Managing Asynchronous Operations in Coarray Fortran 2.0

Chaoran Yang, Karthik Murthy, John Mellor-Crummey
Rice University, Houston, Texas
{chaoran, ksm2, johnmc}@rice.edu

***Abstract*—As the gap between processor speed and network latency continues to increase, avoiding exposed communication latency is critical for high performance on modern supercomputers. One can *hide* communication latency by overlapping it with computation using non-blocking data transfers, or *avoid* exposing communication latency by moving computation to the location of data it manipulates. Coarray Fortran 2.0 (CAF 2.0) — a partitioned global address space language — provides a rich set of asynchronous operations for avoiding exposed latency including asynchronous copies, function shipping, and asynchronous collectives. CAF 2.0 provides `event` variables to manage completion of asynchronous operations that use explicit completion. This paper describes CAF 2.0's `finish` and `cofence` synchronization constructs, which enable one to manage implicit completion of asynchronous operations.**

`finish` ensures global completion of a set of asynchronous operations across the members of a team. Because of CAF 2.0's SPMD model, its semantics and implementation of `finish` differ significantly from those of finish in X10 and Habanero-C. `cofence` controls local data completion of implicitly-synchronized asynchronous operations. Together these constructs provide the ability to tune a program's performance by exploiting the difference between *local data completion*, *local operation completion*, and *global completion* of asynchronous operations, while hiding network latency. We explore subtle interactions between `cofence`, `finish`, events, asynchronous copies and collectives, and function shipping. We justify their presence in a relaxed memory model for CAF 2.0. We demonstrate the utility of these constructs in the context of two benchmarks: Unbalanced Tree Search (UTS), and HPC Challenge RandomAccess.

We achieve 74%–77% parallel efficiency for 4K–32K cores for UTS using the T1WL spec, which demonstrates scalable performance using our synchronization constructs. Our `cofence` micro-benchmark shows that for a producer-consumer scenario, using local data completion rather than local operation completion yields superior performance.

I. Introduction

Writing parallel programs using message passing is considered harder than writing parallel programs using shared-memory. Partitioned Global Address Space (PGAS) languages can simplify programming for scalable distributed-memory systems by providing a shared-memory abstraction. They also enable application developers to exploit locality for performance by giving them control of data layout and work assignment. The current Fortran standard, Fortran 2008 [1], includes a set of PGAS extensions to support SPMD parallel programming. These extensions closely follow the vision of Coarray Fortran proposed by Numrich and Reid in 1998 [2]. PGAS constructs in Fortran 2008 target a small set of applications, such as dense matrix computations. Unfortunately, Fortran 2008 lacks features such as image subsets and collective communication primitives, which limit the expressiveness of the language [3].

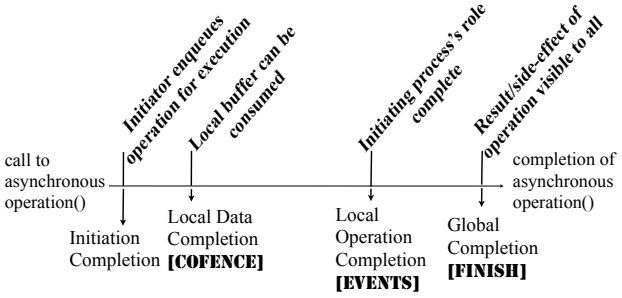

Fig. 1. Execution timeline of an asynchronous operation indicating the different points of completion. *Finish*, *cofence*, and *events* are different synchronization constructs which help achieve different points of completion.

To address the shortcomings of Fortran 2008, we have been exploring the design and implementation of a richer set of PGAS extensions to Fortran 95 that we call Coarray Fortran 2.0 (CAF 2.0) [4]. CAF 2.0 is designed to be a PGAS model that balances ease of use for application developers and control over performance-critical decisions for scalable performance. CAF 2.0 provides a rich set of primitives to control data distributions, communication, and synchronization. Section II briefly outlines the range of features in CAF 2.0.

Asynchronous operations are a key component of programming models for large-scale parallel systems. They are necessary for hiding the latency of synchronization and data movement. Ravichandran et al. [5] describe a strategy for global load balancing in an application with dynamic work. Although they indicate that they would have liked to use a PGAS model for their implementation, they felt that the lack of a latency avoidance mechanism, such as function shipping, made PGAS models unsuitable. Instead, they handcrafted an implementation of a function-shipping like strategy, using MPI and a dedicated thread to manipulate remote task queues. Our design for CAF 2.0 addresses the need for latency hiding at the language level by including three kinds of asynchronous operations: asynchronous copies, asynchronous collectives, and function shipping.

An asynchronous copy is a one-sided operation that may be initiated by any process image (the initiator) to move data from any process image (the source) to any process image (the destination). Asynchronous collective operations enable

each member of a *team* to overlap group coordination with other work. Function shipping enables a programmer to move computation (or control) to a remote process image to avoid exposing latency.

Asynchronous operations are useless without completion guarantees. Figure 1 shows four forms of completion worth considering for an asynchronous operation: *initiation completion*, *local data completion*, *local operation completion*, and *global completion*. For an asynchronous operation A initiated by image i, we define these four completion points as follows:

- A is *initiation complete* on image i when A has been queued up for execution;
- A is *local data complete* on image i when any inputs of A on i may be overwritten, and any outputs of A on i may be read.
- A is *local operation complete* when any pair-wise communication for A involving image i is complete, at which point data on i touched by A can be read or modified.
- A is *globally complete* when A is local operation complete with respect to all participating images.

For an asynchronous copy from p to q initiated by p, local data completion and local operation completion are equivalent. An asynchronous broadcast is local data complete on image p when data is ready to be consumed on p; it is local operation complete on p whenever its role in the broadcast (e.g., distributing to its children in the broadcast tree) is complete. An asynchronous broadcast is, however, only globally complete when the source data for the broadcast has been delivered to all participating images.

CAF 2.0 provides programmers with the ability to manage these four forms of completion for asynchronous operations. In CAF 2.0, the return of an asynchronous operation call guarantees only *initiation completion*. A programmer can be notified about *local operation completion* of an asynchronous operation by explicitly specifying an `event` variable. This paper describes two CAF 2.0 synchronization constructs: `finish` and `cofence`. `finish` is a block structured construct inspired by the `async` and `finish` constructs in X10 [6] but adapted for an SPMD model; a `finish` block ensures *global completion* of a set of asynchronous operations initiated within. `cofence` is inspired by the SPARC V9 [7] `MEMBAR` instruction; it controls *local data completion* of implicitly synchronized asynchronous operations. `finish`, `cofence`, and `event`s enable a programmer to precisely specify the least restricted form of completion needed to guarantee correctness and, at the same time, provide maximum opportunity for overlapping communication and computation to improve performance. Our experiments show these constructs enable one to manage asynchrony effectively on scalable systems.

In the next section, we summarize the key features of Coarray Fortran 2.0 to provide context for the work described in this paper. Section III describes how we manage asynchrony in CAF 2.0 using `finish` and `cofence`. Section IV describes experiments that show the utility of `finish` and `cofence`. Section V discusses related work on synchronization. Section VI summarizes our conclusions.

II. Background

Although features of CAF 2.0 are described elsewhere [8], [9], we briefly review key features here to make this paper self-contained and provide a context for discussing `finish` and `cofence`, which are the focus of this paper.

