several new capabilities including a much more comprehensive one-sided communication (or RMA) model. These new additions not only address the critiques raised about MPI-2 RMA but also provide new functionality with performance benefits over existing PGAS runtime systems. But whether MPI-3 can live up to the goal of forming a runtime basis for PGAS programming models with these new additions is yet to be validated in practice.

In this paper, we investigate the capability of MPI in serving as the basis of a PGAS programming model such as CAF. We redesigned the runtime system of CAF 2.0, which was originally built on top of GASNet, to use MPI-3. The paper describes various design choices we made during this process. Further, we present the performance of CAF over MPI-3 relative to that of the original CAF implementation, and demonstrate that the MPI runtime system can achieve a comparable or better performance in several cases. We also point out some cases where the performance falls short of the existing implementation; in such cases we present a detailed analysis of the performance difference.

This paper is organized as follows: Section 2 presents an overview of CAF-2.0 and MPI-3 and serves as background knowledge for the later discussion of the paper. Section 3 describes the design of the CAF-MPI runtime system. In Section 4 we evaluate our implementation using three HPC Challenge Benchmarks [11] and the CGPOP miniapp [27], which uses both CAF and MPI. In Section 5 we discuss possible extensions to the MPI-3 RMA interface that may provide further benefits when using MPI as the basis of PGAS languages. We discuss related work in Section 6 and present a summary of our work in Section 7.

2. Overview of CAF and MPI-3

To make the paper relatively self-contained, this section briefly introduces the necessary context for this paper. We will give an overview of CAF [18] and MPI-3 [19]. For further background information, refer to the cited papers.

2.1 Coarray Fortran 2.0

We choose Coarray Fortran 2.0 (CAF 2.0) [18], developed by Rice University, as our PGAS programming model to test the new capabilities of MPI-3. CAF 2.0 is an open source implementation of CAF and also contains a richer set of PGAS extensions than CAF-1 that is integrated into the Fortran 2008 standard. CAF 2.0 better represents the semantics and requirements of modern PGAS languages and would provide a better picture of the appropriateness of MPI-3 as a basis for these models. We describe the PGAS extensions in CAF 2.0 that are not included in Fortran 2008 standard in this section.

Teams. In CAF 2.0, a team is a first-class entity that represents a group of processes, analogous to an MPI communicator. Teams in CAF 2.0 serve three purposes: (a) the set of images in a team serves as a domain onto which coarrays may be allocated; (b) a team provides a name space within which images can be indexed by their relative rank within that team; (c) a team provides an isolated domain for members of a team to communicate and synchronize collectively. When a CAF program starts, all images belong to the same team named `TEAM_WORLD`, and new teams can be created with the `team_split` operation.

Synchronization. CAF 2.0 provides a rich set of synchronization constructs. These constructs are designed to enable programmers to structure correct parallel programs while allowing maximum opportunity to hide communication latency with computation. These constructs were previously introduced by Yang et. al. in [32].

Pair-wise synchronization between process images in CAF 2.0 can be achieved with *events*, a first-class entity in CAF 2.0. An event must be initialized with `event_init` before any other operation can occur. Events can be allocated as coarrays to enable them to be posted by other process images. One posts an event with the `event_notify` operation. An `event_wait` operation blocks execution of the current thread until the event is posted. `event_trywait` is a non-blocking operation which tests whether an event is posted and returns immediately.

To synchronize asynchronous operations (described in the next subsection), CAF 2.0 provides two synchronization models: the *implicit synchronization* model, and the *explicit synchronization* model. All asynchronous operations in CAF 2.0 optionally accept event arguments. Asynchronous operations invoked with an event argument are *explicitly synchronized operations*. The events passed to asynchronous operations are notified by the runtime system when the synchronization point is reached. One can test or wait for the event to synchronize these operations.

One can also use the implicit synchronization model to synchronize asynchronous operations. CAF 2.0 provides `cofence` statements and `finish` blocks to serve this purpose. A `cofence` statement blocks the current thread until all asynchronous operations issued before the statement complete locally. More specifically, a `cofence` ensures all local buffers used by asynchronous operations issued previously are ready to be reused. A `cofence` also serves as a compiler barrier; operations are not allow to be reordered across a `cofence` statement.

`finish` is a block-structured global synchronization construct in CAF 2.0, demarcated by `finish` and `end finish` statements. `finish` in CAF 2.0 is similar to the `finish` block in X10; it ensures that all asynchronous operations issued within the block are globally complete before current thread exits from the block. However, unlike the `finish` statement in X10, a `finish` statement in CAF 2.0 is a collective operation. A `finish` statement takes a team variable, and each process image within the associated team should create a `finish` block that matches those of its teammates. When the thread exits from the block, all operations issued by all images within the team are globally complete. `finish` blocks can be nested within each other; this is useful because it allows asynchronous operations issued within the outer `finish` block to proceed while the process is waiting for the inner `finish` block to complete, which yields more opportunity for communication-computation overlapping.

Asynchronous operations. CAF 2.0 provides three categories of asynchronous operations: *asynchronous copy*, *asynchronous collectives*, and *function shipping*. Scherer et. al. and Yang provide detailed descriptions of these operations elsewhere [26, 31].

A `copy_async` operation transfers data from a buffer (the source) to another buffer (the destination) asynchronously. The source and destination may be local or remote coarrays. An asynchronous copy takes three optional event arguments: the predicate event, the source event, and the destination event. The copy may proceed after its predicate event is posted. The source event, when posted, indicates the source buffer is ready to be reused. Notification of the destination event indicates the data has been delivered to the destination; the destination is ready to be read.

Asynchronous collectives have similar functionality as their synchronous versions except that they are non-blocking. For example, `team_reduce_async` performs a reduction across a team asynchronously. An asynchronous collective takes two optional event arguments: the data completion event and the operation completion event. The data completion event indicates the local buffer is ready to be read, and the operation completion event indicates the local buffer is ready to be modified.

