A New Vision for Coarray Fortran

John Mellor-Crummey, Laksono Adhianto, and William Scherer III

Department of Computer Science
Rice University
Houston, TX, USA
{johnmc, laksono, scherer}@rice.edu

Abstract—**In 1998, Numrich and Reid proposed Coarray Fortran as a simple set of extensions to Fortran 95 [7]. Their principal extension to Fortran was support for shared data known as *coarrays*. In 2005, the Fortran Standards Committee began exploring the addition of coarrays to Fortran 2008, which is now being finalized. Careful review of drafts of the emerging Fortran 2008 standard led us to identify several shortcomings with the proposed coarray extensions. In this paper, we briefly critique the coarray extensions proposed for Fortran 2008, outline a new vision for coarrays in Fortran language that is far more expressive, and briefly describe our strategy for implementing the language extensions that we propose.**

I. Introduction

In 1998, Numrich and Reid proposed a small set of extensions to Fortran 95 to support parallel programming that they dubbed Coarray Fortran (CAF) [7]. They envisioned CAF as a model for SPMD parallel programming based on a static collection of asynchronous process images (known as *images* for short) and a partitioned global address space. Their principal extension to Fortran was support for shared data in the form of coarrays. Coarrays extend Fortran's syntax for type declarations and variable references with a bracketed tuple that is used to declare shared data or access data associated with other images. For example, the declaration

```
integer :: a(n,m)[*]
```

declares a shared coarray with $n \times m$ integers local to each image. Dimensions in the bracketed tuple are called codimensions. Coarrays may be declared for primitive or user-defined types. The data for a coarray associated with an image may be a singleton instance of a type rather than an array of type instances. Instead of explicitly coding message exchanges to obtain data belonging to other images, a CAF program can directly access a coarray associated with another image by appending a bracketed tuple to a reference to a coarray variable. For instance, any image can read the first column of data in coarray a from image p by executing the right-hand side reference a(:,1)[p].

Numrich and Reid's design for CAF included several synchronization primitives. The most important of these are the synchronous barrier `sync_all`; `sync_team`, which is used for synchronization among dynamically-specified teams of two or more processes; and `start_critical/end_critical`, which control access to a global critical section.

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. Coarrays are shared data allocated collectively across all images. A coarray can have multiple codimensions enabling one to conveniently index a coarray distributed over a grid of process images that is logically multidimensional. Our earlier criticisms about Numrich and Reid's teams in CAF supporting only all-pairs communication rather than efficient collective operations led the Fortran Standards Committee to consider support for pre-arranged image teams. Unfortunately, support for image teams has been tabled for Fortran 2008, although it may be considered for inclusion in the future. A previous detailed critique [5] of the coarray extensions proposed for Fortran 2008 and a recent review of the latest working draft for Fortran 2008 [3] revealed several shortcomings in emerging coarray extensions that limit their expressiveness:

- Coarrays must be allocated over all images; there is no support for processor subsets.
- Coarrays must be declared as global variables; one cannot dynamically allocate coarrays into a local variable in a scope.
- The coarray extensions lack any notion of remote pointers, which are essential for remotely manipulating of any kind of linked data structure.
- Reliance on named critical sections for mutual exclusion hinders scalable parallelism.
- Fortran 2008's `sync images` statement (a reworked version of Numrich and Reid's `sync_team`) enables one to synchronize directly with one or more images; however, this construct doesn't provide a safe synchronization space. As a result, synchronization operations in user's code that are pending when a library call is made can interfere with synchronization in the library call.
- There are no mechanisms to avoid or tolerate latency when manipulating data on remote images.

- There is no support for collective communication.

These shortcomings caused us to rethink the CAF model. Our interest is in developing an expressive set of parallel extensions for Fortran that map well onto parallel systems of all sizes, ranging from multicore nodes to petascale platforms. In this paper, we propose a new vision for coarray-based extensions to the Fortran language. Our design focuses on three core tenets: orthogonality, expressiveness, and simplicity. In a nutshell, it provides full support for processor subsets, logical topologies that are more expressive than multiple codimensions, dynamic allocation of coarrays, scalable mutual exclusion, safe synchronization spaces, latency hiding, collective communication, and a memory model that enables one to trade ease of use for performance.

