An Evaluation of Anticipated Extensions for Fortran Coarrays

Shiyao Ge, Deepak Eachempati, Dounia Khaldi, Barbara Chapman
Department of Computer Science
University of Houston, Houston,Texas, 77004
Email: {sge2, dreachem, dkhaldi, bchapman}@uh.edu

***Abstract*—A set of parallel features, broadly referred to as Fortran coarrays, was added to the Fortran 2008 standard. It is expected that several new parallel features, designed to complement or augment this feature set, will be added to the next revision of the standard. This includes statements for forming and changing between image teams, as well as statements for performing communication and synchronization with respect to image teams. In this paper, we describe an early implementation and evaluation of these anticipated features within the OpenUH compiler and its CAF runtime system. We demonstrate the utility of team-based barriers in comparison to the existing `sync images` statement for performing synchronization amongst a team of images. Techniques for hiding synchronization and incorporating locality-awareness with a collectives implementation based on 1-sided communication are described, and we present the impact of these optimizations for *allreduce* operations based on message length. Our results showed better performance for medium to large sized messages compared to the corresponding *allreduce* implementation using the Cray Fortran Coarrays implementation. Using the new teams and collectives features, we obtained 6.2% performance improvement compared to an original Fortran 2008 version of the CG benchmark from the NAS Parallel Benchmark Suite for class D, when using 1024 images.**

***Keywords*-Coarray Fortran; teams; PGAS; memory hierarchy; intra- and inter-node runtime; collective operations**

I. Introduction

Coarray Fortran (CAF), originally proposed as a small extension to Fortran 90/95 [16], was incorporated into the most recent version of the Fortran standard, Fortran 2008. It is part of a class of parallel languages following the Partitioned Global Address Space (PGAS) paradigm. In CAF, an *image* represents an executing process in an SPMD program with its own copy of data, some of which is accessible to other images via the coarray abstraction. Compared with existing parallel programming implementations, the Fortran 2008 coarrays model contains a relatively small set of features. To meet the demands of supporting more sophisticated operations and leveraging features on modern architectures, some new coarray language facilities are being developed by the WG5 working group for the next version of the Fortran language standard. Once finalized, this extended set of features will be published as a *Technical Specification (TS) 18508*.

The purpose of this paper is to describe an implementation of the new features being proposed for Fortran coarrays within the OpenUH CAF compiler, including techniques developed within its CAF runtime system for efficient support of collectives and barriers in the context of team-based execution. The full set of proposed features is, as of this writing, in the drafting stage [4]. We have focused in this work on the implementation of features for which the description has stabilized in various iterations of the drafting process, and therefore we expect these features to be part of the next Fortran standard with perhaps some minor syntactic alteration. Most of this effort, thus far, has been focused on two aspects of the TS – support for *teams* and support for team-based collectives and barriers.

Using teams, a CAF program has a mechanism for decomposing a problem into sub-problems that may be worked on concurrently by different process groups. This concept has already been introduced in many parallel programming APIs, such as MPI [17] with the use of communicators. In CAF, *team* constructs are used to divide an application region into loosely coupled team regions that are handled by different subsets of images. A common example would be to have a logical 2-dimensional grid of images partitioned into row and/or column teams, which may each independently perform collective operations. Another example would be applications consisting of multiple parallel tasks that are mostly independent of eachother. Rather than sequentially executing each task on all images, a more efficient utilization may be achieved by mapping them onto teams.

Teams are of special importance for PGAS models, since this concept enables the allocation of memory in a subset of images and not only all images. In this paper, we present our design and implementation of teams. We show how we redesigned our managed symmetric heap to accommodate memory allocations for distinct teams, describe how images collectively form teams and change from one team context to another along a teams hierarchy, and also how images reference other images based on team-relative image indices.

The contributions of this paper are:

- a description of an early implementation of additional parallel processing features, including teams, collectives, and barrier operations, which are complementary to the existing Fortran coarrays model and being developed for incorporation into the next revision of the Fortran standard;
- synchronization-hiding and locality-aware optimization

techniques for collective operations in the runtime;
- evaluation of enhanced coarray features using benchmarks to assess the usefulness of team-based synchronizations and collectives.