Function shipping is an new addition of CAF 2.0 that enables programmers to transfer computation to where data is located, rather than moving data towards computation. The design of CAF 2.0 function shipping mechanism allows the shipped function to perform the full range of CAF 2.0 operations, e.g., spawning more functions, performing blocking communications, and synchronization.

2.2 MPI-3 Remote Memory Access Extensions

The MPI-3 Standard improved the MPI-2 RMA functionality in a number of ways including the addition of atomic operations, request-generating RMA operations, new synchronization operations in passive target epochs, a new unified memory model, and new window types. This section serves as a introductory description of the features relevant to this paper.

One-sided atomic operations. MPI-3 RMA provides several operations that enable one to manipulate data in target windows atomically. **MPI_ACCUMULATE** and **MPI_GET_ACCUMULATE** operations allow remote atomic updates of data, with **MPI_GET_ACCUMULATE** further fetching the original remote data back to the origin. The MPI-3 Standard also defines **MPI_FETCH_AND_OP**, which is a special case of **MPI_GET_ACCUMULATE** for single element predefined datatypes (to provide a fast-path for such operations), and **MPI_COMPARE_AND_SWAP**, which provides a remote compare-and-swap operation.

Request-generating operations. MPI-3 adds 'R' versions of most of the RMA operations that return request handles, such as **MPI_RPUT**, **MPI_RGET**, and **MPI_RACCUMULATE**. The request handles returned by these operations can be later passed to MPI request completion routines, such as **MPI_WAIT**, to manage local completion of the operation. These request-generating operations may be used only in passive target synchronization epochs.

MPI-allocated windows. MPI-3 adds several new routines for creating windows. **MPI_WIN_ALLOCATE** allows the MPI implementation to allocate memory for a window, rather than using a user allocated buffer. When MPI allocates memory for a window, it has the opportunity to allocate special memory regions such as aligned memory segments across a group or shared memory regions, which reduces the overhead of RMA operations performed on that window. **MPI_WIN_ALLOCATE_SHARED** will allocate memory that is shared by all processes in the group of the communicator when the processes belong to the same shared-memory node. **MPI_WIN_CREATE_DYNAMIC** creates a window without memory attached; one can dynamically attach memory later with **MPI_WIN_ATTACH**.

Unified memory model. MPI-2 RMA assumes no coherence in the memory subsystem or network interface, resulting in logically distinct *public* and *private* copies of a window. This conservative model (the *separate* model) is a poor match for systems where coherent memory subsystems are available. The new *unified* memory model added in MPI-3 better exposes these hardware capabilities to the user. This assumption relaxes several restrictions present in the *separate* model such as access to non-overlapping regions in a window by an **MPI_PUT** and a regular store operation concurrently, thus allowing for higher concurrency in access to the window data.

Passive target synchronization. MPI-3 adds a pair of locking routines to lock and unlock all targets of a window simultaneously. With **MPI_WIN_LOCK_ALL**, finer-grained synchronization can be achieved, for example using request-generating operations described earlier in this section. Two new synchronization routines, **MPI_WIN_FLUSH** and **MPI_WIN_FLUSH_ALL**, were also added that allow the user to complete all operations initiated by an origin process to a specified target or to all targets.

3. Designing CAF over MPI-3

In this section, we describe the key details of a new implementation of CAF 2.0 using MPI-3 as its communication substrate. CAF 2.0 features that are trivial to implement with MPI, e.g., teams and collectives, are left out due to space constrains. In the rest of the paper, we will sometimes refer to our implementation of CAF as CAF-MPI and the original CAF 2.0 as CAF-GASNet.

3.1 Coarrays

Coarrays are the main addition of CAF to Fortran 95. Coarrays add a codimension to a plain Fortran array. The codimension indicates the process image on which an array is located. Coarrays on remote images can be accessed with a Fortran 95 array section syntax plus a codimension. Thus, reading from or writing to a remote coarray is a one-sided operation that is mapped naturally to **MPI_GET** and **MPI_PUT**.

The original CAF 2.0 runtime system represents a reference to a remote memory location with a `(image_ID, address)` tuple. Because MPI RMA hides the absolute address of a remote memory in the window object, and currently provides no interface to access this information, we augmented the tuple with a remote coarray location that includes with an window object and the offset from the start of the window. Thus, the new remote memory reference become a `(window, rank, displacement)` tuple.

CAF-MPI allocates coarrays using **MPI_WIN_ALLOCATE**. With **MPI_WIN_ALLOCATE**, an MPI implementation can potentially improve performance by allocating aligned memory segments or shared memory regions for the window.

The semantics of coarray read and write operations require the effect of the operation to be globally visible after the operation completes. Proper synchronization is needed to ensure this semantic. Because the active target synchronization model in MPI requires the participation of the target processes, it is more convenient to use the passive target synchronization model in CAF. With the new additions of synchronization routines in the MPI-3 passive target model, we can lock all targets with **MPI_WIN_LOCK_ALL** when a coarray is allocated. Blocking read and write operations in CAF-MPI use **MPI_WIN_FLUSH** to ensure remote completion. The target processes of a window are only unlocked when the coarray is deallocated.

3.2 Active Messages

Active Messages (AM) [30] are an integral component of GASNet. The CAF runtime system of makes heavy use of them in various places, e.g., function shipping and event mechanisms. For CAF-MPI, we implemented Active Messages using MPI's two-sided communication routines. Our design of the AM interface is a near-exact replica of the AM interface in the GASNet core API to maintain maximum compatibility with the original CAF 2.0 runtime system. GASNet's AM API consists of three categories of AM: the short Active Messages carry only a few integer arguments, the medium Active Message can carry an opaque data payload in addition to integer arguments, and the long Active Message can also carry an opaque data payload but user needs to specify a predetermined address in the target process's memory space to receive the data payload. The number of integer arguments that a short AM can carry can be queried with `gasnet_AMMaxArgs()` function; the size of the data payload a medium AM can carry can be queried with `gasnet_AMMaxMedium()`. Providing different APIs for different message sizes allows the compiler to generate more efficient code when the message size is known at compile time.