II. Coarray Fortran 2.0

Here, we describe an expressive set of coarray-based extensions to Fortran that we believe provide a productive parallel programming model. Compared to the emerging Fortran 2008, our coarray-based language extensions include some additional features:

- **process subsets** (§III), which support coarrays, collectives, and relative indexing for pairwise operations
- **topologies**, which augment teams with a logical communication structure (§IV)
- **dynamic allocation/deallocation of coarrays** (§V)
  - local variables within subroutines: declaration and allocation of coarrays inside procedures scope is critical for library based-code.
  - team-based coarray allocation and deallocation
  - global pointers in support of dynamic data structures (§VI)
- **enhanced support for synchronization** (§VII) for fine control over program execution
  - safe and scalable support for mutual exclusion (§VII-A)
  - split-phase barriers (§VII-B) for overlapping communication and computation
  - events, which provide a safe space for point-to-point synchronization (§VII-C)
- **collective communication** (§VIII)
- **a memory model** (§IX) that enables one to trade ease of use for performance

Most of these ideas are inspired by features in MPI [10] and Unified Parallel C [2]. Here, we describe their realization as a cohesive whole to support parallelism in Fortran.

III. Process subsets

Processor subsets is a useful abstraction for decomposing work in a parallel application. Processor subsets

---

team_split(existing_team, color, key, new_team,
       new_color = result_color)
       err_msg = emsg_var)

color, key, and result_color are integers
existing_team and new_team are team variables
emsg_var is a scalar character variable

Fig. 1. Team_split: Forming new teams from an existing one.

---

can be used in coupled applications (*e.g.*, *ocean* and *atmosphere* subsets in a climate application) as well as within dense numerical linear algebra (*e.g.*, *row* and *column* subsets to coordinate operations such as pivot selection and row interchange). Drafts of Fortran 2008 included support for image teams; however, these teams were designed solely to support collective communication. Here we describe a broader vision for teams.

In CAF 2.0, a *team* is a first-class entity that consists of an ordered sequence of process images. Teams need not be disjoint and a process image may be a member of multiple teams. A team serves three purposes. First, it represents a set of process images. This set of images can serve as a domain onto which coarrays may be allocated. Second, it provides a namespace within which process images and coarray instances can be indexed by an image's rank r in a team t, where $r \in \{0..\text{team\_size}(t) - 1\}$, rather than an absolute image ID. As identified by Skjellum [9], relative indexing by rank is particularly useful for supporting the development of libraries, where code needs to be reusable across sets of processor images. Third, a team provides a domain for collective communication.

When a CAF program is launched, all process images are initially part of a pre-defined team known as team_world. New teams may be constructed from an existing team by using the collective team_split operation shown in Figure 1. This operation was inspired by the functionality of MPI's MPI_Comm_split [6]. As with MPI_Comm_split, each process image invoking team_split on an existing team provides a positive integer color (or color_undefined) and a key. Images that supply the same positive value for color will be assigned to the same new team. If an image provides the value color_undefined, it will not be assigned a new subteam. Members of a subteam result are ordered by the supplied key; if two members of the existing team supply the same color and key, their rank in the new team will be ordered by their rank in existing_team.