A. Team

A team is a first-class entity that represents a process subset in CAF 2.0. Teams in CAF 2.0 serve three purposes: a) the set of images in a team serves as a domain onto which shared distributed data objects, known as coarrays, may be allocated; b) a team provides a name space within which process images can be indexed by their relative rank; c) a team provides an isolated domain for its members to communicate and synchronize collectively.

All process images initially belong to `team_world`; new teams are created by invoking `team_split`.

B. Event

Events in CAF 2.0 serve as a mechanism for managing completion of asynchronous operations and pair-wise coordination between process images. Like other variables, events to be accessed remotely are declared as coarrays. Local events (those not in a coarray) are only useful for managing completion of locally-initiated operations.

Events can be passed as optional parameters to asynchronous operations, which use them to initiate or signal completion of various forms. The occurrence of an event can also be signaled explicitly using `event_notify`. An `event_wait` operation blocks execution of the current thread until an event has been posted.

C. Asynchronous Operations

1) Asynchronous Copy: CAF 2.0 provides a predicated asynchronous data copy. The `copy_async` API, as shown below, copies `srcA` on image `p2` to `destA` on image `p1`.

```
copy_async(destA[p1],srcA[p2],preE,srcE,destE)
```

An asynchronous copy may proceed after its (optional) predicate event `preE` has been posted. Notification of event `srcE` indicates that read of data `srcA` is complete; `srcA` can be overwritten on `p2`. Notification of event `destE` indicates that the data has been delivered to `destA` on `p1`. `p1` and `p2` may either be local or remote images. `preE`, `srcE`, and `destE` each may either be a local event or an event located on a remote image.

2) Function shipping: Function shipping in CAF 2.0 enables programmers to move computations, encapsulated in functions, to other process images for execution. A programmer can use function shipping to move computation to the location of remote data it manipulates, to avoid exposing communication latency, or to share dynamically-generated work. Figure 2 shows a PGAS work stealing algorithm by Dinan et al. [10]. Without function shipping, each steal attempt in the algorithm requires five round trips of communication. Coding a steal attempt into a shipped function, as shown in

```
1  while ¬have_work() ∧ ¬terminated do
2    v ← select_victim()
3    m ← get(v.metadata)
4    if work_available(m) then
5      lock(v)
6      m ← get(v.metadata)
7      if work_available(m) then
8        w ← reserve_work(m)
9        m ← m − w
10       put(m, v.metadata)
11       queue ← get(w, v.queue)
12     end if
13     unlock(v)
14   end if
15 end while
```

Fig. 2. A PGAS work stealing algorithm that takes 5 round trips for each steal attempt; remote operations are shown in **bold** font. [10]

```
1  while ¬have_work() ∧ ¬terminated
2    spawn steal_work(select_victim(), me)
3  function steal_work(v, d)
4    m ← v.metadata
5    if work_available(m) then
6      lock(v)
7      m ← v.metadata
8      if work_available(m) then
9        w ← reserve_work(m)
10       v.metadata ← m − w
11       spawn provide_work(v.queue, w, d)
12     end if
13     unlock(v)
14   end if
15 end function
```

Fig. 3. Uses function shipping to rewrite the algorithm shown on the left. Each steal attempt uses 2 round trips.

Figure 3, localizes remote `get`, `put`, `lock` and `unlock` operations. Thus, the function shipping version reduces the communication cost of a steal attempt to only two round trips (two `spawn` operations). Several other parallel languages support mechanisms for function shipping, including X10 [6] (`async at`) and Chapel [11] (`begin at`).

CAF 2.0 function shipping is asynchronous. Like other asynchronous operations in CAF 2.0, its completion can be explicitly coordinated using optional event variables. The following line of code shows how to ship a function foo to image p for execution.

```
spawn(e) foo(A[p], B(i))[p]
```

In the above example, event e will be notified when execution of the spawned function completes.

Arguments passed to a shipped function are treated differently based on their types. An array or scalar argument passed to a shipped function is copied and transferred to the destination image on which the shipped function will execute. A coarray section is passed to a shipped function by reference. In the example above, `A` is a coarray; a reference to coarray `A` is passed to the shipped function `foo`. Thus, `foo` can manipulate the section of coarray `A` local to p. `B` is a local array; its ith value is copied to p.

3) Asynchronous Collectives: CAF 2.0 asynchronous collectives enable one to overlap collective communication with computation. Our vision for asynchronous collectives in CAF 2.0 includes support for `alltoall`, `barrier`, `broadcast`, `gather`, `reduce`, `scatter`, `scan`, and `sort`. The following line of code shows an example of an asynchronous broadcast.

```
team_broadcast_async(A(:),root,myteam,srcE,localE)
```

Like asynchronous copy, asynchronous collectives have optional `event` parameters. In the example above, `srcE` indicates local data completion; `localE` indicates local operation completion.

III. Managing asynchrony in CAF 2.0

With the spectrum of parallel programmers ranging from domain scientists mapping applications onto clusters to computer scientists trying to tune applications for maximal performance, we believe it is necessary to provide expressive language-level mechanisms for coordinating communication that suit developers with varying needs and levels of expertise. In CAF 2.0, we provide two models for coordinating asynchrony: *implicit completion* and *explicit completion*.

Using *implicit completion*, the casual programmer relinquishes the ability to manage completion of an individual asynchronous operation. When initiated with implicit completion (i.e., without `event` variables), the completion of an asynchronous operation is not known until the next `cofence` or the end of an enclosing `finish` block. Using *explicit completion*, advanced programmers can use `event` variables to acquire completion of an individual asynchronous operation until the latest possible moment. The former is intended as the default synchronization model and the latter is intended for tuning the performance of particularly costly communications.

For performance, CAF 2.0 uses a relaxed memory model [12]. Relaxed memory models do not guarantee a strict order between operations on data and synchronization objects unless these operations are ordered by synchronization. The relaxed model applies to all asynchronous operations, coarray read/write, and event notify/wait. Other operations that have local side effects follow a *data-race-free-0* model [13]; in short, operations that access process-local variables are free to be reordered unless they are (a) `SYNC` statements (as defined in the Fortran 2008 standard), and/or (b) violate data or control dependences. The guiding principles for the CAF 2.0 relaxed memory model are:

a) Processor consistency: This principle dictates that all process images see writes of a single process image in the same order. The writes of different process images may be seen in different orders by different process images [13].

b) Different forms of consistency at synchronization points: Different synchronization operations guarantee differ-

Operation type	Image's role	Local data completion (`cofence`)	Local operation completion (`event_wait`)	Global completion of operations with implicit synchronization (`finish`)
Asynchronous broadcast*	Root	Data buffer can be safely modified	All pair-wise communication involving this image is complete	Data is ready on every participating image
	Participant	Data may be read	All pair-wise communication involving this image is complete; Data can be safely modified	Data is ready on every participating image
Asynchronous copy	Reading from a local buffer	Source buffer may be written	Destination buffer may be read	
	Writing to a local buffer	Destination buffer may be read		
Spawn	Initiator	Arguments to the spawn operation are evaluated; local data passed as arguments may be written	Spawn is complete on the target image	Any asynchronous operation with implicit completion initiated by this spawn is globally complete

Fig. 4. Asynchronous operations and different stages of completion. *We discuss asynchronous broadcast as a representative example for asynchronous collectives.

ent levels of completion. Below, we outline the semantics associated with `finish`, `event_notify`, `event_wait`, and `cofence`.