Since AMs are used in many places in CAF 2.0's runtime system, its performance is critical. To ensure a fast message injection rate for AM, we used **MPI_ISEND** to send all messages. Integer arguments of medium data payload are internally buffered to use **MPI_ISEND**, and waiting for local completion of the send operation is delayed until the next synchronization point. The data payload of long AMs are sent with an extra blocking **MPI_SEND**. Theoretically, this extra send could be replaced by an **MPI_PUT** operation to avoid the internal data buffering within MPI. However, because the current MPI standard does not provide a mechanism to notify the target on the arrival of an **MPI_PUT**, it is hard to ensure that the AM is invoked only after its data payload arrives. Hence we stayed with the **MPI_SEND** based design in our approach.

3.3 Asynchronous Operations

The original CAF 2.0's asynchronous progression model is based on a common progress engine that all aspects of the CAF runtime plug into. Thus, if a process is waiting on one CAF event, the runtime can make progress on other operations that are internally queued up. This progress model is similar to what most runtime systems, including MPI, use.

When redesigning CAF's runtime system with MPI, one of the restrictions we faced was with the asynchronous progress engine as it relates to MPI RMA operations. Specifically, the MPI-3 Standard does not provide a mechanism to test for remote completion for all MPI RMA operations. The 'R' versions of MPI RMA operations (e.g., **MPI_RPUT** and **MPI_RGET**) provide requests that can be tested or waited on for completion but completion of this request only refers to local completion for **MPI_RPUT** and **MPI_RACCUMULATE**, while it refers to both local and remote completion for **MPI_RGET** and **MPI_RGET_ACCUMULATE**. For remote completion of **MPI_PUT** and **MPI_ACCUMULATE**, MPI-3 only provides **MPI_WIN_FLUSH** and **MPI_WIN_FLUSH_ALL**, apart from the epoch close operations such as **MPI_WIN_LOCK** and **MPI_WIN_LOCK_ALL**. These operations can be blocking (e.g., when someone else is holding the lock) and do not have a request handler that can be used to test for their completion.

To workaround this issue, we use the following mapping of operations:

1. For one-sided communication operations, if no local or remote completion event is requested, we use **MPI_PUT** and **MPI_GET**.
2. If a local or remote completion event is requested for GET-style operations, we use **MPI_RGET** which returns an MPI request on which we can wait or test.
3. If only a local completion event is requested for a PUT-style operation, we use **MPI_RPUT** which returns an MPI request on which we can wait or test.
4. If a remote completion event is requested for a PUT-style operation, we use active messages (that are based on **MPI_SEND** and **MPI_RECV** as described in Section 3.2) to transfer data in the source buffer to the target process, copy data into the destination buffer, then post the destination event associated with the asynchronous operation. This option is obviously not as efficient as a direct **MPI_PUT** or **MPI_GET**, which can be implemented more efficiently on current network hardware. But it provides the necessary functionality. We will further discuss this limitation of MPI-3 and other possible solutions in Section 5.

3.4 Explicit Event Notification

There are two obvious approaches to implement the event mechanism in CAF-MPI. One approach is to leverage the newly added **MPI_FETCH_AND_OP** operation in MPI-3 to notify an event and use the **MPI_COMPARE_AND_SWAP** operation to busy-wait for the event to be posted in `event_wait`. The second approach is to use **MPI_ISEND** to notify an event and use **MPI_RECV** to wait for the event to be posted. The performance implication of these two methods are unclear at this point and may largely depend on the underlying MPI implementation. CAF-MPI used the second method since the performance of **MPI_SEND** and **MPI_RECV** routines are more well-tuned to date, and a two-sided communication model fits more naturally the `event_notify` and `event_wait` model.

`event_wait` is a blocking operation; it blocks the process until the event being waited upon is posted. To be semantically correct, `event_wait` also forbids asynchronous operations in program order after an `event_wait` from being reordered to a position before the `event_wait` (i.e., it also functions as a compiler barrier). This restriction is guaranteed by the code generation process of CAF-MPI source-to-source translator. `event_wait` uses a blocking network polling operation to wait for a specific message to arrive; the polling operation internally uses MPI blocking receive. The benefit of using a blocking polling operation allows the MPI runtime to make progress internally to respond to other processes' requests.

The semantics of the `event_notify` operation specify that when a process posts the notification for an event, the target of the event can only see the notification after all previous operations issued by the notifying process are complete at their respective targets. However, the notification itself is nonblocking to avoid a possible deadlock situation caused by circular `event_wait` and `event_notify` chains. We implement `event_notify` with a release barrier and an short AM request. The release barrier holds a request handle to every asynchronous operation initiated locally. Upon `event_notify`, the release barrier waits to complete all its request handles with **MPI_WAITALL**; this ensures local completion of these operations. Furthermore, a **MPI_WIN_FLUSH_ALL** is required to ensure the remote completion of all previous operations. The actual notification in `event_notify` is performed by sending an AM request to the target process. We use an **MPI_ISEND** to avoid the deadlock possibility mentioned above.

3.5 `cofence` and `finish`

Thanks to the new additions of request-generating RMA operations in MPI-3, local completion of RMA operations can be easily waited upon using the request handles of RMA operations. The `cofence` statement takes an optional argument that a user can use to request local completion notification of PUT or GET operations. Thus, CAF-MPI runtime system internally maintains an array of request handles of implicitly synchronized PUT operations and another array of request handles of implicitly synchronized GET operations. The `cofence` statement translates to an **MPI_WAITALL** call for the local completion of these operations.

`finish` is implemented in the same way as in the original CAF implementation; it uses a distributed termination detection algorithm presented by Yang [31]. Yang's algorithm detects termination by repeatedly performing `SUM` reductions across a team to compute the global difference between the number of shipped functions and the number of completed functions shipped from others. Global termination occurs when a sum reduction yields zero for the difference. This algorithm uses n rounds of reductions in the worst case, where n is the length of the longest chain of function shipping calls in the `finish` block. We also implement a fast version of `finish` that can be used when function shipping is not used in an application. This version of `finish` involves calling **MPI_WIN_FLUSH_ALL** on every window that the local process has touched within the block followed by an **MPI_BARRIER** across the team associated with the `finish` block.