As is well understood, through judicious choice of color and key, one can use team_split to create a new team in which the participating process images are simply a permutation of the images in the existing team,

```fortran
real, allocatable :: x(:,:)[*] ! 2D array
real, allocatable :: z[*]
team :: subset
integer :: color, rank

color = floor((2*team_rank(team_world))
              /team_size(team_world))

! split into two subsets:
!    top and bottom half of team_world
team_split(team_world, color, &
        team_rank(team_world), subset)

! each image allocates a singleton for z
allocate( z [@team_world] )

! members of the two subset teams
! independently allocate their own coarray x
allocate( x(0:n,0:n)[@ subset])

rank = team_rank(subset)
if (rank > 0) then
  x(:,n)[rank - 1 @ subset] = x(:,0)
endif
if (rank < team_size(subset) - 1) then
  with team subset
    x(:,0)[rank + 1] = x(:,n)
  end with team
endif
```

Fig. 2.   Allocating and accessing coarrays on processor subsets.

or to create one or more subset teams. One might create a new team that is a permutation of an existing team to order process images within the new team so that adjacent images are closer in the physical topology of the target platform on which the program is executing.

A new idea here is the ability to bind team pairs by providing an optional `result_color` argument to `team_split`. In the typical case, the new team returned would be based on the `color` argument. However, if one wants to enable a pair of teams to interact, one can construct a team that functions like an MPI intercommunicator by supplying a result color different from `color`. For instance, one could use one team split operation to create *atmosphere* and *ocean* teams. Then, in a second team split operation with the same color and ranks supplied to the initial split, images could specify the *other* team's result color. This would provide members of the ocean team with a team that would enable them to interact with members of the atmosphere team; likewise, members of the atmosphere team would receive a team that enables them to interact with members of the ocean team.

a) **Data allocation:**   Both Numrich and Reid's original CAF and the Fortran 2008 working draft require that coarrays be allocated across all process images. For applications where processor subsets need to work independently, it is unreasonable to ask that all processors be involved if a subset needs to dynamically allocate

some shared data. Second, if one writes a parallel library that might be used concurrently by different processor subsets, it is unreasonable to require that all shared data allocated by the library (a) be known to the library's callers or (b) be associated with global variables within the library package. These observations led to our design, which supports dynamic allocation of coarrays on processor subsets, and dynamic allocation of coarrays into local variables. Unlike prior proposals, we only allow one to specify a single codimension for a coarray in its declaration. Rather than supporting multidimensional coarrays, we support more general structured indexing of process images through topologies associated with teams, which we describe in the next section.

Although coarrays are associated with process images, each coarray allocation or indexing operation is explicitly or implicitly associated with a team. When one allocates or indexes a coarray, one may specify an explicit team. If no team is specified explicitly, the default team is used. A `with team` statement (inspired by the `with` statement in PASCAL) is a block structured construct for changing the default team within its scope. If a coarray allocation or access is not enclosed in the scope of a `with` statement, `team_world` is used as the default team. When one or more coarrays are allocated on images associated with a given team, a barrier synchronization is performed on the team to ensure that all coarrays have been allocated and are ready for use. Indexing with a codimension is done with a relative rank with respect to an explicit or default team. Figure 2 shows examples of allocating coarray variables across different teams, using `team_size` and `team_rank` primitives to interrogate the team characteristics, and indexing coarrays with respect to the team `subset`, specified both explicitly (using @) and implicitly (using a `with`).

IV. Topologies

Fortran 2008 and earlier flavors of CAF only provided multidimensional coarrays as a form of structured namespace for interprocessor communication. Any other structured organization for indexing process images must be implemented in user code using arithmetic on image IDs or using index arrays. In CAF 2.0, we associate a logical topology with a team to provide a structured namespace for intra-team communication that is relative to members' ranks in the team, not to their absolute image ID. Like MPI, CAF 2.0 supports two types of topologies: Cartesian and graph.

For CAF 2.0 we have settled upon a one to one association between teams and topologies. Although it may seem desirable to change the topology for a team, we note that calling `team_split` with a constant for the `color` parameter allows one to create a clone of the

team that has not yet been associated with any topology; a different topology can be used in conjunction with a team's clone.

A. Topology API

To associate a topology with a team, one invokes `topology_bind`, which has parameters for the team and the topology to be associated, and returns an error if a topology has already been associated with the team. Similarly, `topology_get` extracts the topology associated with a team, or returns an error if there is none.