`finish` guarantees global completion of implicitly completed asynchronous operations. We discuss the semantics in detail in Section III-A.

`cofence` guarantees local data completion of implicitly completed asynchronous operations. We discuss the semantics in detail in Section III-B.

When an `event` is notified as a side effect of an asynchronous operation, it guarantees local completion of that operation. For example, consider the code snippet below depicting a broadcast by process image `P0`, and a subsequent `event_wait` on event `doneE`.

```
team_broadcast_async(A(:), P0, myteam, srcE, doneE)
call event_wait(doneE)
```

When `doneE` is notified on `P0`, the role played by `P0` in the broadcast is complete, i.e., all stages of the broadcast in which `P0` participates are complete. However, notification of `doneE` does not imply that the broadcast is complete on other process images.

`event_wait` and `event_notify` obey acquire and release semantics. `event_notify` has release semantics. This principle allows the execution of operations that follow an `event_notify` in the program to begin before it, provided that data and control dependence constraints are not violated. `event_wait` has acquire semantics. This principle allows a process image to wait for an event earlier than it occurs in program order, provided that it does not violate data and control dependences. A summary of the asynchronous operations, and the different points of completion for each of the operations is described in Figure 4.

In the following sections, we describe the `finish` and `cofence` constructs in more detail.

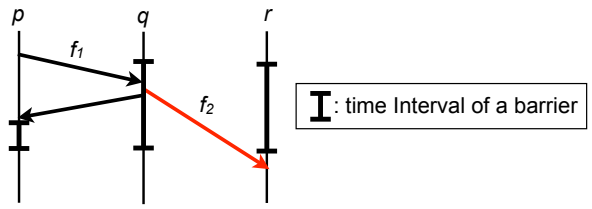

Fig. 5. A case where termination detection using a barrier fails: shipped function f_2 lands on process image r after r exits the barrier.

```
1  finish (ateam)
2    ...
3  end finish
```

Fig. 6. `finish` construct in CAF 2.0.

A. Finish

Adding function shipping into an SPMD language such as CAF 2.0 enriches the set of algorithms that can be expressed efficiently in SPMD fashion. Because a spawned function can transitively spawn more functions, efficiently detecting global completion is difficult. CAF 2.0, like UPC, follows the SPMD model, in which computation begins in multiple places. One might think that in an SPMD model, termination detection can be achieved simply by having each process image wait for completion of all asynchronous operations that it initiated within a `finish` block, and then perform a barrier across the associated team to complete the `finish` block. However, this method can fail when shipped functions are spawned transitively. Consider the case depicted in Figure 5. Image p ships a function f_1 to q, which ships another function f_2 to r. p enters in a barrier after it is notified that f_1 completes. Because image r may receive function f_2 after r notices that p reached the barrier, r may exit from the barrier before f_2 completes. Thus, a barrier can't detect termination correctly. This motivated us to develop a new way to detect global completion of asynchronous operations in CAF 2.0, the result of which is the `finish` construct.

In the following sections, we first describe the semantics of `finish`. Then we present the algorithm that `finish` uses to detect global completion. In the final section, we prove that the algorithm to detect global completion has critical path length $O((L+1)\log p)$, where L is the length of the longest spawning chain of shipped functions, p is the size of the team associated with the `finish`.

1) Semantics: As shown in Figure 6, CAF 2.0 `finish` is a block-structured construct, demarcated by `finish` and `end finish` statements. A `finish` block involves a collective operation over a team. Every image within the associated team needs to create a `finish` block that matches those of its teammates. When a process image encounters an `end finish` statement, it blocks until all of its teammates are done with the current `finish` block. `finish` blocks can be nested and the team associated with a nested block can differ from the team of its parent `finish` block.

When using `finish` to control completion of an asynchronous collective operation, the team associated with the asynchronous collective operation must be the same team or a team that is a subset of the team of the enclosing `finish` block.

The semantics of CAF 2.0 `finish` differs from those of `finish` in X10. X10's `finish` construct guarantees completion of asynchronous activities rooted at a `finish` block executed at a single place. CAF 2.0 `finish` is a collective operation; it provides barrier-like synchronization that guarantees completion of asynchronous operations spawned within a `finish` block entered by multiple process images independently. `finish` in CAF 2.0 is similar to clocks in X10 [14], or phasers in Habanero Java [15] with respect to synchronizing asynchronous activities across process images. To our knowledge, there are no scalable implementations of clocks and phasers for distributed-memory systems at present.

GASNet [16], the communication system used by CAF 2.0 runtime system and Berkeley UPC, provides *access regions* to implicitly synchronize multiple non-blocking operations initiated within. CAF 2.0 `finish` semantics differ from those of GASNet access regions. An access region in GASNet guarantees completion of non-blocking `get` and `put` initiated by a thread, whereas `finish` in CAF 2.0 ensures global completion of asynchronous operations, including asynchronous collectives initiated by members of a team. Unlike `finish` blocks, GASNet access regions cannot be nested.

2) Algorithm for termination detection: In the 80's, many algorithms were proposed to detect termination in distributed systems. However, limitations of these algorithms make them unsuitable for today's supercomputers. These limitations include requiring consistent global time across the system [17, §6.1], a synchronous communication model [18], message channels that obey the First-In-First-Out (FIFO) rule [19], and storage requirements that grow at least linearly with the number of processes [20] [17, §6.2, §7]. The current implementation of `finish` in X10 suffers from a different problem: it is centralized, which is an obstacle to scaling; we discuss this further in Section V.

For CAF 2.0, we develop a new termination detection algorithm for SPMD programs that targets modern, large-scale parallel systems. Figure 7 shows the pseudo-code of our algorithm. It detects termination by repeatedly performing sum reductions (line 8) across all members of a team to compute the global difference between the number of sent messages and received messages.[1] Global termination occurs when a sum reduction yields zero for the difference. We use a synchronous team-based `allreduce` to perform the sum reductions.

When computing termination, we consider the interval in time between when a `finish` scope starts and ends as divided into a series of epochs. These epochs are numbered consecutively starting from zero; we consider odd-numbered and even-numbered epochs separately. As input, our algorithm

[1]For simplicity, we use term *message* to denote a spawned function or communication associated with an asynchronous copy or asynchronous collective.

```
1  function detect_termination(finish f) {
2    epoch e = f.even_epoch
3    while (work_left) {
4      wait_until(e.sent == e.delivered
5        && e.completed == e.received)
6      if (f.present_epoch != f.odd_epoch)
7        next_epoch(f)
8      allreduce(SUM, e.sent-e.completed,
9        work_left, f.team)
10     e = next_epoch(f)  }}
11
12 function next_epoch(finish f) {
13   if (f.present_epoch != f.odd_epoch) {
14     f.present_epoch = f.odd_epoch
15   } else {
16     epoch ep = f.even_epoch,
17           op = f.old_epoch
18     ep.sent = ep.sent+op.sent
19     ep.delivered = ep.delivered+op.delivered
20     ep.received = ep.received+op.received
21     ep.completed = ep.completed+op.completed
22     op.sent = 0
23     op.delivered = 0
24     op.received = 0
25     op.completed = 0
26     f.present_epoch = ep }
27   return f.present_epoch }
28
29 function message_handler(..., fromOddEpoch)
30 { ...
31   if (fromOddEpoch) {
32     f.present_epoch = odd_epoch
33     ... }
```

Fig. 7. The termination detection algorithm in `finish`.

takes a `finish` structure containing state information for two epochs, known as `even_epoch` and `odd_epoch`; an epoch pointer, `present_epoch`, which will always point to `even_epoch` or `odd_epoch`; and a team variable, which represents the team executing the finish scope. An `epoch` structure contains four integers that count the number of messages a process has sent, completed, and received locally, as well as the number of functions it has delivered remotely.