4. Evaluation

We have evaluated CAF-MPI on two platforms: Fusion, an InfiniBand cluster at Argonne National Laboratory, and Edison, a Cray XC30 system at Lawrence Berkeley National Laboratory. The hardware characteristics of these systems are given in Table 1. For each platform, we compare the performance of CAF-MPI with CAF-GASNet. We evaluate the performance of our implementation with three HPC Challenge Benchmarks: HPL, FFT, and RandomAccess [11]. These benchmarks exhibit different communication vs. computation ratios and data access patterns and thus can serve as representative examples for a wide range of applications. We also demonstrate the performance of CAF-MPI with the CGPOP miniapp, which uses both CAF and MPI simultaneously. All benchmarks are compiled with Intel compilers and optimization level "-O3" on both platforms.

4.1 RandomAccess Benchmark

The HPC Challenge RandomAccess benchmark evaluates the rate at which a parallel system can apply read-modify-write updates to randomly indexed entries in a distributed table. Performance of the RandomAccess benchmark is measured in Giga Updates Per second (GUP/s). GUP/s is calculated by identifying the number of table entries that were randomly updated in one second, divided by 1 billion (10^9).

The CAF 2.0 implementation of RandomAccess uses a software routing algorithm that uses a hypercube-based pattern of bulk communication to route updates to the process image co-located with the table index being updated. The CAF 2.0 primitives most heavily used in the RandomAccess benchmark are coarray `write` and `event_notify`.

Because the performance of RandomAccess benchmark largely depends on the performance of one-sided communication of the communication library used. The result of RandomAccess benchmark is a good indication of the overhead of the communication library on top of the underlying network hardware. Figure 3 shows the performance difference of RandomAccess between CAF-MPI and CAF-GASNet on Fusion. The CAF-GASNet version of RandomAccess outperforms the CAF-MPI version by a small constant factor up to 64 cores; this indicates that the overhead of the MPI-3 RMA implementation has a constant overhead for each operation which is higher than the overhead of GASNet RMA.

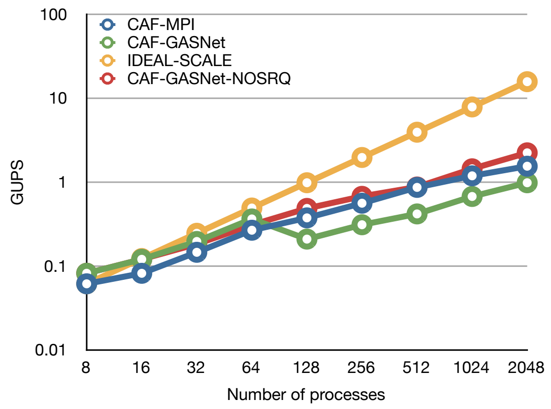

Figure 3. Performance of RandomAccess benchmark on Fusion.

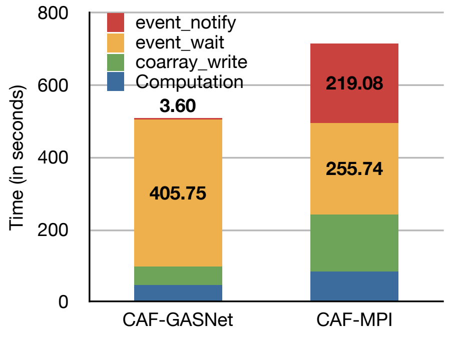

Figure 4. Time decomposition of RandomAccess on 2048 cores on Fusion.

CAF-GASNet's performance drops at 128 cores. Further investigation shows that this is caused by the use of the Shared Receive Queue (SRQ) in GASNet. The default configuration of GASNet library automatically enables SRQ as soon as doing so reduces the memory usage of GASNet. The effect of using SRQ in MVAPICH2 is not noticed in our experiment of up to 2048 cores. By disabling the use of SRQ in GASNet, CAF-GASNet performs roughly the same as CAF-MPI. The CAF-GASNet version shows slightly better scalability than the CAF-MPI version on 1024 and larger cores.

We profiled the 2048-core run of RandomAccess with HPCToolkit [2] to analyze CAF-MPI; the analysis results are shown in Figure 4. The CAF-MPI version of RandomAccess spends around 200 seconds in `event_notify` while the CAF-GASNet version spends almost none. This is because of how `event_notify` is implemented in these two libraries. In CAF-MPI, `event_notify` invokes **MPI_WIN_FLUSH_ALL** to ensure that all previously issued operations have been completed before performing the notification. The current implementation of **MPI_WIN_FLUSH_ALL** in all MPICH derivatives (including MVAPICH and Cray MPI) performs a flush operation on each process within the communicator; hence the execution time of **MPI_WIN_FLUSH_ALL** grows linearly with the number of processes. This is, of course, a

System	Nodes	Cores per Node	Memory per Node	Interconnect	MPI Version
Cluster (Fusion)	320	2 x 4	36GB	InfiniBand QDR	MVAPICH2-1.9
Cray XC30 (Edison)	5,200	2 x 12	64GB	Cray Aries	CRAY-MPICH-6.0.2

Table 1. Experimental platforms and system characteristics.

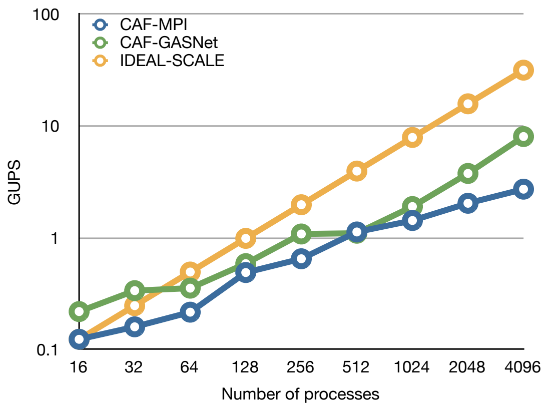

Figure 5. Performance of RandomAccess benchmark on Edison.

