Hiding Latency in Coarray Fortran 2.0

William N. Scherer III, Laksono Adhianto, Guohua Jin,
John Mellor-Crummey, and Chaoran Yang
Department of Computer Science, Rice University
{scherer, laksono, jin, johnmc, chaoran}@rice.edu

ABSTRACT

In Numrich and Reid's 1998 proposal [17], Coarray Fortran is a simple set of extensions to Fortran 95, principal among which is support for shared data known as *coarrays*. Responding to shortcomings in the Fortran Standards Committee's addition of coarrays to the Fortran 2008 standards, we at Rice envisioned an extensive update which has come to be known as Coarray Fortran 2.0 [15]. In this paper, we chronicle the evolution of Coarray Fortran 2.0 as it gains support for asynchronous point-to-point and collective operations. We outline how these operations are implemented and describe code fragments from several benchmark programs to show we use these operations to hide latency by overlapping communication and computation.

Categories and Subject Descriptors

D.3 [**Programming Languages**]: Language Constructs and Features; D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*

Keywords

Coarray Fortran, Parallel programming, Asynchrony, Asynchronous collectives

1. INTRODUCTION

Over the last few years, as multicore processors have become pervasive, there has been a growing urgency to develop simpler models for efficient parallel programming. Many problems in scientific and technical computing have computational needs that require more than just individual multicore processors. The use of clusters and distributed memory supercomputers to solve problems in this space is pervasive. Today, MPI [24] is the dominant parallel programming model for these systems. MPI has many benefits (portability, scalability, high performance); however, it can be awkward to use for some applications because of its two-sided programming model.

PGAS'10, October 12–15, 2010, New York City, USA.

Over the past decade, there has been considerable interest in developing higher level models for parallel programming. Partitioned global address space (PGAS) languages based on one-sided communication such as UPC [26], Coarray Fortran [17], X10 [22], Chapel [4], Titanium [28] and Fortress [21]. These languages are viewed as perhaps the most promising direction for developing programming models for scalable parallel systems.

Coarray Fortran (CAF) was originally designed by Numrich and Reid [17] as a small set of extensions to Fortran 95 to support parallel programming. They envisioned CAF as a model for SPMD parallel programming based on a static collection of asynchronous process images (known as images for short). In 2005, the Fortran Standards committee began exploring the addition of coarray constructs to the emerging Fortran 2008 standard. Their design closely follows Numrich and Reid's original vision.

Coarray Fortran 2.0 (CAF 2.0) is an extension of Coarray Fortran that provides better expressiveness through features such as process subsets, topologies, and fine-grain synchronization. We outlined the motivation and vision for CAF 2.0 in a 2009 paper [15][1]. In this paper, we augment CAF 2.0 with additional primitives that enable users to overlap communication latency with computation. These features are essential for scalable parallel performance. Specifically, we propose the following additions to CAF 2.0:

- an asynchronous copy operation,
- asynchronous collective operations, and
- language support for shipping computation to remote processors.

While our proposed primitive for asynchronous point-to-point communication is one sided, we propose and justify a two-sided design for asynchronous collective operations. An integral part of our design for asynchronous operations is a uniform mechanism for determining when they have completed.

The remainder of this paper is organized as follows. Section 2 provides a brief summary of our prior design for Coarray Fortran 2.0. Section 3 presents the design of our new

[1]Section 2 provides an overview of CAF 2.0 features.

language asynchronous primitives that enable communication to be overlapped with computation. Section 4 describes how these features are used in some example application kernels. Section 5 provides a sketch of how we have implemented these features in our CAF 2.0 prototype. Section 6 discusses related work and we conclude in Section 7.

2. BACKGROUND

To make this paper self-contained, we briefly summarize the overall structure of Coarray Fortran 2.0 (CAF 2.0) described in earlier work [15]. CAF 2.0 differs significantly from the coarray-based extensions being incorporated into Fortran as part of the 2008 standard. While Fortran 2008 focused on adding the minimum new features to provide some support for parallelism, CAF 2.0 takes a rather different approach, adding a richer set of features to provide what we believe is a *productive* parallel programming model. Below we outline the key abstractions that CAF 2.0 adds to Fortran:

- **process subsets**, known as *teams*, which support coarrays, collective communication, and relative indexing for pairwise operations
- **topologies**, which augment teams with a logical communication structure
- **dynamic allocation/deallocation of coarrays and other shared data**
  - pointers, allocatable coarrays, and dynamic allocation of coarrays with locally-scoped names
  - team-based coarray allocation and deallocation
  - global pointers, which are essential for representing and traversing distributed linked data structures
- **enhanced support for synchronization** for fine control over program execution
  - safe and scalable support for mutual exclusion
  - events, which provide a safe space for point-to-point synchronization
  - split-phase barriers for overlapping communication and computation
- **collective communication**
- **a memory model** that enables one to trade ease of use for performance

Most of these ideas are inspired by features in MPI [24] and Unified Parallel C [5]. These features provide a core for comprehensive support for parallelism in Fortran.

A clear omission in the feature set above is support for multithreading, which is necessary for exploiting the full potential of multicore and manycore architectures. It is missing not because we don't think that it is important; but rather because we have not yet explored the underlying implementation considerations that will be important for producing an appropriate design.

In this paper, we augment this feature set with new support for both pairwise and collective asynchronous operations. This addition provides critical support for hiding communication latency by overlapping it with computation.

```
event_init(event e)
event_notify(event e)
event_wait(event e)
event_trywait(event e, logical success)
event_getid(event e, integer(8) id)
```

Figure 1: CAF 2.0 event API.