The paper is organized as follows. We describe Fortran 2008 coarrays and anticipated TS 18508 features in Section II. The design and implementation of image teams within the OpenUH compiler are presented in Section III. In Section IV, we describe the algorithms and techniques that we applied for team-based collectives and barriers. Experimental results using our microbenchmarks and the Conjugate Gradient benchmark from the NAS Parallel Benchmark Suite are discussed in Section V. We discuss related work in coarray implementations and optimization of collective operations in Section VI. Finally, we conclude and discuss future work in Section VII.

II. Background

In this section, we briefly describe the Fortran coarrays feature set in the most recent Fortran 2008 standard. Then, we discuss the major additions which are expected for the next standard, to be described in a forthcoming technical specification document.

A. Overview of Fortran 2008 Coarrays

CAF was initially a proposed extension to Fortran 95, and has recently been incorporated into the Fortran 2008 standard. There are a fixed number of executing copies of the program called images, and globally-accessible data are declared using an extended array type referred to as coarrays. Coarrays are data entities that are accessible to remote images and are declared with the codimension attribute specifier. In Fortran 2008, if a coarray is allocated on any image, there must be a corresponding coarray of the same name, dimension, and size allocated at every other image. Subscripts of coarrays are specified with square brackets and provide a clear, straightforward representation of remote access to another image's memory using 1-sided communication semantics. For example, the assignment statement *A(:)[k] = B(:)* signifies a remote *put* operation from a local source array B to a target destination array A at image k.

CAF also includes global (`sync all`) and pairwise barrier (`sync images`) statements and a memory fence (`sync memory`) as part of the language. Coarrays of `lock_type` may be used to acquire or release lock variables on a specified image, using the `lock` and `unlock` statements. Additionally, a critical section may be defined by demarcating a region of code with the `critical` and `end critical` statements.

B. Expected Additional Parallel Features in Fortran

As pointed out in [11], the set of features introduced in Fortran 2008 for writing coarray programs lacks several important features required for writing scalable parallel programs. A technical specification document, *TS 18508*, is in the drafting process by the WG5 Fortran standards working group [4], and its purpose is to address many of these limitations. It describes features for coordinating work more effectively among image subsets, for expressing common collective operations, for performing remote atomic updates, and for expressing point-to-point synchronization in a more natural manner through *events*.

The proposed support for teams provides a capability similar to the teams feature in CAF 2.0 [15]. The *initial team* contains all the images and executes at the beginning of the program. All images in a team may collectively form a new team with the `form team` statement. This statement will create a set of new sibling teams, such that all the images in the current team are uniquely assigned to one of these new teams. The operation is similar to the `mpi_comm_split` routine in MPI, or the `team_split` routine in CAF 2.0. An image has a separate *team variable* for each team of which it is a member.

An image may change its current team by executing a `change team` construct, which consists of the `change team` statement (with a team variable supplied as an argument), followed by a block of statements, and ending with the `end team` statement. The `change team` construct may also be nested, subject to the following constraints: (1) the team it is changing to must either be the initial team or a team formed by the currently executing team, and (2) all images in the current team must change to a new team. The image inquiry intrinsics `this_image()` and `num_images()` will return the image index and total number of images for the current team, respectively, though an optional team variable argument may also be supplied to return the image index or number of images for another team. Image selection via cosubscripts is relative to the current team by default, and there is additional syntax for allowing cosubscripts to select an image with respect to a specified team.

Teams may be used to improve an application's performance and memory utilization. Suppose there are a collection of parallel, coarse-grained tasks[1] in a particular application's workload with little or no inter-task dependencies, and each of these tasks has diminishing parallel efficiency when executed with an increasing number of images. Without teams, an application would have to execute each of these parallel tasks on all images in some sequential order. On the other hand, with the images partitioned into suitably sized image teams, these tasks may be mapped onto and executed by the teams. If this mapping is well-balanced, overall utilization is not decreased, while the aggregate parallel efficiency is improved. Regarding memory, using teams one can declare and allocate coarrays during the execution region

[1]By task, we mean generically some work together with the data it is operating on, as opposed to explicit, asynchronous tasks supported by other programming models.

of a `change team` block, and in this case the coarray will only be allocated across all images *in the same team*. Hence, coarrays may be allocated only for the images operating on it, thus utilizing more efficiently the available memory on each image.