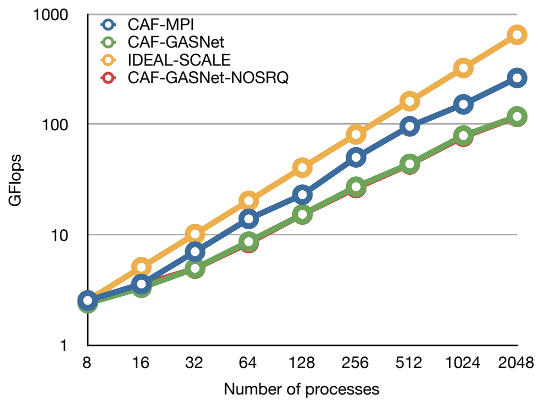

Figure 6. Performance of FFT benchmark on Fusion.

simple performance scalability issue that can be addressed within the MPI implementation, but as of the writing of this paper, this issue exists.

The performance of RandomAccess on Edison, shown in Figure 5, tells roughly the same story as it does on Fusion. The MPI-3 RMA operations in Cray MPI are currently implemented internally with send and receive routines rather than directly leveraging RDMA support in the network. This causes a more obvious performance loss of CAF-MPI version of RandomAccess on Edison. Better implementations of MPI RMA on Cray platforms, such as foMPI [10], already exist and deliver performance competitive with CAF and UPC. However, the foMPI implementation is not integrated into Cray MPI yet.

4.2 FFT Benchmark

The HPC Challenge FFT benchmark measures the ability of a system to overlap computation and communication while calculating a very large Discrete Fourier Transform of size m. Performance of the FFT benchmark is measured in GFLOP/s, with calculated performance defined as $5\frac{m\log_2 m}{t}10^{-9}$, where m is the size of the DFT and t is the execution time in seconds. Parallel FFT algorithms have been well studied in the past [3, 13, 28].

The CAF 2.0 FFT implementation uses a radix-2 binary exchange formulation that consists of three parts: permutation of data to move each source element to the position that is its binary bit reversal; local FFT computation for as many layers of the DFT calculation as can fit in the memory of a single processor; and remote DFT computation for the layers that span multiple processor images. The CAF 2.0 FFT implementation solely uses all-to-all operation for data movement.

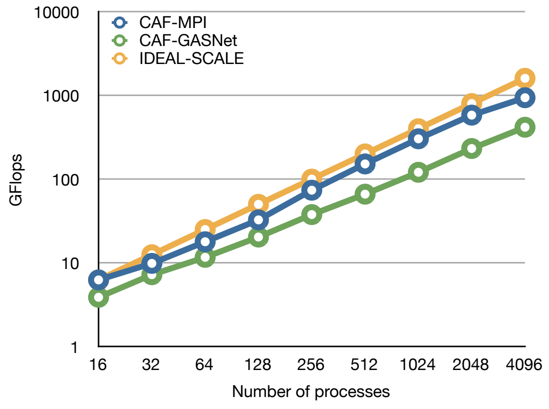

Figure 7. Performance of FFT benchmark on Edison.

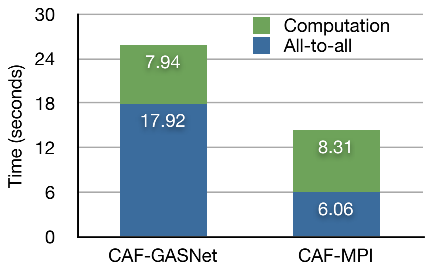

Figure 8. Time decomposition of FFT on 256 cores on Fusion.

Figures 6 and 7 show the performance of the FFT benchmark on Fusion and Edison, respectively. The CAF-MPI version consistently outperforms CAF-GASNet on both plat-

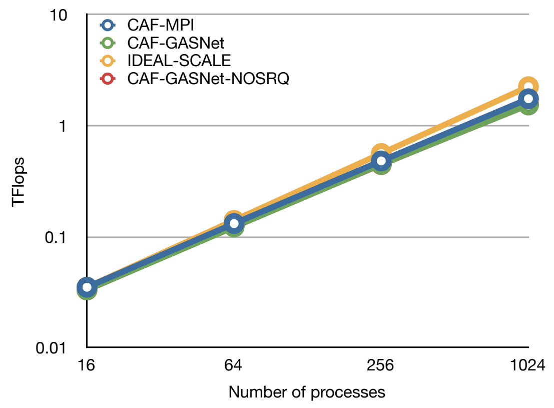

Figure 9. Performance of HPL benchmark on Fusion.

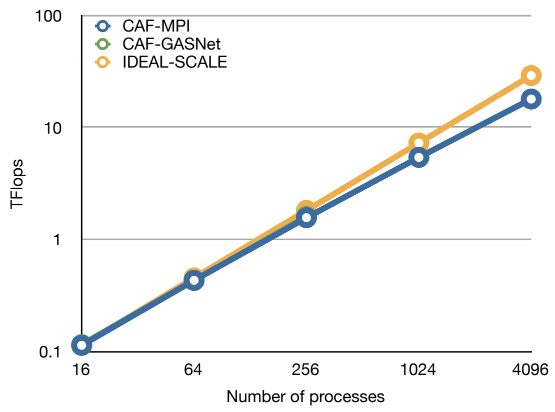

Figure 10. Performance of HPL benchmark on Edison.

forms at different scales. To analyze the performance difference of the two versions of FFT, we use HPCToolkit to profile the benchmark run on 256 cores on Fusion. Figure 8 illustrates how much of the execution time of FFT is spent in local computation and communication (all-to-all to be specific). We can see that the performance difference in two versions of FFT is caused largely by the performance of all-to-all operations used in CAF's runtime system. Because GASNet currently does not have collectives, CAF-GASNet implements alltoall operation with GASNet's PUT, GET, and Active Messages. This is not as well tuned as **MPI_ALLTOALL**, which has been used and well-optimized for many years on most computing platforms today.