2.1 Events

Events were previously described elsewhere [15]; we review their semantics here to provide context for their use to signal the completion of asynchronous operations. We originally conceived events as a mechanism to support point-to-point synchronization. For that purpose, one allocates one or more event scalars or arrays. Figure 1 shows the operations on events with the types of their arguments and return values. One can think of an event as a counting semaphore. One must invoke `event_init` to initialize the count to zero before any other operation. An `event_notify` increments an event's count. Waiting for an event will delay until a process image can atomically decrement an event's count from a value greater than zero. `event_trywait` is a non-blocking operation that will attempt to atomically decrement an event's count from a positive value; it will set the logical variable `success` to indicate whether it succeeded or not. Additionally, every event has associated with it a 64-bit unique ID. The event ID may be retrieved via a call to `event_getid`. Although event IDs are not needed by other operations on events, as we explain in Section 3.4, these IDs make it possible for one to operate on sets of events. Typically, events are used as a one-way synchronization channel between process image pairs; one process notifies the event and the other awaits notifications; however, it is also possible for multiple notifications to accumulate in a single event and have a process wait for them all together, such as waiting on NSEW neighbors in a stencil calculation. As described in the next section, when used in conjunction with asynchronous communication, an event need not be a coarray.

3. ASYNCHRONY IN CAF 2.0

In this section, we describe the design of the language features in Coarray Fortran 2.0 for supporting asynchrony and latency hiding. We start with the predicated asynchronous copy, a one-sided point-to-point non-blocking communication. Then, we discuss support for asynchronous collective operations. We finish the section by discussing function shipping as a mechanism for executing a function on a remote image.

3.1 Models of asynchrony

In Coarray Fortran 2.0, we support two models of synchrony. In the *explicit* model, we post an event to signal that an operation has completed. This unifies the design and eliminates the need for handles which other approaches require. Further, it allows an expert user to write an event-driven processing loop that can receive and react to any event that is notified from among a set of expected notifications, making use of the Coarray Fortran 2.0 eventset primitives described in Section 3.4.

Alternatively, one can use the *implicit* model. Here, rather than signaling an event when the operation is complete, we provide synchronization to programmers via a dynamically scoped, nestable `finish` block. All asynchronous operations within a `finish` scope are guaranteed to be finished before exiting the `finish` block. We note that this concept of finish blocks is from the X10 programming language [22]. Each `finish` block is associated with a specific `team`, which may be omitted to have the the runtime supply `TEAM_DEFAULT`. Note that only implicit operations are guaranteed to be complete at the close of a `finish` block; explicit operations are allowed to *escape*.

3.2 Predicated asynchronous copy

On large-scale parallel systems, hiding communication latency is essential if a program is to achieve high performance. In studies with the HPC Challenge RandomAccess and FFT benchmarks, we quickly recognized the need for asynchronous data copies to overlap communication with computation. In the benchmark codes, we initially identified the need to overlap computation with streaming writes of remote data. In particular, we wanted to issue a non-blocking PUT to remote data and notify the consumer awaiting the data when the PUT is complete; while this communication and synchronization are in flight, we want to continue with local computational work. Later, we identified a similar need to overlap a GET communication with computation and determine when the GET completes. We realized that under many circumstances a compiler might not be able to determine that it is safe to transform a GET or PUT into a non-blocking form and overlap it with computation. In particular, it can be very difficult to determine that a GET or PUT could be overlapped with a procedure call without changing a program's semantics in the case when code is being separately compiled. We realized that programmers know what communication can and should be overlapped with computation, and what CAF needs is a suitable language construct to make it possible to express such asynchronous communication.

As Coarray Fortran was originally conceived by Numrich and Reid, there were no language constructs that enable users to hide communication latency. In addition, their design included implicit memory fences at subroutine boundaries to avoid having a GET or PUT operation, in flight at a procedure call, cause data races with accesses by a callee to the target coarray [17]. A challenge was to create a design for adding asynchronous copies to CAF that enables application programmers to: (1) express that reading or writing of remote coarray data may be overlapped with computation; (2) make it possible to determine when such asynchronous operations are complete; (3) have the language constructs be sufficiently general to allow an application to await completion of pending asynchronous operations in any order; (4) make the completion of asynchronous operations orthogonal to program scopes, i.e. an application need not await completion of asynchronous operations within the same routine in which they are issued; completion may be requested at any point in the future by any routine whatsoever; and (5) provide a syntactic construct that is easy to use.

To satisfy these criteria, we designed the asynchronous copy primitive shown in Figure 2. In our design, `copy_async` is a statement rather than a subroutine in CAF 2.0. Both the source and destination of the copy must be coarray references. The source and destination may be scalar values, whole arrays, or array sections. Either may refer to local or remote coarray data. The events ev_before and ev_after respectively indicate that the source data is ready to copy and that the write of the destination data is complete. In the implicit asynchrony model, ev_after is not specified and an `end finish` statement blocks until the write is complete.

`copy_async`(var_dest, var_src *[, ev_after] [, ev_before]*)

var_dest	=	a coarray reference target
var_src	=	a coarray reference source
ev_after	=	an optional event indicating that the write to dest is complete
ev_before	=	an optional event indicating that the source data is ready

Figure 2: Asynchronous copy statement in CAF 2.0.