Initially, `present_epoch` points to `even_epoch`. A process proceeds from an even epoch into an odd epoch when a) it enters an `allreduce` (line 7), or b) when it receives a message from a process in an odd epoch (line 32). A process proceeds from an odd epoch into an even epoch when it exits an `allreduce` (line 10).

We enforce a condition that must be satisfied before initiating a new round of termination detection: a process waits until all messages it sent were received and all spawned functions received completed execution before the process performs a new sum reduction (line 4). As proven in the next section, this prerequisite reduces the number of rounds of sum reductions and detects termination at the earliest possible time.

Because a message sent from an odd epoch will cause the receiver process to proceed into an odd epoch, the sending, reception, delivery and completion of the message

are all counted by counters in the odd epoch. Therefore, messages sent or received by an image once it has initiated an `allreduce` are not counted in that round of sum reduction; they will be included in the next round. This property ensures the time cut constructed using an `allreduce` is always consistent and guarantees our algorithm is correct. For lack of space, a formal proof of correctness is omitted here; it can be found in Yang's thesis [21].

3) Communication bound: This section proves that the global termination detection algorithm described above has a critical path of length $O((L+1)\log(p))$ in the worst case, where p is the size of the team associated with a `finish` block and L is the length of longest chain of shipped functions in the `finish` block. We define the length of a chain of shipped functions inductively. A spawned function that spawns no functions has chain length 1. Otherwise, a spawned function has a chain length of one more than the maximum chain length of any function it spawns. For example, a spawn chain consisting of f_1, which spawns f_2, which spawns f_3 has a chain length of three. We first prove that our algorithm uses at most $L+1$ rounds of sum reduction.

Theorem 1. *The algorithm listed in Figure 7 uses at most $L+1$ rounds of sum reduction to detect termination, where L is the length of the longest spawning chain of shipped functions.*

Proof: We prove this theorem by induction.

a) Base case: $L = 0$ means no functions are shipped within a `finish` scope. In this case, the number of messages sent, received, delivered, and completed are $(0, 0, 0, 0)$ on each image. These counters will sum reduce to zero after they are collected by an `allreduce` operation. Thus, one `allreduce` detects termination correctly.

b) Inductive hypothesis: Assume that when the longest chain of spawned functions in a `finish` scope is L, at most $L+1$ rounds of reductions are needed to detect termination.

c) Induction step: We must show that at most $L+2$ rounds of reductions are needed to detect termination when the longest chain of spawned functions is $L+1$.

In a `finish` scope where the longest chain of spawned functions has length $L+1$, at the time that the $L+1$ round of reduction completes, the longest chain of shipped functions that has been spawned is $\geq L$ (specifically, L or $L+1$); otherwise, the system would have terminated in an earlier round by the inductive hypothesis. If it is $L+1$, termination is detected successfully with $L+1$ rounds of reduction. If it is L, we know that all shipped functions have completed, except the shipped function that is the last function in the chain of length $L+1$. We denote the shipped function that has yet to complete as m. Let $t_e(k, i)$ denote the k^{th} time when image i switches from an even epoch to the succeeding odd epoch. Let $t_s(m)$, $t_r(m)$, and $t_c(m)$, respectively, denote the time when m is sent, received, and completed. Let i be the image that sends m, and j be the image that completes m. We know that $t_c(m) > t_e(L+1, j)$. In our algorithm, because an image waits for all messages it sends to land before it enters `allreduce`, we have $t_r(m) < t_e(L+1, i)$. Then, because an image completes all messages it receives before it starts another round of reduction, we have $t_c(m) < t_e(L+2, j)$. Because m is the only message left after $L+1$ reductions, the $L+2$ reduction detects termination.

Since both the base case and the induction step have been proven, Theorem 1 holds by induction. ■

Theorem 1, combined with the fact that an all-to-all reduction can be accomplished in $O(\log p)$ rounds using one pass through a reduction tree and one pass through a broadcast tree, shows that our termination detection algorithm has a critical path length of at most $O((L+1)\log p)$, when the longest chain of spawned functions is L. This theorem also holds when nested `finish` blocks exists [21]. In cases where no nested function shipping is involved, $L = 1$; our `finish` algorithm has a critical path length of $O(\log p)$.

B. Cofence

The `cofence` statement enables a programmer to demand local data completion of implicitly-synchronized asynchronous operations. An asynchronous operation A is local data complete on the initiating image i, when 1) any inputs of A on i may be overwritten and 2) any outputs of A on i may be consumed.

The `cofence` API is as follows:

```
cofence(DOWNWARD=READ/WRITE/ANY,
        UPWARD=READ/WRITE/ANY)
```

The construct takes two optional arguments. The first argument specifies which class of asynchronous operations (i.e., `write/read`) with implicit synchronization appearing in statement instances encountered before a `cofence` statement may defer completion until after the `cofence` statement instance.[2] The second argument specifies which class of asynchronous operations with implicit completion, whose statement instances are encountered after a `cofence` statement instance, may be initiated before the `cofence` statement instance completes. Depending upon the argument values passed, a `cofence` allows reads, writes, or both to pass across in the specified direction. If an asynchronous operation both reads and writes local data, then a `cofence` that allows either a read or write to pass across may not have any practical effect if the unconstrained action (e.g., a read) must occur before a constrained action (e.g., a write). The optional arguments enable one to relax requirements for implicit completion for performance tuning, without requiring a programmer to manage completion of each asynchronous operation explicitly.

Currently, GASNet, the communication layer used by CAF 2.0, ensures that a non-blocking communication operation, involving a `read` from a local buffer, is local data complete before the call returns. This completion semantics makes it difficult to perform computation between initiation

[2]Here, we use the term statement instances rather than statements to accommodate the case in which a `cofence` and asynchronous communication with implicit completion are present in a loop.

```
1  copy_async(inbuf(i)[succ], outbuf(i))
2  ...
3  cofence() ! no copy_async may cross
4  ...
5  copy_async(inbuf(i+1), outbuf(i+1)[succ])
6  copy_async(inbuf(i+2)[succ+2], outbuf(i+2))
7  ...
8  cofence(DOWNWARD=WRITE) !copy_async at line 5
9                          !may complete below
```

Fig. 8. Cofence and asynchronous copy

```
1  if(my_image_rank .eq. p) then
2    ...
3    call broadcast_async(buf, p)
4    ...
5    cofence(WRITE, WRITE)
6    !buf can be safely overwritten
7    buf = ...
8  else
9    call broadcast_async(buf, p)
10   cofence()
11   !read buf
12 endif
```

Fig. 9. Cofence and asynchronous broadcast. On a participating process image, say q, the cofence captures the arrival of broadcast data.

```
1  subroutine foo()
2    copy_async(...)
3    cofence()
4  end subroutine
5  program
6    copy_async(...)
7    finish
8      spawn foo()[random_process_image]
9      cofence()
10   end finish
11 end program
```

Fig. 10. Cofence inside shipped functions

completion and local data completion. To circumvent this limitation, one can offload asynchronous communication calls to a communication thread so that the main thread can perform useful work as soon as it initiates a communication. This strategy of using an extra communication thread might not be advantageous on systems with few threads per node, e.g., Blue Gene/P. However, emerging systems such as Blue Gene/Q and Intel MIC have a substantial number of hardware threads per node. On such platforms, dedicating a small number of threads on each node for communication could be beneficial. In the following sections, we detail the interactions between asynchronous operations in CAF 2.0 and `cofence`.