One of the major omissions in the coarrays feature set provided in Fortran 2008 was collective operations. To partially address this, support for broadcasts and reductions are expected to be added in the next standard. For reductions, the user may use one of the predefined reduction subroutines – `co_sum`, `co_min`, and `co_max` – or a more generalized reduction subroutine – `co_reduce` – which accepts a user-defined function as an argument. The user-defined operator function passed into `co_reduce` may be defined to operate on derived types, enabling the implementation of operations that require some intermediate state to be tracked (e.g., to return the maximum Q out of P values from P images). All of the reduction subroutines may optionally specify a *result_image* argument, indicating that the reduction result need only be returned to one image (a *reduce* operation, rather than an *allreduce*). For all reduction subroutines, the operation is assumed to be commutative and associative. For broadcasts, the user may use the `co_broadcast` intrinsic subroutine. The advantage of using these intrinsic routines rather than collectives support from a library such as MPI is that the compiler can issue warnings or errors for incorrect usage. For each of these collective subroutines, there is no requirement that the arguments be coarrays, though an implementation may optimize for this case. Collective operations are assumed to be among all images in the current team, including barriers via the `sync all` statement. The `sync team` statement will be added for an image to perform a barrier synchronization with a specified team, rather than only the current team.

III. Supporting Teams in Coarray Fortran

We have implemented most of the additional features described in the Technical Specification draft in the OpenUH [3] compiler, as we describe in this section and Section [**?**]. Previously, we implemented support for Fortran coarrays in OpenUH in accordance with the Fortran 2008 specification [6] [5]. We depict the OpenUH Coarray Fortran implementation in Figure 1.

We added support into the Fortran front-end of OpenUH for parsing the `form team`, `change team`, `end team` and `sync team` constructs. We added the new type `team_type` to the type system of OpenUH and support for `get_team` and `team_number` intrinsics. We also extended the CAF intrinsics `this_image`, `num_images`, and `image_index` for teams. During the back-end compilation process in OpenUH, team-related constructs are lowered to subroutine calls which constitute the *libcaf* runtime library interface. In the runtime, we added a `team_type`

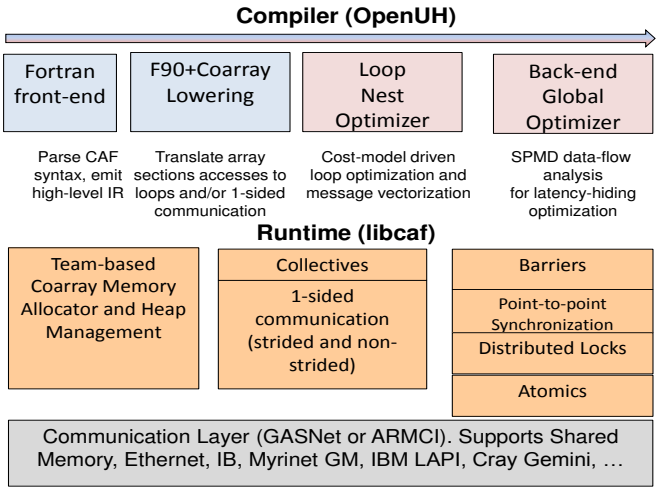


Figure 1: OpenUH Coarray Fortran team implementation

data structure for storing image-specific identification information, such as the mapping from a new index to the process identifier in the lower communication layer. The runtime also provides support for the team-related intrinsics `get_team` and `team_number`.

Before team support was added into our implementation, coarray allocation was globally symmetric across all images, with each coarray allocated at the same offset within a managed symmetric heap. With teams, however, this global symmetry is no longer necessary. According to the draft of the technical specification, symmetric data objects have the following features, which simplify the memory management for teams. First, whenever two images are in the same team, they have the same memory layout. Second, an image can only change to the initial team or teams formed within the current team. Third, when exiting a given team, all coarrays allocated within this team should be deallocated automatically. And fourth, if an image needs to refer to a coarray of another image located in a sibling team, the coarray should be allocated in their common ancestor team.

Team variables are opaque, first-class objects which may be used to query information for a specified team or change to a specified team. In our implementation, a team variable refers to an associated team data structure, depicted in Table I. During the formation of a team, the values for the fields of this data structure are computed and populated, including (1) the list of images that are on the same node, (2) the number of images within the same node, and (3) fields used to facilitate execution of collective operations. This information is used many times in the runtime by different parallel algorithms (for example, collectives and barriers as described in Section IV).