1) Graph topology

Any topology can be expressed as a graph $G = (V, E)$. To create a graph topology in CAF 2.0, one simply calls `topology_graph(n,c)`, where n is the number of nodes in the graph, and c is the number of edge classes. For an undirected graph, one might use a single edge class: neighbors. For a directed graph, one could use two edge classes: successors and predecessors. Additional flavors of edge classes could be used to distinguish edges within or between processor nodes. Our general interface leaves it to the imagination of the user. To populate edge classes in graph `g`, one may call `graph_neighbor_add(g, e, n, nv)` to add one or more image neighbors (`nv` can be a scalar or a vector value) to edge class `e` for image `n`. The operation, `graph_neighbor_delete(g, e, n, nv)` can be used in the course of updating `g`'s edges.

To index a scalar coarray `f` using a graph topology `g` associated with a team `t`, one uses the syntax `f[(e,i,k)@t]`. The tuple (e, i, k) references the kth neighbor of image i in edge class e in the topology bound to `t`. If the team `t` is implicit (*e.g.*, inherited from a `with` statement or `team_world`), the parenthesis of the tuple may be omitted for convenience, simplifying the syntax to `f[e,i,k]`. One can use the intrinsic `graph_neighbors(g, e,n)` to determine the number of image `n`'s neighbors in edge class `e` in graph `g`.

2) Cartesian topology

In a sense, Cartesian topologies are just a subset of general graph topologies; however, they are common enough to merit explicit treatment and custom support. To define a Cartesian topology, one calls `topology_cartesian`, which takes as parameters the extent of each dimension. As toroidal topologies are common for periodic boundary conditions, a negative extent for a dimension indicates that that the topology of the dimension is circular.

Accessing a node in a Cartesian topology can be done by specifying a comma-separated tuple of indices $(d_1, d_2, d_3, ..., d_n)$ where one would otherwise specify an image rank, e.g. $my\_data(3)[(x+1, y+1)@team\_grid]$.

allocate (type_spec :: **allocation_list**,
      STAT=stat_var,
      ERR_MSG=emsg_var,
      SOURCE=src_expr,
      SHARED=is_shared)

`type_spec` is an intrinsic or derived type*
`stat_var` is a scalar integer variable*
`emsg_var` is a scalar character variable*
`src_expr` is a scalar character variable*
`is_shared` is a scalar logical variable†

* Fortran 2003 feature.
† Proposed Coarray Fortran extension.

Fig. 3. The `allocate` statement.

As with graph topologies, if the team is implicit, one may omit the tuple's parenthesis; in this way, we support syntax as simple as multidimensional coarrays, although our indexing support is more general in that any dimension of the Cartesian topology may be circular for periodic boundary conditions.

It is also highly desirable to support relative indexing within a topology. We do this by prepending a backslash to the dimension or tuple that is relative: $foo[\backslash(3, -4)]$ specifies an offset of $(+3, -4)$ from the current image's position in the topology, but $foo[3, \backslash -4]$ is absolute in the first dimension and an offset of -4 in the second.

V. `allocate` **statement**

Executing an `allocate` statement associates storage with a pointer or allocatable. Figure 3 shows the components of an `allocate` statement. For CAF 2.0 we introduce a new optional argument `SHARED`, which indicates whether the target object should be allocated in private memory of the image, or in shared memory co-located with the image. Data accessible from coarrays must be allocated in the shared segment. The shared specifier is necessary because when data is allocated for a linked shared data structure in a parallel program, one typically allocates and initializes an object before linking it into a shared structure. Only as an object is linked does it become clear that it should be shared.

VI. Pointers

As Coarray Fortran was defined in 1998, pointer components were allowed within a coarray of a user-defined type. It was legal to remotely dereference a pointer component within a coarray. Given a user-defined type GRAPH with a pointer component `edgelist(:)` and a coarray `g` of type GRAPH, on image `q` could execute a remote access `g[p]%edgelist(i)`, which would dereference the remote pointer `edgelist` on image `p`. The pointer component `edgelist` could only

be associated with data on one's local image. This style of pointer enables one to allocate and access shared data of size that differs among process images. However, this style of pointer is insufficient for remotely manipulating linked data structures.