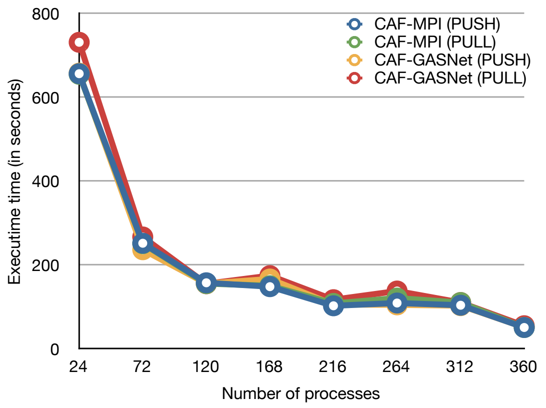

Figure 11. Performance of CGPOP on Fusion.

4.3 HPL Benchmark

The High-Performance Linpack (HPL) [1] benchmark measures the ability of a system to deliver fast floating point execution while solving a system of linear equations. HPL is one of the most common implementations of parallel LU factorization for distributed memory systems. HPL has been previously ported to CAF 2.0 by Jin et. al. [12]. HPL differs greatly from the RandomAccess and FFT benchmarks described earlier because its performance is mostly dominated by computation rather than communication.

Figures 9 and 10 show the performance of HPL on Fusion and Edison, respectively. Between the two CAF implementations, HPL's performance difference is hardly noticeable. Because the performance of HPL benchmark is mostly computation bound, the performance difference of using different communication library has little effect on HPL's overall performance.

4.4 The CGPOP Miniapp

The CGPOP miniapp is the conjugate gradient solver from LANL POP 2.0, an important multi-agency code used for global ocean modeling and is a component within the Community Earth System Model. CGPOP is the performance bottleneck for the full POP application; it implements a version of conjugate gradient that uses a single inner product to iteratively solve for vector x in the equation $Ax = b$. The algorithm consists of a number of linear algebra computations interleaved with two communication steps. The `GlobalSum` function performs a 3-word vector reduction, while `UpdateHalo` function performs a boundary exchange between neighboring sub-domains. The CGPOP miniapp was ported to CAF for evaluating the performance gains of using a PGAS programming model for the performance critical part of POP. It is a hybrid MPI+CAF application that uses CAF coarray primitives for data exchange and **MPI_REDUCE** for reduction operations.

Figures 11 and 12 show the performance results of CGPOP on Fusion and Edison, respectively. On Edison, we can hardly see a difference in the performance of CGPOP between the two CAF implementations. On Fusion, the performance of the two CAF implementations only differs slightly when running on a small number of cores. Since both CAF versions of CGPOP use **MPI_REDUCE**, the only possible cause of performance difference in two versions of CGPOP is the use of PUT and GET operations used by CAF-MPI and CAF-GASNet. The lack of a performance difference indicates that both implementations are equally efficient in the raw **MPI_PUT** and **MPI_GET** operations.

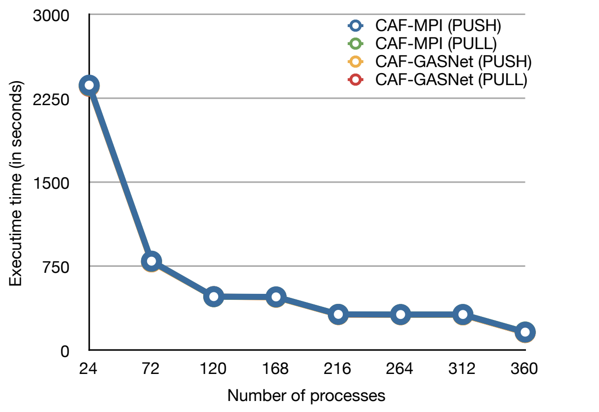

Figure 12. Performance of CGPOP on Edison.

5. Discussion

In this section, we discuss our findings during the process of redesigning CAF 2.0 runtime system to use MPI. Using MPI as the basis of a PGAS runtime system, such as CAF, reveals several advantages of using MPI over a low-level networking layer such as GASNet. We also discovered several features that the current MPI Standard lacks which can be useful in building a PGAS runtime system.

The benefits of MPI's rich interface. Compared with GASNet and other low-level networking layers, MPI provides a much richer set of API functions to support different high-level libraries, languages, and applications. We found this rich interface to be immensely time-saving when building a runtime system that delivers high performance across various platforms. For example, because GASNet does not have collectives, collective operations in CAF are handcrafted in its runtime system on top of GASNet. Not only is this time consuming, but also based on our evaluation, not as performant as the MPI collectives. Because collectives in MPI are well-optimized over the years by different MPI implementations on different platforms, harnessing the performance benefits of these fully-optimized collectives in MPI is valuable for a programming model, such as CAF, which provides both one-sided and two-sided operations.

The need for Active Messages in MPI. One of the goals of building an interoperable runtime system for CAF is to allow the runtime system to make progress for both CAF and MPI when applications make blocking calls in either one. The current implementation of CAF-MPI can stay true to this goal for all parts of its runtime except active messages. Because we built the AM subsystem on top of **MPI_SEND** and **MPI_RECV**, the CAF-MPI runtime has to do further processing of a message to invoke the AM handler. The MPI runtime itself cannot invoke the handler. Thus, if the application is blocked inside an MPI call, no progress is made on such AM handlers.

Having AM support inside MPI can solve this problem. AMs are essential for building runtime systems for various programming models, especially for models such as X10, Chapel, and CAF 2.0 that supports dynamic task parallelism. Zhao et. al. [33, 34] have made an effort in this direction to support MPI-interoperable Active Messages, but such a model is not a part of the current MPI Standard.