A. Memory Management

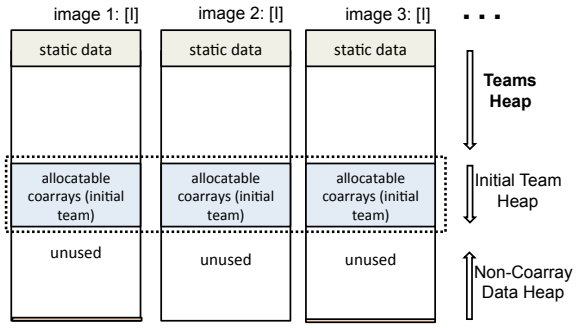
Before incorporating support for teams, we implemented the managed heap as follows. At the beginning of the

Table I: Team data structure

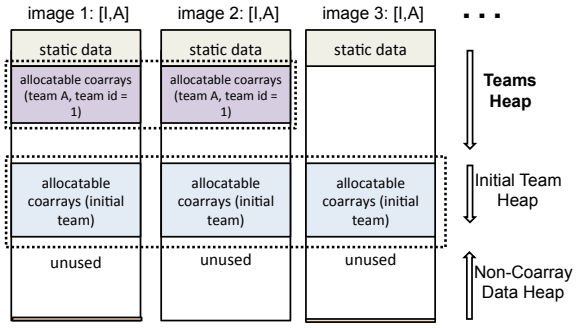
Field	Description
team_num	a team number or id, assigned during `form team` statement
this_image	image index for current image in team
num_images	number of images in team
intranode_set	ordered list of image indices in same compute node
leader_set	ordered list of image indices of node leaders in team
leaders_count	number of node leaders in team
image_index_map	maps image index to image index in inital team
sibling_maps	image index mapping for each sibling team created by same `form team` statement
bar_parity	parity variable for dissemination barrier
bar_sense	sense variable for dissemination barrier
intranode_bar_flags	direct shared pointers to intra-node barrier partners' flags
bar_rounds_info	partner information for inter-node in dissemination barrier rounds
coll_sync_flags	bcast, reduce, and allreduce sync flag
allreduce_bufid	selects between two allreduce buffers
reduce_bufid	selects between two reduce buffers
bcast_bufid	selects between two bcast buffers
allocations	a list of symmetric memory slots allocated for this team
parent	pointer to parent team structure

program, the images collectively allocate a pinned and registered memory segment which may be used for remote memory accesses. Static data which is allocated for the entire lifetime of the program is placed in a reserved space at the top of this segment. The rest of the segment is treated as a managed heap. Allocatable coarrays are symmetrically and synchronously allocated on all images from the top of this heap. We also allow non-symmetric allocations from the bottom of the heap. This serves a few different purposes. We can allocate temporary communication buffers from the bottom and avoid the cost of pinning and registering it. Additionally, even though Fortran 2008 requires all coarrays to be symmetric across all images, the coarrays may indirectly point to non-symmetric data. This is achieved by declaring the coarray to be of a derived data type with a pointer or allocatable component, for which the target data may be allocated independently of other images. This allocated data may then be remotely referenced using the coarray.

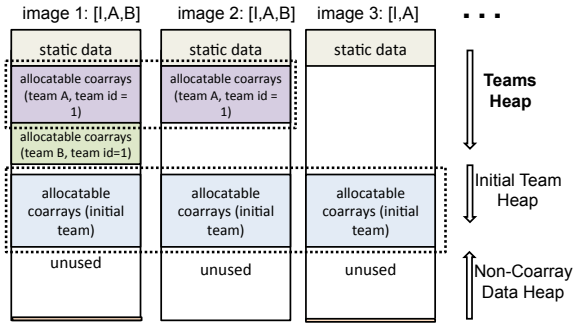
In order to support coarray allocation with respect to teams, we considered a few different approaches. The first approach was to reserve a fixed-size memory container for each team, within which any coarray allocations may be made. This could be achieved, for instance, through the use of *mspaces* in dlmalloc [13]. However, such an approach would require foreknowledge of how much space is required for each team, and in general we expected a considerable waste of allocated space using this approach. The second approach which we settled on is to instead reserve a fixed