1) Asynchronous copy: The semantics of local data completion w.r.t. an asynchronous copy operation are:

- On the initiating process, if the source buffer of the copy is local, local data completion of the copy operation ensures that the source buffer can be safely modified.
- On the initiating process, if the destination buffer of the copy is local, local data completion of the copy operation ensures that the destination buffer is ready to be consumed.

Figure 8 shows an example in which both `cofence` and `copy_async` constructs are used. The asynchronous copy operation at line 1 copies `outbuf(i)` on the initiating image to coarray `inbuf` on image `succ`. The operation is local data complete when `outbuf(i)` can be modified on the initiating image without potentially affecting the value written to `inbuf(i)`. The point to note is that this completion does not include whether the copy operation is complete on `succ`, i.e., whether `inbuf(i)` is available for consumption. The `cofence` at line 8, however, allows the `copy_async` operation at line 5 to pass downwards, namely, it may complete after the `cofence`. At the same time, the `cofence` at line 8 ensures local data completion of the `copy_async` operation at line 6.

2) Asynchronous collectives: Consider the code fragment in Figure 9. This example shows the use of an asynchronous broadcast operation along with `cofence`. The `cofence` on line 5 guarantees that process image p can reuse buffer `buf` without affecting the correctness of the broadcast operation. However, the `cofence` at line 5 allows other asynchronous operations writing other local data to move across unconstrained. In this snippet of code, the root image of the broadcast (p) is using the time between local data completion and local operation completion to ready the buffer for the next round of broadcast. On other process images involved in the broadcast, e.g., process image q, the cofence on line 10 captures the arrival of broadcast data on q.

3) Shipped functions: When a `cofence` is used inside a shipped function, it should only capture the completion of implicit asynchronous operations launched by the shipped function (i.e., with dynamic scoping). Accordingly, the cofence on line 3 in Figure 10 does not guarantee local data completion of the asynchronous operation in line 6, but only of the asynchronous operation in line 2. A cofence on line 9, present after the spawn, captures the completion of argument evaluation (if any) for the spawn function, and does not provide any guarantee w.r.t. execution of the spawned function. In other words, if any local buffers are passed as arguments to the spawn then they can be safely modified only after the `cofence`.

4) Events: Events contain an implicit `cofence`. In this section, we describe the semantics of two event mechanisms: event wait and event notification. The semantics are as follows:

a) An event_notify acts as a release operation: An `event_notify` by a process image p indicates that p has completed a certain set of updates to shared data of interest to other process image(s). Since, in general, its not possible to identify the updates of interest to other process image(s), e.g., q, who might be waiting on the event, the `event_notify` should prevent operations from moving downwards. On the other hand, p does not give any guarantee to q when it executes operations following the

event_notify. Thus, the event_notify can be porous to operations that appear afterward, i.e., these operations can be initiated before the event_notify. Hence, we implement the event_notify as a release operation [22]. We use epochs (inspired by MPI's access epochs [23]) to allow for such an implementation.

b) An event_wait acts as an acquire operation: An event_wait indicates that a process image q cannot execute any operations after the event_wait until a notification for that event arrives. Since it does not give any guarantee about operations before, we have modeled the event_wait as an acquire operation, i.e., operations before an event_wait can be initiated or completed after the event_wait.

IV. Finish and Cofence in Practice

In this section, we present an evaluation of cofence and finish constructs in CAF 2.0 by implementing a micro-benchmark, and two algorithm kernels: RandomAccess and Unbalanced Tree Search. We choose these two kernels in order to demonstrate the expressive power of asynchronous copy and function shipping when combined with the finish construct and event objects in Coarray Fortran 2.0. The micro-benchmark, and UTS experiments have been carried out on a Cray XK6 Supercomputer (Jaguar) at ORNL, whereas the RandomAccess experiments have been carried out on a Cray XE6 supercomputer (Hopper) at NERSC.

A. Cofence micro-benchmark

We evaluate the utility of cofence using a producer consumer micro-benchmark. Figure 11 shows a sketch of our micro-benchmark. We await the completion of asynchronous copies using either a cofence, an event_wait, or a finish. In the benchmark variant using a cofence, the producer (process 0) uses local data completion to determine that it can reuse the src buffer. It is able to prepare the src buffer for the next round without waiting for delivery of the buffer contents to the destination process image. The benchmark variant using event_wait waits for each copy to be delivered to its destination process image. This variant incurs a delay proportional to the communication latency. The variant using finish waits for global completion of all copies and incurs a cost proportional to $O(log\ p)$ communication latencies. Figure 12 shows the cost of these three synchronization strategies. The cofence variant is the fastest. The event variant is slower because it must wait for data to be delivered to the destination image before execution can proceed. The finish variant is the slowest because it enforces global completion.

B. RandomAccess Benchmark

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

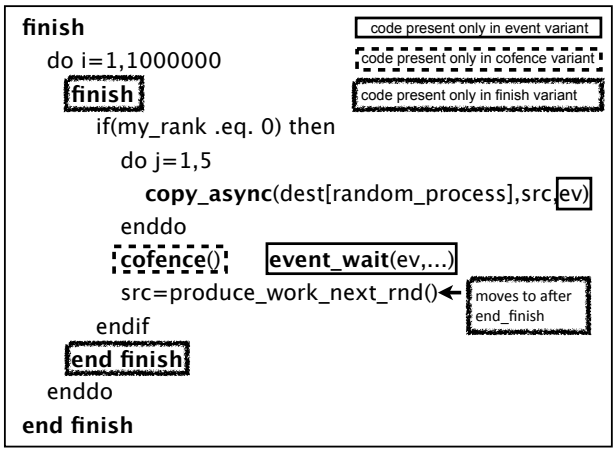

Fig. 11. Sketch of the micro-benchmark to depict use of cofence and copy_async. The size of the copied data is 80 bytes.

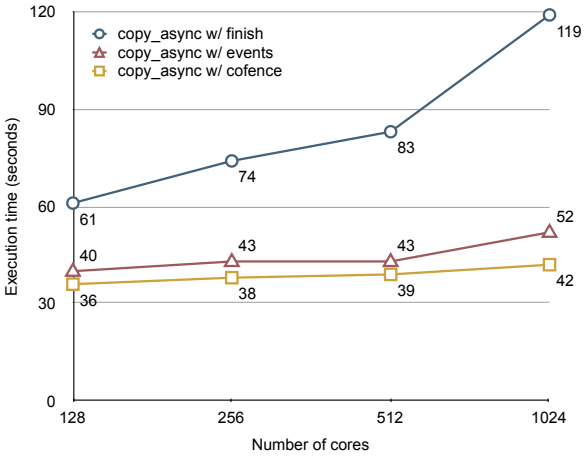

Fig. 12. A micro-benchmark showing the advantage of using a cofence with copy_async. The size of the copied data is 80 bytes, and the number of random processes to which data is sent in each iteration is 5.

to measure the performance of both function shipping in CAF 2.0 and our finish implementation. We implemented two versions of RandomAccess in CAF 2.0. In the reference version, each update performs a get of the entry in the distributed table, performs a local update, and then writes the updated value back using a put. The reference version has data races: a put can happen between a get/put pair updating a location. Using function shipping, we wrote a version of RandomAccess benchmark that uses shipped functions to perform read-modify-write operations on the process where a table entry resides. Thus, the get and put operations in RandomAccess become local load and store operations. Moreover, shipped functions guarantee the read-modify-write operations are atomic. To measure the performance of our finish algorithm, we grouped 2048/1024/512 updates in a bunch; we use a finish block to enclose a bunch, so that 2048/4096/8192 instances of our finish algorithm will be invoked when we update a local table of 2^{22} entries.