Consider a distributed hash table implemented using bucket chains. One might want to count the entries in a remote bucket list by writing a loop like the following:

```
item = table[p]%head
count = 0
do
  if (.not. associated(item)) exit
  count = count + 1
  item => item%next
enddo
```

With the limited pointers originally proposed for CAF, it would not be possible to write such a loop because `item` would need to point to remote data.

To support construction and manipulation of linked distributed data structures, we propose the attribute `copointer` to declare a pointer that one can associate with shared data that may be remote. To ensure that accesses to remote data are textually identifiable, we propose that one add an empty bracket pair when dereferencing a remote `copointer`. We propose the intrinsic `imageof(p)` to determine the target image for a `copointer`. A typical use of `imageof` would be to determine whether a copointer is associated with data on the local image; if so, one can drop the empty bracket pair and access the pointer target locally more efficiently. Figure 4 shows examples of how one may associate, use, and inspect a `copointer`.

VII. Synchronization

A. Mutual exclusion

Based on our feedback [5], locks were added to the most recent working draft of Fortran 2008 to support mutual exclusion. We further support deadlock-free multilock synchronization by allowing the programmer to transparently acquire a set of locks as a single logical operation.

CAF 2.0 provides three language constructs for mutual exclusion.

1) **Lock.** This is the standard mutual exclusion state variable; `lock` and `unlock` statements acquire and release it, respectively.
2) **Lockset.** Locksets foster safety in multi-lock operation by performing acquires of component locks in a globally-defined canonical order.
3) **Critical section.** Critical sections in CAF 2.0 are simply a block-structured construct for acquiring and releasing a lock or lockset, either of which may be dynamically allocated.

```
integer :: wrank, wsize, a(:,:)[*]
integer, copointer :: x(:,:)[*]

allocate(a(1:20, 1:30)[@ team_world]
wrank = team_rank(team_world)
wsize = team_size(team_world)

! associate copointer x with a
! remote section of a coarray
x => a(4:20, 2:25)[mod(wrank + 1, wsize)]

! imageof intrinsic returns the target
! image for x as a rank in team world
prank = imageof(x)

if (prank .eq. wrank) then
    ! update a location on the local image
    ! (unchecked) through the copointer x
  x(7,9) = 4
else
    ! update a location on a remote image
    ! through the copointer x
  x(7,9)[] = 4
endif
```

Fig. 4.   Using a copointer.

Creating a lock or lockset is not a collective operation; neither is acquiring a lock, lockset, or critical section. To associate a lockset with specified locks, one uses the `lockset_create` statement as shown in Figure 5.

Figure 5 shows that lockset `ls` can be acquired by image 1 if and only if all of locks `l1`, `l2`, and `l3` are released (or not yet acquired) by images 2, 3, and 4 respectively. This example shows the advantages of using locksets: convenience, efficiency and deadlock-freedom.

B. Split-phase barriers

Split-phase barriers enable one to overlap communication with computation. Split-phase barriers comprise two functions: `barrier_notify`, a signal that the image has completed all work upon which other images depend following the barrier; and `barrier_wait`, a query awaiting all other images' completion of at least the notify portion of the barrier, implicitly proclaiming that the current image has completed all work before the barrier. Split-phase barriers can operate on processor subsets. Thus, multiple split-phase barriers can be simultaneously active – even for the same image – without conflict [5].