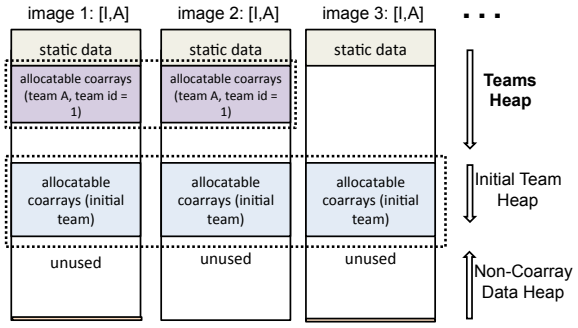
(a) Step 1: Allocate Coarray in initial team



(b) Step 2: Form new team A and allocate Coarray on image 1, 2



(c) Step 3: Form new team B and allocate Coarray on image 1



(d) Step 4: End team, exitting from team B

Figure 2: Evolving state of managed heap during team-relative symmetric allocations

size heap for all teams except the initial team, which we call the *teams heap*. This turns out to be sufficient, since a coarray may only be allocated for a non-initial team when that team is active and none of its descendant teams have currently allocated coarrays. The initial team is an exception, since at any time an image may execute a `change team` statement to change back to the initial team, and hence a separate heap for the initial team is still maintained.

```
type(TEAM_TYPE) :: I,A,B
integer :: id
integer,allocatable :: d1(:)[:], &
                       d2(:)[:], &
                       d3(:)[:]

I = get_team()
allocate(d1(10)[*])
id = (this_image()-1)/2+1
form team(id, A)
change team(A)
  allocate(d2(10)[*])
  if(team_number() .eq. 1) then
    form team(this_image(), B)
    change team(B)
      allocate(d3(10)[*])
    end team !exit team B
 end if
end team
```

Figure 3: Code depicting allocation of coarrays inside teams

Figure 2 illustrates how our managed heap evolves over the course of the example program shown in Figure 3. When a coarray is allocated while executing a `change team` block, corresponding allocations should occur on all other images in the *current* team, rather than for all images. Upon exiting the `change team` block, any allocations that had occurred within it are implicitly freed if they were not already freed by a `deallocate` statement. An image may only *change* to a team with the `change team` construct if it was formed by its current team with a `form team` statement or if it is the initial team. The latter scenario requires that the state of symmetric allocations belonging to the initial team should not be affected by allocations (not yet freed) belonging to a non-initial team. To support this, we reserve a fixed section of memory from the top of our managed heap for symmetric allocations by a non-initial team. We divide the list structure for memory allocations into two lists: one is for symmetric allocations by any non-initial team, and the other is for symmetric allocations by the initial team and all non-symmetric allocations. When changing to a new team, the *allocations* field in the team structure will be set to the current position in the non-initial team allocations list.

B. Forming and Changing Teams

The `form team` statement forms multiple teams by subdividing the set of images that are members of the current team. Each image in the current team must call the statement, specifying the number of the new team it will join and a team variable which it may use to refer to the newly formed team. A third optional argument may be specified to request a particular image index within the new team. It is otherwise implementation-defined how these image indices are assigned to team members; in our implementation, image indices are assigned to each image in the order of their indices in the current team.

Forming a new team entails a coordinated exchange of information from every image in the current team. Our implementation is currently as follows. All the images collectively perform an *allgather* operation to exchange the specified team numbers and (if given) image indices. Once this step completes, each image can determine the members (and their respective image indices) for each team formed. Based on this, each image can fill in the relevant information in the team data structure which it associates with the newly formed team. If a requested image index was specified with the `form team` statement, this can be directly assigned to the *this_image* field. Otherwise, the new image index is determined by sorting the set of image indices which specified the same team number. The *image_index_mapping* field is a pointer to an array which maps an image's image index in the new team to its image index in the initial team. The leader set contains the image indices for images in the team serving as designated leaders for their respective compute nodes. The intra-node set contains the image indices for all images in the team that share the same compute node. The leader set and intra-node set may be computed based on the runtime's determination of the process-to-node layout for the job. The *siblings* field, shown in Table I, is a pointer to an array of image maps for every other new team formed. This is useful because an image may also access a coarray belonging to a different team, using an extension to the normal image selector syntax (e.g., $a[i, team\_number = 2]$). While at present we precompute all image maps for the formed team and sibling teams during team formation, we note that there are caching techniques for significantly reducing the space requirements for image index maps [10] that we are considering to improve scalability.