The `copy_async` primitive is quite expressive. It can be used for local-to-local, remote-to-local, local-to-remote, or remote-to-remote copies. A local-to-local `copy_async` could be used to hide the latency of a local copy operation by having it executed asynchronously by another core or a DMA engine. A typical use of a remote-to-local copy would be for an asynchronous prefetch. If one specifies the optional ev_before event, the copy will not execute until ev_before is notified; this enables an application developer to specify a predicated prefetch that will not begin execution until the source data is available. A typical use of a local-to-remote `copy_async` is to export to a remote node values that have just been computed, such as depositing boundary layer data into a ghost region on a remote processor. While `copy_async` supports remote-to-remote copies for completeness, we don't have a compelling case for this use at present.

3.3 Asynchronous collective operations

Although they are widely used in parallel applications, the current Fortran 2008 draft standard does not include collective operations; therefore, we previously proposed a set of synchronous collective operations [15]. In this paper, we propose adding two-sided asynchronous collective operations to Coarray Fortran 2.0. When designing asynchronous collectives for a language with one-sided communication, one may ask why not to use a one-sided design. A two-sided design provides us two benefits. First, it enables each processor to have explicit control of how many collective operations may be pending on that processor at a time. Second, each processor has the flexibility to control buffer allocation and specify where incoming results should be placed. Table 1 shows the collective operations we propose adding to Coarray Fortran 2.0. There are two purposes for collective operations: communication and synchronization; we discuss each of these in turn.

3.3.1 Communication

Depending on the types of collected data output, there are three kinds of collective communication. First, in *all-to-one* collective communication like `team_reduce_async` and `team_gather_async`, the result is gathered by one process

Statement	Description
`team_barrier_async([event] [, team])`	barrier synchronization between image processes
`team_broadcast_async(var, root[, event] [, team])`	broadcasts data from an image to all images in a team
`team_gather_async(var_src, var_dest, root[, event] [, team])`	collects individual data from each image in a team at one image
`team_allgather_async(var_src, var_dest[, event] [, team ])`	gathers data from all images and distributes it to all images
`team_reduce_async(var_src, var_dest, root, operator[, event] [, team])`	reduces data; the result is stored to an image of the team
`team_allreduce_async (var_src, var_dest, operator[, event] [, team])`	reduces data; the result is stored to all images of the team
`team_scatter_async(var_src, var_dest, root[, event] [, team])`	distributes individual data from an image to each image in a team
`team_alltoall_async(var_src, var_dest[, event] [, team])`	sends distinct data from each image to every image in a team
`team_sort_async(var_src, var_dest, comparison_fn[, event] [, team])`	sorts arrays of the same size and type within a team

Argument descriptions:

`typedef::var_src`	local source variable	`team::team`	process subset (default team if not specified)
`typedef::var_dest[*]`	target Coarray Fortran variable	`event::event`	event variable (if using explicit asynchrony)
`integer::root`	the rank of the root image		

Table 1: Asynchronous collective operations in Coarray Fortran 2.0.

(named `root`). Second, in *one-to-all* collective communication like `team_broadcast_async` and `team_scatter_async`, the root process sends data to all processes. Third, *all-to-all* communication like `team_allgather_async`, `team_allreduce_async`, and `team_alltoall_async` have all processes participate in sending and collecting data.

Since processes can only know about coarray data on the other processes, there is a chain of data exchange between processes. Some collective communication like `team_reduce_async` for MIN and MAX operations are *replication oblivious*, which means contributions from an image can be processed more than once without changing the result. These collectives can be implemented efficiently with the $\lceil log(P) \rceil$ depth dissemination pattern [8]. Non-replication oblivious communication such as `team_gather_async` and `team_scatter_async` can be implemented with a tree algorithm.

Unlike asynchronous copy, we do not support predication for asynchronous collective communication. While this would be straightforward to implement, we have yet to identify a use case for which it would be desirable.

3.3.2 Synchronization

Asynchronous barriers, also known as split-phased barriers, have been known and studied for a long time now [7]. Their semantics are straightforward: One API call declares that an image has completed all work intended to precede the barrier, and a second API call awaits the work of other images to be complete. Between the two calls, each processor may perform an arbitrary amount of other work that does not need to touch values that might yet still be changed by other processes before the barrier in progress is complete.

Asynchrony in barriers is useful for two main reasons. The classical purpose of split-phased barriers is to tolerate asynchrony between process images. With normal barriers, if an image completes all of its work for one stage of a computation, it is blocked until the other images complete their work as well. Lack of asynchrony tolerance manifests itself as idling at the barrier. Split-phased barriers allow the synchronization latency to be overlapped with work initiated between the initiation of the barrier and its completion, which reduces idling.

Second, it allows the overhead of performing the barrier itself to be overlapped by useful computation. This is important because barrier synchronization can be quite expensive.

Table 1 shows the Coarray Fortran 2.0 runtime interface for the asynchronous barrier. To wait for the barrier, one simply invokes `event_wait` on the event supplied to `team_barrier_async` or closes the surrounding `finish` block.

3.4 Function shipping

Another way to achieve high performance on large-scale parallel systems is to co-locate computation with remote data. In the SPMD programming model, the standard owner-compute approach partitions data and the responsibility for its computation among processes. In practice, however, it may occur that a locally-originated request for computation needs to read and write remote data. In this situation, the process could copy remote data to local memory, update it locally, and write the data back. Alternatively, it can send a closure that specifies a function and any necessary arguments to the remote image for execution in its own locality domain. The first approach suffers from the overhead of two round trips of communication. The second, though it reduces communication overhead, requires the programmer to code for the computation on the remote process, separating it from its logical origin. This could make the program more obscure and harder to maintain.

```
finish (team)
  spawn f(table(i,j)[p], n)[p]
  ...
end finish

event ev
...
spawn(event=ev) f(table(i,j)[p], n)[p]
event_wait(ev)
```

Figure 3: Implicit and explicit model examples of CAF 2.0 function shipping.