The syntax of our split-phase barrier is as follows:

```
barrier_notify([team])
barrier_wait([team])
```

If no `team` is specified, the current default team is used (see the `with team` construct in §III).

```
type(Lock_Type) :: l1, l2, l3
type(Lockset_Type) :: ls
ls = lockset_create(/ l1, l2, l3 /)

! ... image 1              ! ... image 2         ! ... image 3        ! ... image 4
lockset (ls)              lock (l1)             lock (l2)           critical (l3)
  ... ! critical region     ....                  ....                ...
unlockset (ls)            lock (l2)             unlock (l2)         end critical
....                        ...
                          unlock (l2)
                          unlock (l1)
```

Fig. 5. Example of using lock, lockset, and critical. In this example, image 4's use of the `critical` construct simplifies programming and ensures that `l3` is unlocked.

C. Events

Because costly group communication is not always necessary to support the coordination needs of applications, we envision point-to-point synchronization via events. At the most basic level, an event is a shared counter object that supports two operations: an atomic increment (a `notify` operation), and spinning until a particular count is reached (an `await` operation). Images may allocate coarrays of events as their needs demand. Remote update via `event_notify` and local spin operations using `event_wait` are all that is needed to effect safe one-way synchronization between pairs of images. Unlike Fortran 2008's `sync images`, events offer a safe synchronization space: libraries can allocate their own events that are distinct from events used in a user's code.

VIII. Collective communication

Collective subroutines are not new in CAF 2.0; they were part of the 2007 draft of Fortran 2008, which also includes collective team reduction and some pre-defined collective subroutines such as `co_sum`, `co_maxval`, `co_minval`, and `co_product`. However, the emerging Fortran 2008 standard does not include these features and even though collective operations are widely used in parallel applications. Built-in collective operations are likely to provide better performance and portability if they are implemented as part of a language runtime rather than having users roll their own.

We propose the collective statements shown in Table I for CAF. These operations inspired by MPI's collectives. All collective statements require a local data variable source (`var_source`), a target Coarray Fortran variable (`var_dest`) and optionally the team where all image members will participate. If a team is not explicitly specified, then the team specified in an enclosing `with` statement or `team_world` will be used.

The `co_reduce` function collectively reduces (local) variables `var_src` within team `team` into a specified target coarray `var_dest` in image `image_dest`. The reduction operation `type_op` can be one of the

predefined operations such as `co_sum` for summation, `co_max` for finding the maximum value, `co_min` for finding the minimum value and `co_prod` for product multiplication. The only difference between `co_reduce` and `co_allreduce` is that the former will store the result in a coarray in the target image `root_rank`, while the latter propagates the results to all images within the team `team`.

See Reid and Numrich [8] for a further discussion about collectives.

IX. Memory consistency models

Although a strict consistency memory model is helpful to ensure correctness of the program, it can preclude some optimizations for program performance. Programmers and compilers need the flexibility to effect relaxed instruction ordering for regions of code where overall system performance is the paramount concern.

Our vision for Coarray Fortran supports three memory models that may be selected for program regions via compiler directives:

`!$caf consistency(strict)`.The strict consistency model enforces sequential consistency for coarray communication. Memory operations may only execute after previous ones complete, and all updates become visible immediately. This is the default model.

`!$caf consistency(relaxed)`. Under the relaxed consistency model, memory get and put operations can be reordered by the compiler and runtime systems; however, processor consistency is guaranteed: absent some other intervening write, an image is guaranteed to read back the last value it has written.

`!$caf consistency(none)`. Unlike the relaxed model, an image will not necessarily read back the latest value it has written to a remote node, again assuming no intervening writes by other images. This can occur, for example, in complex interconnection networks with dynamic routing if the read is routed as as to arrive at the destination image node ahead of the write. From an implementation perspective, this mode essentially disables all runtime control over message ordering in