During the team formation step, we also allocate various synchronization flags to be used for team-based barriers and collective operations. For barriers, we distinguish flags to be used for synchronization within a node via shared memory (available through *intranode_bar_flags*), versus flags used to synchronize between images on separate nodes (available through *bar_rounds_info*). These flags are also stored in the team data structure associated with each team, shown in Table I. Pointers for accessing a partner image's synchro-

nization flag at each round of the barrier are also precomputed and stored at this stage. Synchronization flags are also allocated and reserved for supported collective operations (specifically, *allreduce*, *reduce*, and *broadcast*) that may be executed by the new team. This is necessary since we implement collectives using 1-sided communication which is decoupled from synchronization. Since these collectives entail different communication structures, in order to allow for their execution to partially overlap we allocate a distinct set of synchronization flags for each type during team formation. More details on our support for team-based collectives and barriers are given in Section IV.

The `change team` statement is used to change the *current team* in which the encountering image is executing to a team referenced by a team variable argument. When an image executes this statement in our implementation, it will simply change an internal *current_team* pointer to the address of the team structure referenced by the team variable. Next, all images changing to the same team will synchronize via an implicit team barrier (note that the program should generally ensure that all or none of the images in a team reach the statement, though its possible for an image to check for stopped or failed images during its execution). When `end team` is encountered, the runtime will set the internal *current_team* to point to the parent of the current team. If leaving a team which has itself created child teams, then the team structures allocated for each of those child teams may be freed, and the corresponding team variable will be set to a `NIL` value to indicate that it is no longer associated with a team. If the team had allocated coarrays out of the symmetric heap, the associated slots describing these allocations (in the *allocations* field) are freed. Finally, all images in the team it returns to must synchronize via an implicit team barrier. Note that whenever an image switches to a different team, through the `change team` or `end team` statements, it will always synchronize will all images which are members of that team. This ensures that an image will never be executing in a team while other members are executing in a different team. We make use of this fact in the synchronization-avoidance optimizations we implemented for collectives.

IV. Supporting Team-based Collectives

We currently support *reduce*, *allreduce*, *broadcast*, and *allgather* collective operations. The *reduce* and *allreduce* support handles both pre-defined reduction operations (`sum`, `min`, and `max`) as well as user-defined reduction operations. The *allgather* support is used exclusively to facilitate formation of teams, and was implemented using Bruck's algorithm [2] with 1-sided communication. The *reduce*, *allreduce*, and *broadcast* implementations are used for the corresponding intrinsic subroutines – `co_reduce`, `co_sum`, `co_min`, `co_max`, and `co_broadcast`. We implemented the respective binomial tree algorithms for reduction and broadcast, and the recursive doubling algorithm for *allreduce*. These well-known algorithms, which we implemented using 1-sided communication, are described in [19]. Each of these algorithms complete in $\log P$ steps for P images in a team, where on each step pairs of images are communicating – either a write from one image to the other (*reduce* and *broadcast*), or independent writes from both images to the other image in the pair (*allreduce*).

These collectives require that all images in the current team participate. Special care is needed to ensure that the writes among the images are well-synchronized. We developed a number of techniques to make these operations fast, in particular to hide or eliminate synchronization costs among the communicating images, which we describe in this section.

A. Basic Scheme

Our collectives implementation makes use of a *collectives buffer space* – a fixed size, symmetric region which is reserved from the remote access memory segment. By default, the size is 4 MiB per image, but this may be adjusted through an environment variable. If this reserved buffer space is large enough, it will be used to carry out any of the data transfers required during the execution of a collective operation. Otherwise, all the images in the team will synchronously allocate a symmetric region of memory from the heap to serve as the buffer space for the execution of the operation. In order to carry out a *reduce*, *broadcast*, or *allreduce* operation, each image will reserve from its buffer space a *base buffer*. The base buffer is used to hold the result of each step in the collective operation. Additionally, for *reduce* and *allreduce* each image will reserve at least one work buffer. The work buffers are used to hold data communicated by a partner on each step, which will be merged into the base buffer using the specified reduction operation. Our binomial tree reduction and broadcast algorithms assume that the root image will be image 1 in the team. Therefore, we incur the cost of an additional communication to image 1 (for *broadcast*) or from image 1 (for *reduce*) when this is not the case. In the event that these operations are operating on large arrays which can not be accommodated within the buffer space available (determined by the image heap size), we can break up these arrays into chunks, and perform our algorithm on each of these chunks in sequence.