Statement	Description
`eventset_init`	Initialize a freshly allocated eventset
`eventset_add`	Add a single event to an eventset
`eventset_addarray`	Add an array of events to an eventset
`eventset_remove`	Remove a single event from an eventset
`eventset_destroy`	Remove all events from an eventset and reclaim resources associated with it

Table 2: Eventset API for manipulating events in the set.

Function shipping in CAF 2.0 solves this dilemma by enabling a program to send computation to a remote process while preserving the consistency of logical and physical code. As shown in Figure 3, a spawn statement causes a subroutine or function to be executed on a target remote node. Semantically, coarray function arguments reference data on the remote node; local data arguments are carried to the remote node. When a function has a non-void return value assigned to a coarray variable on remote process, it is forwarded to that process directly. In our design, any function or subroutine can be executed on a remote node. Return values of spawned functions are only guaranteed to be available after the immediately enclosing `finish` statement ends or when the caller-specified event is signaled.

3.5 Event sets

Because we expect to use events as the basis of our asynchronous point-to-point and collective communication operations, it is important that they be sufficiently flexible and expressive. In particular, we need to support multi-event manipulation functionality akin to the capabilities of the Berkeley sockets `select` statement [25] and the MPI `WAITANY` and `WAITSOME` functions [24] that await completion of asynchronous send or receive operations.

Logically, an eventset is a set of ordered tuples ⟨`event`, `count`⟩, where `count` is the number of times that `event` has been observed to trigger within an eventset API call, and is used for fairness purposes as detailed below. Table 2 details the API for initializing and destroying, and adding and removing elements from an event set. The parameter supplied to `eventset_init` receives a handle to the newly initialized eventset; this handle is then used as an input for the rest of the eventset API statements. It is an error to use an eventset handle after calling `eventset_destroy`: The eventset is no longer initialized.

Statement	Description
`eventset_waitany`	Wait until one event has triggered, checking in priority order
`eventset_waitany_fair`	Wait until one of the events with the least number of recorded triggers has triggered
`eventset_waitall`	Wait until all events have triggered
`eventset_notifyall`	Notify all events

Table 3: Eventset API for manipulating events.

Table 3 presents the portion of the eventset API that deals with notifying or waiting on sets of events. The statements `eventset_waitall` and `eventset_notifyall` wait for and signal, respectively, every event associated with the set. The `eventset_waitany` statement checks each event associated with the list to see if that event has triggered. When it finds one, it increments the trigger count associated with the event and resorts the list of tuples so as to avoid starvation even in the case where a single node is triggered with high frequency. Returning the ID of the triggered event allows the caller to know which event was triggered and react appropriately. `eventset_waitany_fair` behaves similarly to `eventset_waitany`; however, only those events with the fewest trigger counts are considered. This is useful in cases where at each stage of an algorithm, it is necessary to process each of several asynchronous events exactly once. Without this, the same effect could be obtained by maintaining a pair of eventsets, *current* and *next*, and explicitly migrating events from *current* to *next* as they come in and are processed. Once *current* is empty, swap the sets. Here, a small addition to the API dramatically increases programmer convenience and productivity for this case.

4. CAF 2.0 ASYNCHRONY IN PRACTICE

This section presents examples from our CAF 2.0 implementations of several benchmark kernels that highlight how we use the asynchronous primitives we propose.

4.1 HPC challenge benchmarks

The HPC Challenge Benchmark suite [13, 20] is a collection of kernels, each with different sensitivities to computational throughput, memory latency, memory bandwidth, communication latency, and communication bandwidth. The aim of this benchmark suite is to bound the performance of real applications by including representative kernels at various extremes. Because of the diverse characteristics of these kernels, they have been a proving ground for parallel programming models in the form of the HPC Challenge Awards Competition–Class 2 Most Productivity [12].

Here, we present snippets from CAF 2.0 implementations of the HPC Challenge RandomAccess, Fast Fourier Transform (FFT), and High Performance Linpack (HPL) benchmarks.

4.1.1 RandomAccess

The HPC Challenge *RandomAccess* benchmark evaluates the rate at which a parallel system can apply updates to randomly indexed entries in a distributed table of 64-bit words. Each table update consists of generating a random index into the table and performing a read-modify-write operation on the selected table entry. To map this code onto

```
module module_route
  ...
  event, allocatable, dimension(:) :: delivered[*]
  event, allocatable, dimension(:) :: received[*]
  integer(8), allocatable, dimension(:,:,:) :: fwd[*]
 contains
   ...
   subroutine route
   ...
   do i = world_logsize-1, 0, -1
     partner = mod(world_rank+distance+world_size, &
                   world_size)
     ...
     call split(..., fwd(1:,out,i), fwd(0,out,i), ...)
     if (i < world_logsize-1) then
       event_wait(delivered(i+1))
       call split(fwd(1:,in,i+1),..., fwd(1:,out,i), &
                  fwd(0,out,i), ...)
       event_notify(received(i+1)[from])
     endif

     copy_async(fwd(0:outgoing_size,in,i)[partner], &
                fwd(0:outgoing_size,out,i),  &
                delivered(i)[partner], received(i))
     ...
     from = mod(world_rank - distance + world_size, &
                world_size)
   enddo
   end subroutine route
end module module_route
```

Figure 4: Using `copy_async` for routing updates in RandomAccess.

```
module module_table
  integer(8), allocatable :: table(:)[*]
  ...
  subroutine apply_global_updates(buffer, size)
    integer(8) :: buff(:)
    ...
    finish
      do i = 1, size
        pe = ishft(buff(i), -local_table_logsize)
        pe = iand(pe, world_size_minus_one)
        index = iand(buff(i), local_table_size - 1)
        if (pe == world_rank) then
          table(index) = ieor(table(index), buff(i))
        else
          spawn remote_update(table,index,buff(i))[pe]
        endif
        update_index = update_index + 1
      end do
    end finish
  end subroutine apply_global_updates
  ...
  subroutine remote_update(table, index, value)
    integer(8) :: table(:)[*]
    ...
    table(index) = ieor(table(index), value)
  end remote_update
  ...
end module module_table
```

Figure 5: Using function shipping for element-wise verification.

a parallel system, each processor independently performs a subset of the updates to the global table.