TABLE I
COLLECTIVE STATEMENTS SUPPORTED IN COARRAY FORTRAN 2.0

Statement	Description	Syntax
co_broadcast	collectively broadcast a data from an image to all images in a team	co_broadcast(var_source, var_dest, root_rank [, team])
co_gather	collect individual data from each image in a team at one image	co_gather(var_source, var_dest, root_rank [, team])
co_allgather	gathers data from all images and distribute it to all images	co_allgather(var_source, var_dest [, team])
co_reduce	collective reduction, the result is stored to an image of the team	co_reduce(var_source, var_dest, root_rank, operator [, team])
co_allreduce	collective reduction, the result is stored to all images of the team	co_allreduce(var_source, var_dest, operator, [, team])
co_scan	collective partial reduction (scan), each image store the result of reduction from its neighbour	co_scan(var_source, var_dest, [, team])
co_scatter	distribute individual data from an image to each image in a team	co_scatter(var_source, var_dest, root_rank [, team])
co_shift	move data from another image at an offset within a team	co_shift(var_source, var_dest, image_offset [, team])

For most statements
typedef::var_source   local source variable
typedef::var_dest[*]   target Coarray Fortran variable
integer::root_rank   the target image
team::team   process subset (team_world if not specified)

For the reduction statements:
integer::operator
Possible operator: co_max,
co_min, co_sum, co_prod

order to maximize performance; programmers should only use it in cases where they are sure that it is safe and high performance is critical.

X. Implementation

Our implementation of Coarray Fortran 2.0 is a work in progress. Here, we sketch our implementation strategy, which is based on the GASNet communication library [1]. We use GASNet's get and put operations to read and write remote coarray elements. We further use GASNet's active message support to invoke operations on remote nodes. This capability is used during team formation and to look up information about remote coarrays so that one can read and write them directly.

A. Team representation

We use a scalable representation of image teams that is based on the concept of pointer jumping. Each image in a team of size S has $\lceil \log S \rceil$ levels of pointers to a successor and a predecessor. For image i, pointers on level k link i to the representations of team members at ranks $(i + S - 2^k) \mod S$ and $(i + 2^k) \mod S$.

With this representation, each image has enough information to locate an image at any rank. To reach rank j in a team from rank i in a team of size S, one can obviously do this in at most $\log S$ steps by following a chain of pointer-jumping links at distances corresponding to the bits in $i \oplus j$. Less obvious is that for rank i to locate j, one can often follow far fewer links than the number of one bits $i \oplus j$ by exploiting the circularity of our doubly-linked list based representation, and making use of both forward and backward links (*e.g.*, instead of

using three forward power-of-2 hops to accomplish a route of distance +7, one can use a forward route of distance 8 and backward route of distance 1). For a team of size S, where S is not a power of two, one can also exploit the fact that $(i - j) \mod S \neq (j - i) \mod S$. For performance, we plan to have images cache information about how to directly communicate with a fixed modest number of frequent communication partners within one's team.

B. Team formation

Scalable distributed team formation via team_split can be accomplished by sorting (color, key, rank) tuples using parallel bitonic sort, left and right shift operations to determine team boundaries, along with segmented scans to compute one's rank within a team and disseminate the identity of the first and last members of the team and the team size. Subteams can be assembled once each image knows its left and right neighbors at distance one in the circular order of its subteam, the size of the subteam, and its rank in the subteam. Our approach enables us to form a team without using more than $O(\log^2 P)$ space on any image; we use this much space as a scratch buffer for parallel bitonic sort.

C. Collective operations

Our proposed pointer-jumping based representation for a team contains all of the direct connections necessary to support collective communication within a team. **Blocking Barrier** The dissemination barrier algorithm [4] uses all of the direct connections in our pointer-
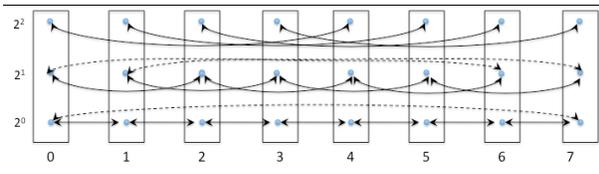
Fig. 6. Members of a team of size S are linked in $\lceil \log S \rceil$ doubly-linked circular lists. In list i, $0 \le i < \lceil \log S \rceil$, a team member at rank j is linked to team members $(j+S-2^i) \bmod S$ and $(j+2^i) \bmod S$, an organization inspired by pointer jumping.

jumping representation for teams. On a team of size S, it involves $\lceil \log S \rceil$ rounds of communication. In round i, $0 \le i < \lceil \log S \rceil$, each image sets a flag on a successor at distance 2^i and spin waits locally.