With RandomAccess, we demonstrate that our implementation of function shipping in CAF 2.0 is efficient and that, given a sufficient number of asynchronous operations, the cost

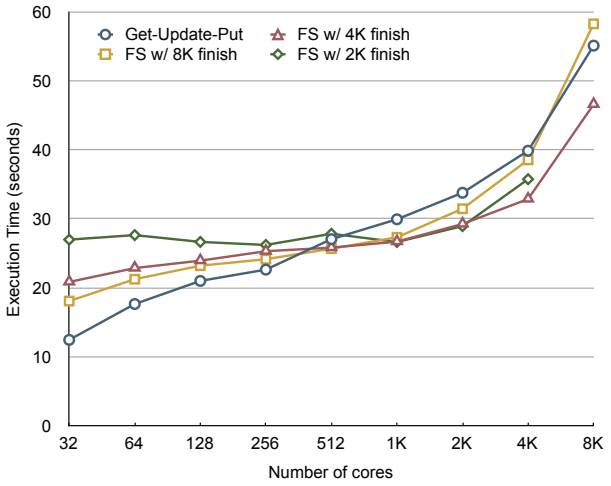

Fig. 13. A comparison of two implementations of the RandomAccess benchmark in CAF 2.0 (FS = function shipping). Program is compiled using PGI compiler, and ran on Jaguar using 8 cores per node.

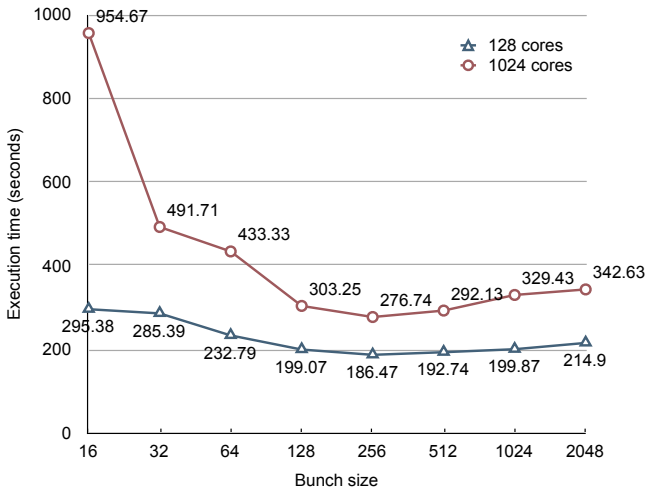

Fig. 14. RandomAccess using function shipping with various bunch sizes. The local table size for each core is 2^{23}. Fortran generated from CAF 2.0 is compiled with the PGI compiler and run on Hopper using 8 cores/node.

of synchronization using `finish` is trivial. Figure 13 shows a comparison of two implementations of RandomAccess benchmark. We measure the running time of applying remote updates on a distributed table of size 8MB on each process image. From 32 to 8192 processes, we see that the performance of our implementation using function shipping is comparable to that of our implementation using `get` and `put`, which exploits RDMA on Jaguar. We do not see a significant difference in execution time when varying the number of invocations of `finish`; this shows that the overhead of synchronization using `finish` in RandomAccess is not excessive.

Figure 14 shows the function shipping implementation of RandomAccess running on 128 and 1024 cores using different bunch sizes. From the figure we see that the overhead of `finish` synchronization dominates the actual work in RandomAccess using a bunch size of 16. As the bunch size increases, the cost of synchronization decreases; the cost of `finish` is trivial when the bunch size is larger than 256. In our experiments, we observed that when using a bunch size larger than 256, less synchronization actually hurts performance. We believe this performance anomaly is related to flow control in GASNet; however, further experimentation will be needed to prove or disprove our intuition.

C. Unbalanced Tree Search

The Unbalanced Tree Search(UTS) benchmark [24] involves building and counting the number of nodes in an unbalanced n-ary tree; the tree is based on a geometric/binomial distribution. Each node in this tree is characterized by a 20-byte descriptor that is used to determine the number of children for the node and (indirectly) for all of its children. A node's descriptor is computed by applying an SHA1 hash on its parent descriptor and the child's index. For more details about UTS, see Olivier et. al. [24]. Since each node's children are determined by its descriptor, the parent-child links need not be explicitly maintained: the UTS tree is virtual.

1) Challenges: The geometric distribution of nodes in the tree causes there to be a great deal of imbalanced work in this benchmark. Randomized work stealing is a good strategy for load balancing as shown by Blumofe [25]. However, work stealing is problematic for UTS for the reasons that follow.

a) Amount to steal: The stolen work might yield only a small subtree before a thread runs out of work. In effect, the time spent in a network transaction to check if a process has work, lock the process's work queue, and then steal the work might not be fruitful. A few papers [24], [26] have looked at this in detail. In our CAF 2.0 prototype, the amount of work each shipped function can steal is limited by the *ActiveMessageMediumPacket* size (a max of 9 items) in GASNet's implementation [16].

b) Timing of the steal: The steal attempt from a image p might land at image q when q is working on a few nodes. Concluding that q has a small amount of work would be erroneous if the nodes on q give rise to a large subtree.

2) Algorithm: Hence, we implement a composite solution of work sharing and work stealing which has been proposed and evaluated by Saraswat et. al. [27]. The highlights of our implementation are as follows:

a) Initial work sharing: Process image 0 builds the initial few levels of the UTS tree and then distributes these nodes to different process images. Each image then works on building the rest of the tree rooted at these nodes.

b) Randomized work stealing: Each process image, after finishing its quota of work, tries to steal work (i.e. nodes whose subtree has not been constructed) from a random process image. If a process image is unsuccessful in n steal attempts (set to 1 in our experiments), it then quiesces by setting up *lifelines* [27]. Line 15 in Figure 15 depicts an attempt to steal from a remote node.

c) Work sharing via lifelines: Process image `A` sets up a *lifeline* on another image `B` to indicate that `A` is available for any excess work that `B` obtains in the future. In our implementation, a process image sets up lifelines on each of its hypercube neighbors (process images at offsets 2^0, 2^1, ...,

```
finish
  ! while there is work to do
  do while(q_count .gt. 0)
    call deq_back(descriptor)
    call work(descriptor)
    !check if someone needs work
    if (incoming_lifelines.ne.0) then
      foreach lifeline
        spawn push_work()[lifeline]
      endfor
    endif
  enddo
  ! attempt to steal from another image
  victim = get_random_image()
  spawn steal_work()[victim]
  ! set up lifelines
  do i = 0, max_neighbor_index-1
    neighbor = xor(my_rank, 2**i)
    spawn set_lifelines(my_rank,i)[neighbor]
  enddo
end finish
```

Fig. 15. Outline of the UTS benchmark.