The need for non-blocking flush operation in MPI. In passive target epochs, the MPI Standard provides two routines: **MPI_WIN_FLUSH** and **MPI_WIN_FLUSH_ALL** for ensuring remote completion of RMA operations without closing the access epoch. These routines can be blocking calls (e.g., if another image is holding the lock). As discussed in Section 3.3, the blocking **MPI_WIN_FLUSH** operation eliminates the potential opportunity for overlapping the latency of communication with local computation. A nonblocking request-based version of **MPI_WIN_FLUSH**, such as **MPI_WIN_RFLUSH**, can solve this problem. The **MPI_WIN_RFLUSH** operation would start the flush process and return a request handle. The request handle can be later passed to MPI completion routines to wait/test for the completion of the operation.

6. Related Work

CAF and GASNet. CAF has been adopted by the Fortran 2008 committee as the parallel programming model in Fortran. GASNet [5] is a portable PGAS runtime used not only by CAF-2.0, but also by UPC and other parallel libraries. There has been previous work to implement GASNet using MPI, culminating in a GASNet conduit built on AM-MPI [4]. However, the AM-MPI implementation is dated and only uses MPI-1.1, which does not include many of the more recent MPI features which provide a much higher performing implementation, including RMA.

MPI+PGAS Implementations. There has been previous work to implement some PGAS-like libraries on top of MPI. Dinan et. al. [9] implemented the aggregate remote memory copy interface (ARMCI), the runtime layer for Global Arrays (GA) on top of MPI using one-sided communication. This work provided portability for GA to new systems which would previously require significant work to provide both an efficient one-sided MPI library and an efficient ARMCI implementation. Our work is similar, but with different goals. In this work, we want to understand the semantics and requirements of a full PGAS language, not just a high-level PGAS-like library. Specifically, a full PGAS language such as CAF enforces a large number of requirements on the MPI implementation, particularly with respect to active messages and event management, which are not required by Global Arrays.

Before this work, the same authors had made efforts to provide a hybrid MPI+UPC programming model [8]. Unified Parallel C (UPC) [29] is a PGAS library intended to simplify data sharing between processes by allowing them to share portions of their address space easily with each other. With the hybrid model between UPC and MPI, Dinan et. al.

demonstrated how applications could take advantage of both the large amount memory which becomes available when using UPC and the portability and existing libraries of MPI. As an example, an application could be written primarily in UPC, but could use an MPI library, such as ScaLAPACK to perform optimized, domain-specific work. However, that work only dealt with the semantic interactions between the two programming models and did not improve the runtime infrastructure of the models. That is, it still relied on each programming model using its own runtime system.

More work was done in this area by researchers at the Ohio State University [14] which was specific to MVAPICH. For this work, the authors unified the runtime systems of MVAPICH and UPC to create a single runtime which supported both programming models. While this work provides similar results to the work presented here, it is not portable across many MPI implementations and therefore requires that MVAPICH be available on the system, something which is not always possible given its reliance on InfiniBand.

Work to unify programming models has existed outside of purely PGAS languages. In 2011, Preissl et. al. [23] examined a communication model that integrated MPI, Fortran 2008 (Coarray Fortran), and OpenMP [21]. This work was specific to the Gyrokinetic Tokamak Simulation code but demonstrated the use of many communication libraries to implement different algorithms within the code. They focused on how PGAS/OpenMP combined could improve over a more traditional MPI+OpenMP algorithm.

Other MPI+X Hybrid Models. The MPI+OpenMP paradigm is just one example of a more classical hybridization of parallel libraries. It has been specifically targeted to improve on-node communication performance by taking advantage of shared address spaces with OpenMP, then switch to MPI when off-node communication is necessary. More recently, accelerators have entered the hybrid programming model space by introducing MPI+CUDA [20] or MPI+OpenCL [15] and allowing an MPI application to offload work to the GPU which can take advantage of the highly parallel architecture available on GPUs.

7. Conclusions and Future Work

In this paper, we demonstrated how MPI-3 can act as a runtime layer for CAF 2.0, thereby providing a basis for interoperability between the MPI and CAF, and reducing the runtime overhead introduced by having independent runtime systems. We demonstrated the new features introduced by MPI-3 and their applicability within a PGAS runtime system. We demonstrated that changing the runtime layer from GASNet to MPI introduces minimal overhead in some cases, and a performance improvement in instances where the CAF operations better map to MPI native operations. Finally, we outlined some improvements which could be adopted by future MPI Standards which would allow work like this to use more optimized MPI operations than what was necessary for this work, such as Active Messages and **MPI_WIN_RFLUSH**.

As future work, we plan to look into several additions to our proposed CAF-MPI framework. One of the short-term goals that we plan to tackle are the performance issues that we identified within the MPI implementation, particularly with respect to the scalability of **MPI_WIN_FLUSH_ALL**. This would improve the performance of operations that rely heavily on CAF events, such as the RandomAccess benchmark.

As a slightly longer term goal, we plan to study the ability of **MPI_WIN_RFLUSH** and its applicability to CAF-MPI. We believe that such an implementation would allow us to move away from **MPI_SEND** and **MPI_RECV** almost entirely within the CAF-MPI runtime, except within active messages. We also plan to use this study as a motivating example to encourage the standardization of such a routine in the next MPI Standard.

Finally, we plan to investigate several applications that can benefit from a hybrid MPI+CAF framework. Two of the first applications we are planning to look at are QMCPACK and GFMC. As described in Section 1, these are existing MPI applications that have a tremendous potential to benefit from using coarrays that would allow their "local" data to be distributed across a small number of processes.

Acknowledgments

This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. We also thank Scott K. Warren and Dung X. Nguyen from Rice Group for porting CGPOP to CAF 2.0 and allowing us to use it for the evaluation of CAF-MPI.

References

[1] A. Petitet and R.C.Whaley and J.Dongara and A.Cleary. HPL - A Portable Implementation of the High-Performance Linpack. `http://bit.ly/JtavrU`, Sept. 2008.

[2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurr. Comput. : Pract. Exper.*, 22(6):685–701, 2010.

[3] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A high performance parallel algorithm for 1-D FFT. In *Proceedings of the 1994 Conference on Supercomputing*, pages 34–40, Los Alamitos, CA, USA, 1994.

[4] D. Bonachea. Active Messages over MPI. URL `http://bit.ly/14VZNOs`.