On distributed-memory parallel systems that lack hardware support for shared memory, fine-grain operations on remote data are expensive. To develop a high performance implementation of RandomAccess in CAF 2.0, we exploit a property of the Class 2 RandomAccess benchmark specification [11] that allows processors to work on bunches of 1024 updates at a time. Each CAF process image generates a batch of 1024 indices of table locations to be updated and then uses $\log P$ rounds of bulk communication in a hypercube-like pattern to route each update to the appropriate process image co-located with the table index being updated. Finally, each process image locally applies updates to its section of the distributed table.

Figure 4 shows a snippet of our CAF 2.0 implementation of a kernel for routing updates to their proper target process images. The code uses three coarrays: a communication buffer (`fwd`) and two arrays of events (`received` and `delivered`). A `copy_async` operation is used to copy a batch of updates from a slab in the `fwd` buffer on the local process image into a slab of the `fwd` buffer on process image `partner`. The `copy_async` operation is predicated to wait on event `received(i)`, which means that the remote buffer is empty and available: The remote image removed data delivered to it in round i in the last invocation of subroutine `route` and acknowledged its receipt by signaling `received(i)` on the process image that provided the data. After the `copy_async` delivers its data into the remote buffer, it signals the event `delivered(i)[partner]`, informing the recipient that the data has arrived. While the `copy_async` is asynchronously forwarding updates to a communication partner, it can execute the `split` operation in the next iteration of the loop. After that computation finishes, each process image has to wait for the delivery of its incoming batch of updates that was initiated asynchronously by its communication partner in the previous iteration of the loop. Here, `copy_async` provides just the right abstraction to simplify communication pipelining across iterations of the routing loop.

We have also built an alternate kernel that applies fine-grain updates using function shipping. This kernel is used in our RandomAccess implementation to perform element-wise verification. Figure 5 shows a snippet of the fine-grain update kernel. While we perform local updates with standard read-modify-writes, we implement updates that target off-node table indexes with function shipping that tells the compiler that computation should be performed on process `pe`. Since the compiler knows `table` is a coarray, it is evaluated to the correct address on remote process when entering the subroutine. `buffer(i)` is shipped to remote process because it is declared as a regular array, not a coarray. We note that, although the whole body of subroutine `apply_global_updates` is enclosed within a finish block, multiple rounds of finish operation will be executed since the kernel will call the subroutine at each round of update. Explicit call to `advance_async` is not needed because the remote update will happen when processes are waiting at the end of finish block. We note that using one finish operation for the whole kernel computation is probably a better choice for performance; however, we present here its use in multiple rounds to better illustrate the usage of `finish` and `spawn`.

```
module module_fft
  complex, allocatable, dimension (:,2) :: c[*]
  event, allocatable, dimension(:) :: ready[*]
  event, allocatable, dimension(:,:) :: copied[*]
  event, allocatable, dimension(:,:) :: prefetch[*]
  ...
contains
  ...
 subroutine fft(n_local_size, block_size)
   complex(8), target  :: cbuff(0:n_local_size-1)
   ...
   ! remote communication
   do l = loc_comm, levels
     partner = ieor(rank, ishft(1,l-loc_comm))
     event_notify(ready(l)[partner])
     ...
     event_wait(ready(l))
     ...
     ! prefetch blocks of data
     lo = index_adjustment
     do outer = 0, nblocks - 1
       hi = lo + block_size -1
       copy_async(cbuff(lo:hi),c(lo:hi,last)[partner],&
        prefetch(outer,l))
        lo = lo + block_size
     end do

     lo = index_adjustment
     do outer = 0, nblocks - 1
       hi = lo + block_size -1
       ! Get a chunk of data
       event_wait(prefetch(outer,l))

       ! Process it
       ...
       ! Send result to back to partner; note when done
       copy_async(c(lo:hi,current)[partner], &
         cbuff(lo:hi), copied(outer,l))
       lo = lo + block_size
     end do

     ! Wait until we have delivered all data to partner
     do outer = 0, nblocks - 1
       eventy_wait(copied(outer,l))
     end do

     ! Sync with partner
     event_notify(ready_to_proceed(l)[partner])
     event_wait(ready_to_proceed(l))
 end subroutine fft
end module module_fft
```

Figure 6: Using `copy_async` to overlap computation and communication in FFT.

4.1.2 FFT

The HPC Challenge *FFT* (Fast Fourier Transform) benchmark measures the ability of a system to overlap computation and communication while calculating a very large Discrete Fourier Transform (DFT). Our 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 (in-core) FFT computation for as many layers of the DFT calculation as all fit in the memory of a single processor, and remote DFT computation for layers that span the cores of multiple images.