$2^{\log_2 p}$, on p images). Process image A uses function shipping to establish lifelines on the remote nodes. Since the spawn executes on the process image where the lifeline representation resides, the communication overhead for this operation is reduced to a single round trip.

d) Termination detection using `finish`*:* We employ `finish` to capture the completion of the shipped steal/share functions; share functions encapsulate the tasks of pushing excess work to lifelines, and setting up of lifelines. A barrier alone will not suffice to capture the completion of these functions. This is because when a process participates in a barrier it should indicate completion of all tasks assigned to it. In this implementation, a process never knows when it has completed all tasks; work can always be pushed to it via lifelines from another process. `finish`, on the other hand, allows a process to participate in a completion consensus, ensuring a minimal number of consensus rounds, and enabling a process to be receptive to incoming work.

3) Performance Analysis: Our implementation of UTS is based on the T1WL version of the UTS benchmark. We implement a UTS tree based on a geometric distribution with expected children size per node of 4, maximum tree depth of 18, and initial seed for the root descriptor as 19.

We present the results of our implementation in Figures 16 and 17. Figure 16 shows the load balance achieved by the implementation on 2048, 4096, and 8192 processes. In the run with 2048 processes, we see that there is lesser variance of the work completed across the processes (from 0.989x to 1.008x) whereas in the run with 8192 processes, we see that there is, relatively, higher variance of the work completed (0.980x to 1.037x). We attribute this to the reduction in the probability of finding work by a process (after it finishes its quota of work) in a larger run. This reduction of work is particularly true in the last stages of the computation.

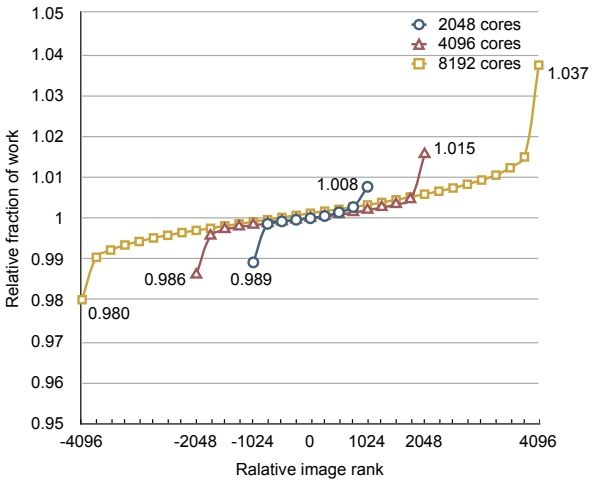

Fig. 16. Load balance achieved by the CAF 2.0 UTS implementation on 2048, 4096, and 8192 processes on Jaguar.

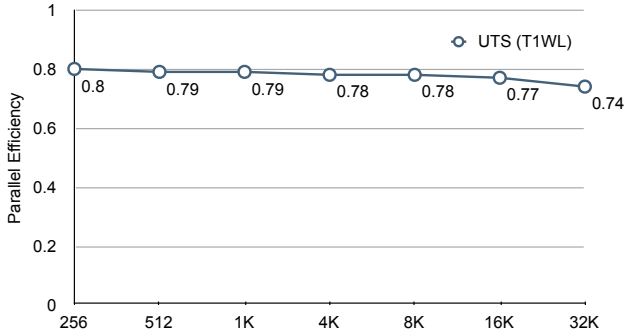

Fig. 17. Parallel efficiency of the CAF 2.0 UTS benchmark application on 256-32678 processes on Jaguar. Efficiencies are w.r.t. single core performance.

With our UTS implementation, we show that the finish construct is capable of handling the termination detection for UTS. This justifies its presence in CAF 2.0. As shown in Figure 17, we achieve 74% parallel efficiency for 32768 processes. Moreover, we achieve comparable efficiency from 256–32768 processes. This shows that the overhead added by the `finish` construct in termination detection does not dramatically increase with machine size. Also, we show in Figure 18 that enforcing an upper bound on the number of speculative waves of termination detection is effective and beneficial. In Figure 18, the number of `allreduce` operations used for detecting termination in our algorithm is about 50% the number of `allreduce` operations used by an algorithm that does not wait for delivery and completion of shipped messages before starting termination detection.

V. Related Work

The ARMCI communication library [28] provides completion semantics similar to CAF 2.0. ARMCI supports three forms of completion for non-blocking operations: initiation, local, and global. Return from a non-blocking ARMCI operation indicates only initiation completion. `ARMCI_Wait` waits for local completion of a non-blocking operation. `ARMCI_Fence`

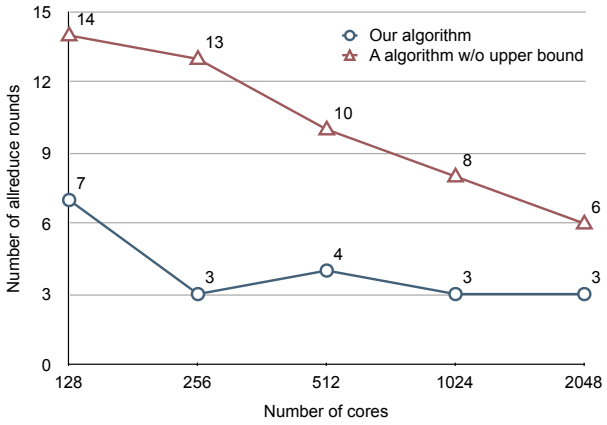

Fig. 18. Rounds of detecting termination used in UTS.

ensures that non-blocking operations are complete on remote processes. Because ARMCI does not provide asynchronous collective operations and function shipping, it neither distinguishes local data and local operation completion, nor does it need a global termination detection construct as CAF 2.0 does.

CAF 2.0 `finish` is inspired by the finish construct in X10. Although they serve the same purpose—synchronizing asynchronous activities—their semantics and implementation differ greatly. The implementation of X10's termination detection uses a *vector counting* algorithm to detect global termination [27]. Each worker maintains a vector that contains a count per place, tracking the number of activities it spawned remotely and completed locally. Once a worker has quiesced (no active tasks in this place), it sends its vector to the place that owns the `finish`. Global termination is detected once the place that owns the `finish` receives vectors from everyone, and the sum-reduced vector is zero. This algorithm suffers from the fact that a single place receives p vectors of size p, where p denotes the number of places. This will be a bottleneck in computations on a large number of places.

Algorithms to determine global completion—*distributed termination detection* algorithms [29]—can be broadly classified as *asymmetric* and *symmetric* algorithms. Asymmetric algorithms rely on a pre-designated process for termination detection. In symmetric algorithms all processes execute identical code and detect termination together. *Scioto* [30], a framework for dynamic load balancing on distributed memory machines, uses a wave-based algorithm similar to that proposed by Francez and Rodeh [31] to detect global termination. Both X10 and Scioto use asymmetric termination detection algorithms; since all dynamic tasks in X10 and Scioto share the same ancestor, it is a natural place to oversee termination detection. Because computation starts from multiple places in CAF 2.0, these algorithms are not suitable for use by CAF 2.0 `finish`. AM++ [32] employs an algorithm that uses four counters [17, §4] and non-blocking global sum reductions to accumulate the counts to determine global termination. The four-counter algorithm *twice* counts the messages sent and received by each process. Equality of these four counters guarantees correct detection of termination by the system. Because this algorithm counts twice, it always incurs an extra global reduction to detect termination; our algorithm does not pay this extra cost.