Broadcast and reductions Tree-based broadcast and reduction operations naturally map onto direct connections in our pointer-jumping representation for teams. For both broadcast and reductions, we use a binomial tree [11] based communication pattern. Since the linked lists in our pointer-jumping representation are circular, they naturally support broadcasts and reductions rooted at any node. Allreduce maps onto direct connections of the pointer-jumping representation equally well.

Replication-oblivious reductions (*i.e.*, operations such as `min` and `max` for which repeated incorporation of a datum would not change the result) could be implemented using a dissemination-based [4] pattern.

D. Locks and events

We support spin-waiting on remote locks and events. Spin waiting on remote locks or events requires only a constant number of messages across a machine's interconnect. Waiting on a remote lock or event causes a record indicating the waiting image to be enqueued on the remote node. At the appropriate time, an active message will signal the waiting image.

E. Copointers

We represent a copointer as a tuple consisting of the target image ID and a Fortran 90 pointer. The target image ID is the image's rank in `team_world`. We initialize a copointer's Fortran 90 pointer for a section of a remote coarray by simply copying the dope vector for the remote coarray to the local image and locally computing the proper subsection. When accessing remote data through a copointer, the data in the copointer representation suffices to synthesize a `get` or `put` operation to access a virtual address in the target image.

XI. Summary and future work

We have sketched a new vision of Coarray Fortran and are actively implementing the features described herein. CAF 2.0 is a work in progress; many details of syntax

remain to be designed, such as support for user-defined reduction operations. Nevertheless, the path forward is clear and our new design is vastly more expressive than prior coarray extensions. CAF 2.0's support for teams consisting of process subsets, coarrays allocated on processor subsets, dynamic allocation of coarrays, copointers, collectives on process subsets, and events for safe pairwise synchronization, represents substantially richer support for parallelism than the coarray extensions in Fortran 2008.

Once we have the core of CAF 2.0 operational, we plan to extend our design with support for remote invocation of special functions that we call cofunctions. The motivation for cofunctions is latency avoidance. However, adding cofunctions to CAF leads to multithreaded images, which increases programming complexity. If one can spawn a cofunction remotely, one should also admit spawning cofunctions locally as asynchronous activities. Once we add cofunctions to the language, we need a way to determine when cofunction invocations have quiesced. We plan to use a block structured `finish` construct, as does IBM's X10 programming language. Once multiple threads are allowed in images, one interesting question that arises is how to prioritize execution between threads. It would seem that the language model should offer some way to control the priorities of activities. Exploring these issues is a topic of future work.

References

[1] D. Bonachea. GASNet Specification, v1.1. Technical Report UCB/CSD-02-1207, U.C. Berkeley, 2002.
[2] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, 2003.
[3] Fortran J3 Committee. Fortran 2008 Working Draft, J3/09-007r1, March. 25, 2009. http://www.j3-fortran.org/doc/standing/links/007.pdf.
[4] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, Feb. 1988.
[5] 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.
[6] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 1.1. http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html, June 1995.
[7] R. W. Numrich and J. Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.
[8] J. Reid and R. W. Numrich. Co-arrays in the next Fortran standard. *Sci. Program.*, 15(1):9–26, 2007.
[9] A. Skjellum, N. E. Doss, and P. V. Bangalore. Writing libraries in MPI. In A. Skjellum and D. S. Reese, editors, *Proceedings of the Scalable Parallel Libraries Conference*, pages 166–173. IEEE Computer Society Press, October 1993.
[10] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The complete reference*. MIT Press, Cambridge, MA, 1996.
[11] J. Vuillemin. A data structure for manipulating priority queues. *Commun. ACM*, 21(4):309–315, 1978.