[5] D. Bonachea. GASNet specification, v1.1. Technical Report UCB/CSD-02-1207, University of California at Berkeley, Berkeley, CA, USA, 2002.

[6] D. Bonachea and J. Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *Int. J. High Perform. Comput. Netw.*, 1(1-3):91–99, Aug. 2004. .

[7] B. Chamberlain, D. Callahan, and H. Zima. Parallel Programmability and the Chapel Language. *Intl. J. of High Performance Computing Applications*, 21(3):291–312, 2007. .

[8] J. Dinan, P. Balaji, E. L. Lusk, P. Sadayappan, and R. Thakur. Hybrid Parallel Programming with MPI and Unified Parallel C. In *7th ACM International Conference on Computing Frontiers*, Bertinoro, Italy, Apr. 2010.

[9] J. Dinan, P. Balaji, J. R. Hammond, S. Krishnamoorthy, and V. Tipparaju. Supporting the Global Arrays PGAS Model Using MPI One-Sided Communication. In *Proc. 26th Intl. Parallel and Distributed Processing Symp. (IPDPS)*, Shanghai, China, May 2012.

[10] R. Gerstenberger, M. Besta, and T. Hoefler. Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided. In *Proceedings of Intl. Conf. for High Perf. Computing, Networking, Storage and Analysis*, SC '13, pages 53:1–53:12, New York, NY, USA, 2013. URL `http://bit.ly/1dCbxe2`.

[11] HPC Challenge Benchmark. HPC challenge benchmark, July 2010. URL `http://icl.cs.utk.edu/hpcc`.

[12] G. Jin, J. Mellor-Crummey, L. Adhianto, W. N. Scherer III, and C. Yang. Implementation and Performance Evaluation of the HPC Challenge Benchmarks in Coarray Fortran 2.0. In *Proceedings of the 2011 IEEE Intl. Parallel & Distributed Processing Symposium*, IPDPS '11, pages 1089–1100, Washington, DC, USA, 2011. .

[13] S. L. Johnsson and R. L. Krawitz. Cooley-Tukey FFT on the Connection Machine. *In: Parallel Computing. Volume*, 18: 1201–1221, 1991.

[14] J. Jose, M. Luo, S. Sur, and D. K. Panda. Unifying UPC and MPI runtimes: experience with MVAPICH. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, page 5. ACM, 2010.

[15] Khronos OpenCL Working Group. The OpenCL Specification, Version 2.0, July 2013. URL `http://bit.ly/15tR61M`.

[16] J. Kim, K. P. Esler, J. McMinis, M. A. Morales, B. K. Clark, L. Shulenburger, and D. M. Ceperley. Quantum energy density: Improved efficiency for quantum Monte Carlo calculations. *Physical Review B*, 88(3), 2013. .

[17] E. Lusk, S. Pieper, and R. Butler. More SCALABILITY, Less PAIN. *SciDAC Review*, (17):30–37, 2010. URL `http://bit.ly/163sZtd`.

[18] J. Mellor-Crummey, L. Adhianto, W. N. Scherer, III, and G. Jin. A new vision for Coarray Fortran. In *Proceedings of the 3rd Conf. on Partitioned Global Address Space Programing Models*, PGAS '09, pages 1–9, New York, NY, USA, 2009. ACM. .

[19] Message Passing Interface Forum. MPI Report 3.0, Sept. 2012. URL `http://bit.ly/UlOwY2`.

[20] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, Mar. 2008. ISSN 1542-7730. .

[21] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 4.0, July 2013. URL `http://bit.ly/13LNHtI`.

[22] S. C. Pieper and R. B. Wiringa. QUANTUM MONTE CARLO CALCULATIONS OF LIGHT NUCLEI. *Annual Review of Nuclear and Particle Science*, 51(1):53–90, 2001. . URL `http://bit.ly/143fd6u`.

[23] 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 Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 78:1–78:11, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0771-0. .

[24] J. Reid. Coarrays in Fortran 2008. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, PGAS '09, pages 4:1–4:1, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-836-0. .

[25] V. A. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 Language Specification, Version 2.2, Sept. 2011. URL `http://bit.ly/1431tse`.

[26] W. N. Scherer, III, L. Adhianto, G. Jin, J. Mellor-Crummey, and C. Yang. Hiding latency in coarray fortran 2.0. In *Proceedings of the 4th Conf. on Partitioned Global Address Space Programming Model*, PGAS '10, pages 14:1–14:9, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0461-0. .

[27] A. Stone, J. Dennis, and M. M. Strout. The CGPOP Miniapp, Version 1.0. Technical Report CS-11-103, Colorado State University, July 2011.

[28] D. Takahashi and Y. Kanada. High-performance radix-2, 3 and 5 parallel 1-D complex FFT algorithms for distributed-memory parallel computers. *J. Supercomput.*, 15(2):207–228, 2000.

[29] UPC Consortium. UPC language specifications v1. 2. Technical report, Lawrence Berkeley National Laboratory, Berkeley, CA, USA, May 2005.

[30] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. *SIGARCH Comput. Archit. News*, 20:256–266, Apr. 1992. ISSN 0163-5964. .

[31] C. Yang. Function shipping in a scalable parallel programming model. Master's thesis, Department of Computer Science, Rice University, Houston, Texas, 2012.

[32] C. Yang, K. Murthy, and J. Mellor-Crummey. Managing asynchronous operations in coarray fortran 2.0. In *Proceedings of the 2013 IEEE International Symposium on Parallel Distributed Processing*, pages 1321–1332, 2013. .

[33] X. Zhao, P. Balaji, W. D. Gropp, and R. S. Thakur. MPI-Interoperable Generalized Active Messages. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, Dec. 2013.

[34] X. Zhao, D. Buntinas, J. A. Zounmevo, J. Dinan, D. Goodell, P. Balaji, R. Thakur, A. Afsahi, and W. Gropp. Toward Asynchronous and MPI-Interoperable Active Messages. In *CCGRID'13*, pages 87–94, 2013.