The CAF 2.0 `cofence` is inspired by the SPARC V9 [7] MEMBAR instruction (for loads and stores to memory only), which has two variants: *ordering* and *sequencing*. The ordering MEMBAR allows precise control over what reorderings it will restrict. Ordering requirements are specified using a bit mask. For example, a MEMBAR #LOADLOAD requires that loads before the MEMBAR complete before any loads after the MEMBAR; this flavor of MEMBAR does not restrict the ordering of stores.

UPC's `upc_fence` [33] separates relaxed operations on shared data before and after the fence. In contrast, CAF 2.0 `cofence` enables a programmer to indicate that some kinds of reorderings are allowed across the fence, which provides greater flexibility for performance tuning. Also, the asynchrony support in UPC is limited to `get` and `put` operations with explicit completion guarantees.

The GASNet communication library [16] provides non-blocking `get` and `put` operations. It provides several routines to control the completion of implicit asynchronous operations. These routines, however, only capture global completion of asynchronous operations, and do not provide the finer grained direction control `cofence` provides. Fortran 2008 provides three primary synchronization constructs: `SYNC ALL`, `SYNC IMAGES`, and `SYNC MEMORY`. `SYNC MEMORY` prevents the movement of any FORTRAN statement that has side effects visible to other process images. `cofence` provides finer-grain control than `SYNC MEMORY`. `SYNC ALL` is redundant with CAF 2.0 `team_barrier(TEAM_WORLD)`. A CAF 2.0 team barrier, which is collective, is more efficient than `SYNC IMAGES` across many images, which must be pairwise. Thus, we drop all versions of SYNC from CAF 2.0.

VI. Conclusions

Avoiding exposed communication latency by manipulating data locally or overlapping communication latency with computation is essential for achieving high performance on scalable parallel systems. Mechanisms for asynchronous communication and function shipping are useful for this purpose. However, asynchronous operations are useless without completion guarantees. In this paper, we describe four notions of completion that are important to programmers: initiation completion, local data completion, local operation completion, and global completion. We describe the properties of two synchronization constructs, `cofence` and `finish`, and how these synchronization constructs interact with asynchronous operations. In conjunction with events, these synchronization constructs enable users to coordinate local data completion, local operation completion, and global completion of asynchronous operations in CAF 2.0. With experiments using up to 32K cores, we show that our synchronization constructs represent practical primitives useful for developing CAF 2.0 programs for scalable parallel systems.

ACKNOWLEDGMENTS

This research was supported by the DOE Office of Science under cooperative agreements DE-FC02-07ER25800, DE-FC02-06ER25754, and DE-SC0008883. This research used resources of the National Energy Research Scientific Computing Center (NERSC) and the National Center for Computational Sciences (NCCS) at the Oak Ridge National Laboratory. Both facilities are supported by the DOE Office of Science. NERSC is supported under Contract No. DE-AC02-05CH11231 and NCCS is supported under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] J. Reid, "Coarrays in Fortran 2008," in *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, ser. PGAS '09. New York, NY, USA: ACM, 2009, pp. 4:1–4:1.

[2] R. W. Numrich and J. Reid, "Co-array Fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, 1998.

[3] J. Mellor-Crummey, L. Adhianto, and W. N. Scherer III, "A critique of co-array features in Fortran 2008," Fortran Standards Technical Committee Document J3/08-126, February 2008, http://www.j3-fortran.org/doc/meeting/183/08-126.pdf.

[4] 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*. New York, NY, USA: ACM, 2009, pp. 1–9.

[5] K. Ravichandran, S. Lee, and S. Pande, "Work stealing for multi-core hpc clusters," in *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*, ser. Euro-Par'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 205–217.

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

[7] SPARC International, Inc., "The SPARC architecture manual, version 9," http://bit.ly/yPbTO6.

[8] 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*, ser. IPDPS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1089–1100.

[9] 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 Fourth Conference on Partitioned Global Address Space Programming Model*, ser. PGAS '10. New York, NY, USA: ACM, 2010, pp. 14:1–14:9.

[10] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 53:1–53:11.

[11] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[12] S. V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," Digital Western Research Laboratory, Palo Alto, CA, Technical Report 95-7, 1995.

[13] S. Adve, "Designing memory consistency models for shared-memory multiprocessors," University of Wisconsin-Madison, Madison, WI, USA, Tech. Rep. 1198, 1993.

[14] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 519–538.

[15] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer, "Phasers: a unified deadlock-free construct for collective and point-to-point synchronization," in *Proceedings of the 22nd Annual Intl. Conference on Supercomputing*. New York, NY, USA: ACM, 2008, pp. 277–288.

[16] D. Bonachea, "GASNet specification, v1.1," University of California at Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/CSD-02-1207, 2002.

[17] F. Mattern, "Algorithms for distributed termination detection," *Distributed Computing*, vol. 2, no. 3, pp. 161–175, 1987.

[18] E. W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren, "Derivation of a termination detection algorithm for distributed computations," *Inf. Process. Lett.*, vol. 16, no. 5, pp. 217–219, 1983.

[19] K. M. Chandy and J. Misra, *A paradigm for detecting quiescent properties in distributed computations*. New York, NY, USA: Springer-Verlag New York, Inc., 1985, pp. 325–341.

[20] D. Kumar, "A class of termination detection algorithms for distributed computations," University of Texas at Austin, Austin, TX, USA, Tech. Rep. CS-TR-85-07, 1985.

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

[22] Intel Corporation, "IA64 volume 2: System architecture, revision 2.3," http://intel.ly/zEgKqN.

[23] Message Passing Interface Forum, "MPI Report 2.0," http://1.usa.gov/x6sKLG.

[24] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng, "UTS: an unbalanced tree search benchmark," in *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing*, ser. LCPC'06. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 235–250.

[25] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, pp. 720–748, September 1999.

[26] Min, I. Seung-Jai, Y. Costin, and Katherine, "Hierarchical work stealing on manycore clusters," in *Proceedings of Fifth Conference on Partitioned Global Address Space Programming Models*, ser. PGAS'11, 2011.

[27] V. A. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy, "Lifeline-based global load balancing," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, pp. 201–212.

[28] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. K. Panda, "High Performance Remote Memory Access Communication: The ARMCI Approach," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 233–253, May 2006.

[29] N. Francez, "Distributed termination," *ACM Trans. Program. Lang. Syst.*, vol. 2, pp. 42–55, January 1980.

[30] J. Dinan, S. Krishnamoorthy, L. Brian, J. Nieplocha, and P. Sadayappan, "Scioto: A Framework for Global-View Task Parallelism," in *Proc. of the 37th Intl. Conference on Parallel Processing*, sept. 2008, pp. 586–593.

[31] N. Francez and M. Rodeh, "Achieving distributed termination without freezing," *IEEE Transactions on Software Engineering*, vol. SE-8, no. 3, pp. 287–292, May 1982.

[32] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "AM++: a generalized active message framework," in *Proceedings of the 19th Intl. Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 401–410.

[33] UPC Consortium, "Formal UPC memory consistency semantics, UPC Specification 1.2," http://1.usa.gov/zyHyO0.