Figure 6 shows a snippet of the code used to perform the cross-core DFT computation. The main processing loop is strip mined to divide the complex data into a series of blocks so that while one block is being processed, the previous one can be communicated in parallel to a partner image. The code uses four coarrays: one (`c`) contains two generations of in-process DFT data, and three are coarrays of events. Of these, `ready_to_copy` indicates that an image has reached the point where it is ready to cooperate with its partner in calculating the DFT for a particular level, `copied` indicates that one round of DFT data has been computed and transferred to to partner image, and `prefetch` indicates that a block of data has been retrieved from the partner images and is ready to be processed.

Examining the `l` loop, we see that each image notifies its partner that it is ready to calculate the "butterfly" at a level by calling `event_notify` on `ready_to_copy`. It then waits for the partner to be ready as well , calling `event_wait` on `ready_to_copy`. This essentially effects a pairwise barrier between the two images. Each image then proceeds to request the data from its partner that it will need to compute this level; `copy_async` is used here for the prefetching. We note that this prefetch is predicated on the data having reached the partner image from the previous round; this takes advantage of the `ev_before` parameter for `copy_async`. A `prefetch` event supplied to `copy_async` is signaled when the data retrieval is complete.

Turning to the `outer` loop, we see that for each block of data, we wait until it has been retrieved by calling `event_wait` on the `prefetch` event. Then, after the data is processed, we use another `copy_async` to send the completed data back to the image's partner for use in a subsequent round. Here, there is no need for a predicate on the `copy_async`: We have already computed the data. Using `copy_async` instead of a simpler put operation here allows transferring the block of data to proceed in parallel with computing the next block.

Although it is not shown here, we note that we could have used an eventset on the prefetched data. The `eventset_waitany_fair` call is perfectly suited for processing each of the prefetch blocks exactly once. This would be useful for the case where blocks arrive out of order.

4.1.3 HPL

The HPC Challenge High Performance Linpack (HPL) benchmark measures a system's ability to deliver high throughput floating point computation when solving a dense system of linear equations. Our CAF 2.0 implementation of HPL implements a sophisticated tiling of the computation, capable of varying both the logical topology used to organize processor cores, as well as the width of data panels used by the processor cores. The matrix of equations is distributed in block-cyclic fashion onto a processor grid organized as one or two dimensions. The block size of the data distribution is determined by the width of data panels preferred by the processor cores. The choice of the block size has significant impact on the node performance of the benchmark. An ideal block size is large enough to achieve high performance while updating the trailing matrices, but small enough to maintain a good load balance for scalability.

To improve performance of benchmark, we use a dual panel structure for factorization. This increases parallelism in factorization and enables us to overlap communication latency with computation by using `team_broadcast_async` to communicate panels. An asynchronous broadcast of a panel is initiated after the panel is factored. A broadcast gets an opportunity to advance each time `advance_async` is invoked. Processors in the column team responsible for the next panel along the critical path start updating and factorizing that panel as soon as they receive the previous panel. The broadcast of each panel is overlapped with the update of the trailing matrix using previously factorized panels and the computation of the next panel factorization. An `event_wait` is performed to finish a `team_broadcast_async`.

5. IMPLEMENTATION

5.1 Asynchronous progress engine

The asynchronous progress engine is a key piece of machinery in the Coarray Fortran 2.0. It implements cooperative multithreading, message-based parallelism in support of asynchrony. There are two main design decisions here: cooperative vs. preemptive multithreading , and message-based (single-threaded) parallelism vs. process-based (separate threads) parallelism. For the first of these decisions, we note that the standard advantages apply here [19]: By not using interrupt-driven multithreading, we avoid reentrancy problems in both the Coarray Fortran runtime as well as the libraries on which it is built; we avoid the overhead of context switching; and we simplify user code as it does not need to worry about these concerns. For the second concern, we share the opinion of Lauer and Needham [14] that both options are equally viable[2]. However, given that the support for multithreading is limited on the BlueGene/P, one of our target platforms, it is simpler to avoid reentrancy altogether and use a message-based system.

With these design decisions in mind, the implementation of the progress engine is fairly straightforward. We maintain a linked list of the asynchronous operations that are currently pending. Associated with each operation is the data structure shown in Figure 7. In it, `status` is an indicator of the current state of the operation, and may be any of `ASYNC_UNINITIALIZED`, `ASYNC_INPROGRESS`, or `ASYNC_COMPLETE`. The `progress_fxn` field is a progress function, invoked on behalf of the operation whenever the progress engine is active. Finally, `async_state_data` is a placeholder for operation-specific data.

5.1.1 Implementing asynchronous operations

To implement an asynchronous operation in the Coarray Fortran 2.0 runtime, one performs the following steps. First, one decomposes the operation into a series of nonblocking steps, and builds a finite state machine that can effect the transitions between these steps appropriately. Next, one couples this progress function with operation-specific data in a call to `async_register`, which inserts the operation into the linked list of pending operations. Finally, after initializing data for the operation, the last step is to toggle the operation's state from `ASYNC_UNINITIALIZED` to `ASYNC_INPROGRESS`.

[2]The same arguments that apply to operating system internals apply equally to the internals of a language runtime.

```
typedef struct async_record_s {
    async_status_t           status;
    struct async_record_s *  next;
    async_progress_fxn *     progress_fxn;
    long                     async_state_data[0];
} async_record_t;

typedef enum {
  ASYNC_UNINITIALIZED, ASYNC_INPROGRESS, ASYNC_COMPLETE
} async_status_t;
```

Figure 7: Asynchronous progress engine: operation data.

5.1.2 Scheduling of asynchronous operations

The asynchronous progress engine is invoked automatically within the context of many Coarray Fortran 2.0 runtime method implementations; however, the user may invoke it directly via the `advance_async` statement. Although not yet implemented, compiler support will eventually automatically insert this statement in translated code.

There are many potential scheduling policies that we could use when the progress engine is invoked. For now we have implemented two of them; further exploration of the scheduling policies is deferred as future work. In the first scheduling policy, we walk the list of pending asynchronous operations and for each, repeatedly call its progress function so long as it indicates progress is being made. Upon reaching the end of the list, we restart from the beginning so long as progress was made for at least one of the pending operations. The second scheduling policy is the same as the first, except the list is not restarted when reaching the end.

Both policies are aggressive in trying to accomplish as much of the asynchronous work as possible when the progress engine is invoked. The decision of which policy to use, therefore, is an assessment of the extent to which asynchronous operations are likely to be part of the critical path for applications.

5.2 Asynchronous collective operations

Our implementation of asynchronous collectives is in its infancy. In this section, we describe the implementation of two asynchronous collectives that we have already implemented. Implementation of remaining primitives in Table 1 will follow.

5.2.1 Broadcast

In the Coarray Fortran 2.0 runtime system, we implemented a split-phase asynchronous broadcast algorithm. The algorithm is binomial tree based and therefore has logarithmic time complexity. All process images in a team make a call to the collective operation, making broadcast a two-sided primitive.

Our implementation of the split-phase asynchronous broadcast supports multithreading. Each thread performs an asynchronous operation on a state machine consisting of multiple states shown in Figure 8. Each processor involving the broadcast operation starts with state `INIT` where initialization of parameters is performed and buffer space for

```
typedef enum {
  INIT, WAIT_RECV, WAIT_SEND, REPLY, WAIT_REPLY, DONE
} abcast_state_t;
```

Figure 8: Data structure states for split-phase asynchronous broadcast.

incoming data is allocated if necessary. Each processor then enters state `WAIT_RECV`. This indicates that the processor may have not received the data and in this case it will return to perform local computation. As soon as a processor receives the data, it initiates an asynchronous operation for each of its successors in the multiple stages of the binomial tree. The processor then enters state `WAIT_SEND`. Meanwhile each of the newly registered asynchronous operations runs on a separate and simpler state machine, waiting for receiving buffer address from a successor and sending the data to the successor. The processor returns to the main state machine at the state `REPLY` after all the asynchronous operations it initiated finish. Each processor entering state `REPLY` sends an acknowledgement to its predecessor and changes to state `WAIT_REPLY` where it makes sure all its successors have received the data it sent out. In the final state `DONE`, processors deallocates local buffers, calls `event_notify`, and marks the operation `ASYNC_COMPLETE`.

5.2.2 Split-phased barriers

Under the hood, the barrier we implement for the Coarray Fortran 2.0 `async_barrier` is the same sense-reversing dissemination barrier that the regular, synchronous barrier uses. Asynchrony is implemented through our asynchronous progress engine. In particular, a progress function for the asynchronous barrier checks each time it is invoked whether the partner flag is set for the current level of the dissemination tree. If so, it moves to the next round and sets the partner's flag for that round. Once the last round of the barrier is complete, the progress function transitions to a `done` state in which it notifies the event associated with the barrier and marks the operation as complete on this image.

5.3 Function shipping

Coarray Fortran 2.0's Function Shipping makes use of the asynchronous progress engine to ship computation to remote image. Every spawned call will be converted into an `async_record_t` struct and added to remote process's asynchronous operation list. The procedure is executed when the progress engine is invoked.

The compiler generates a structure for each spawn call to hold the arguments passed to that call. A pair of functions to marshal and demarshal arguments are also be generated. Arguments to the original function are marshaled into the structure and assigned to the `async_state_data` field in the asynchronous operation data structure. The compiler also creates a wrapper subroutine for the function being spawned, which will first demarshal arguments and call the original function. This wrapper subroutine serves as a progress function to be invoked by the progress engine.

As a special case, when the function being shipped is intrinsic, or when it contains neither loops nor procedure calls, we skip using the asynchronous progress engine. Instead, such functions are executed within an Active Message (AM) handler on the remote process. Return values assigned to local variables are sent back using the AM reply.

5.4 Event sets

Internally, eventsets are maintained as a linked list of event/count tuples sorted by count. When an event is added to an eventset, it is assigned an initial trigger count of zero. It is then inserted in the list just after the last list node with a count of zero; this sets up initial FIFO ordering in the list.

Table 3 presents the portion of the eventset API that deals with notifying or waiting on sets of events. The `eventset_waitany` statement walks the list of events, checking each one to see if it has triggered via a call to `event_trywait`. When it finds one, it increments the trigger count associated with the event and then resorts the list, positioning the triggered node after any others with the same count. `eventset_waitany_fair` behaves similarly to `eventset_waitany`; however, rather than walking the entire list, it checks only those events whose trigger counts match that of the head node. Finally, `eventset_waitall` and `eventset_notifyall` walk the list and wait for or signal each event, respectively.

6. RELATED WORK

Hiding latency is well known to improve application performance. Some research projects have been devoted to transform code from blocking communication into overlapped communication and computation [6, 27].

Asynchrony has been adopted in several modern languages. MPI 2.0 [24] has a range of nonblocking point-to-point communication operations such as `MPI_Isend` but no nonblocking collectives. Introducing asynchrony or nonblocking features to broadcast in MPI has been studied by research groups [2, 10] to tolerate high communication latency. Some nonblocking collective communications in MPI have been introduced [9].

Although X10 does not have collective communication constructs, its `async` statement can be used to emulate asynchronous collective operations [22]. Flat-X10 adds multiplace spawn processes [3]. Habanero's Phaser adds support for asynchronous collectives in X10 [23]. UPC supports primitive collective operations using an asynchronous remote method similar to Coarray Fortran 2.0's function shipping. Support for asynchronous collectives is still under proposal. GASNet implements a range of non-blocking collective operations such as broadcast, reduce, scatter, gather, exchange and allgather with performance comparable to or better than MPI [16]. Aspen supports asynchronous collective communication [1]. OpenCL has an asynchronous copy statement that signals an event upon completion [18]; our asynchronous copy primitive differs in that it is predicated.

7. SUMMARY

As the gap between communication latency and computation speed becomes more and more significant, hiding communication latency is increasingly critical for program performance. We have presented new features in Coarray

Fortran 2.0 to allow programmers to hide latency: asynchronous pair-wise communication with `copy_async`, function shipping for asynchronous remote procedures, and a range of asynchronous two-sided collective operations. We have demonstrated that these new features can be elegantly integrated into three benchmarks: High Performance Linpack (HPL), Fast Fourier Transform (FFT) and RandomAccess kernels from the HPC Challenge benchmark set.

We have prototyped asynchronous features in our runtime. They depend on the asynchronous progress engine, machinery for cooperative multithreading, message-based parallelism. Although their performance is not yet at the level we would like, we are confident that they express the asynchrony primitives that are needed for high performance computation.

Acknowledgments

We acknowledge Fengmei Zhao for her implementation of the bulk of the Coarray Fortran 2.0 translator. We thank Mark Krentel for contributing scripts and a high quality configuration and build system.

Development of Coarray Fortran 2.0 is supported by the Department of Energy's Office of Science under cooperative agreements DE-FC02-07ER25800 and DE-FC02-06ER25754. This research used resources of the National Energy Research Scientific Computing Center at Lawrence Berkeley National Laboratory and the National Center for Computational Sciences (NCCS) at Oak Ridge National Laboratory. Both facilities are supported by the Office of Science of the U.S. Department of Energy. NCCS is supported under Contract No. DE-AC05-00OR22725.

8. REFERENCES

[1] Q. Ali, V. S. Pai, and S. P. Midkiff. Advanced collective communication in aspen. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 83–93, New York, NY, USA, 2008. ACM.

[2] G. Almási, P. Heidelberger, C. J. Archer, X. Martorell, C. C. Erway, J. E. Moreira, B. Steinmacher-Burow, and Y. Zheng. Optimization of MPI collective communication on BlueGene/L systems. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 253–262, New York, NY, USA, 2005. ACM.

[3] G. Bikshandi, J. G. Castanos, S. B. Kodali, V. K. Nandivada, I. Peshansky, V. A. Saraswat, S. Sur, P. Varma, and T. Wen. Efficient, portable implementation of asynchronous multi-place programs. *SIGPLAN Not.*, 44(4):271–282, 2009.

[4] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[5] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, 2003.

[6] A. Faraj and X. Yuan. Automatic generation and tuning of mpi collective communication routines. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 393–402, New York, NY, USA, 2005. ACM.

[7] R. Gupta. The fuzzy barrier: a mechanism for high speed synchronization of processors. In *ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 54–63, New York, NY, USA, 1989. ACM.

[8] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, Feb. 1988.

[9] T. Hoefler, P. Kambadur, R. L. Graham, G. M. Shipman, and A. Lumsdaine. A case for standard non-blocking collective operations. In F. Cappello, T. Hérault, and J. Dongarra, editors, *PVM/MPI*, volume 4757 of *Lecture Notes in Computer Science*, pages 125–134. Springer, 2007.

[10] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and performance analysis of non-blocking collective operations for mpi. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–10, New York, NY, USA, 2007. ACM.

[11] HPC challenge awards: Class 2 specification. `http://www.hpcchallenge.org/class2specs.pdf`, June 2005.

[12] HPC challenge awards competition. `http://www.hpcchallenge.org`, 2009.

[13] HPC challenge benchmark. `http://icl.cs.utk.edu/hpcc`. Last accessed July 13, 2010.

[14] H. C. Lauer and R. M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, 1979.

[15] J. Mellor-Crummey, L. Adhianto, W. N. Scherer, III, and G. Jin. A new vision for coarray fortran. In *PGAS '09: Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, pages 1–9, New York, NY, USA, 2009. ACM.

[16] R. Nishtala. *Automatically tuning collective communication for one-sided programming models*. PhD thesis, University of California at Berkeley, Berkeley, California, 2009.

[17] R. W. Numrich and J. Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[18] OpenCL specification v1.1. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf, June 2010.

[19] J. K. Ousterhout. Why threads are a bad idea (for most purposes). Invited talk, 1996 USENIX Technical Conference, January 1996.

[20] J. J. D. Piotr Luszczek and J. Kepner. Design and implementation of the HPC challenge benchmark suite. *CT Watch Quarterly*, 2(4a), November 2006.

[21] Project Fortress community. `http://projectfortress.sun.com`, 2010.

[22] V. A. Saraswat. X10 language report. Technical report, IBM Research, 2004.

[23] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phaser accumulators: A new reduction construct for dynamic parallelism. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[24] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The complete reference*. MIT Press, Cambridge, MA, 1996.

[25] W. R. Stevens. *UNIX Network Programming: Networking APIs: Sockets and XTI*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[26] The UPC Consortium. UPC language specification. `http://www.gwu.edu/~upc/docs/upc_specs_1.2.pdf`, June 2005.

[27] S. S. Vadhiyar, G. E. Fagg, and J. Dongarra. Automatically tuned collective communications. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 3, Washington, DC, USA, 2000. IEEE Computer Society.

[28] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY 10036, USA, 1998. ACM Press.