B. Multiple Work Buffers

For each step of the *reduce* and *allreduce* operations, intermediate results are written by some subset of images into the work buffer of their partner. Supposing each image reserved only a single work buffer, an image must first wait for a notification that its partner's work buffer is not being used before initiating this write. In order to reduce this synchronization cost, we can have each image reserve up to Q work buffers, where $1 \leq Q \leq \lceil \log P \rceil$. Images can

send out notifications that their work buffers are ready to be written into by their next Q partners on just the first step of every Q steps of the *reduce* or *allreduce* operation. In this way, the cost of the notifications can be amortized over the execution of Q steps. Our implementation will set Q to the largest value that can be accomodated within the buffer space, up to and including $\lceil \log P \rceil$. Alternatively, this parameter may be set explicitly through an environment variable.

C. Hide Startup Synchronization Cost

Even if we can partially hide the cost of $Q - 1$ out of Q notifications using the multiple work buffers strategy, we would still incur the synchronization penalty for the first step of the reduction operation, and the same would apply for broadcasts. This is because an image should not start writing into its partner's buffer space until it is guaranteed that the partner is not using this space (i.e. the partner is not still executing a prior collective operation which is using the space). If the images performed a synchronous heap allocation of its buffer space because there was not sufficient room in the pre-reserved collective buffer space, then all images can safely assume that the new space is not being used for a prior operation. If operating with the collectives buffer space, then some synchronization would be needed on the first step, which we refer to as the *startup* synchronization cost.

To hide this startup cost for the *reduce*, *allreduce*, and *broadcast* operations, we employ a double-buffering scheme. In this scheme, we partition our collectives buffer space into three pairs of buffer spaces, namely reduce buffer spaces (*reduce_bufspace1* and *reduce_bufspace2*), allreduce buffer spaces (*allreduce_bufspace1* and *allreduce_bufspace2*), and broadcast buffer spaces (*bcast_bufspace1* and *bcast_bufspace2*). If at any time more than one collective operation by a team is in progress, then we enforce the constraint that these operation-specific buffer spaces may be used only for their respective collective operation. Hence, these spaces are not exclusively reserved for their respective collective operations, but rather are prioritized for overlapped execution of those operations.

For an *allreduce* operation, our double-buffering scheme allows us to eliminate the need for any startup synchronization, so long as the allreduce buffer space is of sufficient size. If we consider the execution of 3 consecutive *allreduce* operations, the first operation may use *reduce_bufspace1* and the second operation may then use *reduce_bufspace2*. For *allreduce*, it is guaranteed that the second operation will not complete until all the images have completed the first operation. Consequentially, the third *allreduce* operation may freely use *reduce_bufspace1* without concern that the space may be still being used by a prior operation.

For the *reduce* and *broadcast* operations, our binomial tree algorithms still would require a startup synchronization, but this cost may be hidden effectively. Specifically, once an image completes the execution of either *reduce* or *broadcast* while operating on either buffer space 1 or 2, it will send out notifications indicating this buffer space is free to all images that will write to it on a subsequent *reduce* or *broadcast* operation. The cost of these notifications may therefore be hidden within the period between the operation's completion and the start of the next operation of the same type which will use the same buffer space.

D. Reducing Write Completion Synchronization

Because we use 1-sided communication for all data transfers during the execution of our collective operations, the receiver of the data must wait for some notification from the sender that the data transfer has completed. The image performing the write could perform a blocking *put* for the write, followed by a notification that the *put* has completed at the target. However, this would incur the additional cost of waiting for the completion of the first *put* and sending the completion notification back to the target. To reduce this additional overhead, we take advantage of the cases where the delivery of data into the target memory is guaranteed to be byte ordered. To the best of our knowledge, this is the case for all current hardware implementations of the InfiniBand standard, though on other systems using other interconnects that do not ensure this ordering (e.g., Cray Gemini), this optimization is not applicable. For this case, other approaches may be considered, such as utilizing GASNet long active messages to set a flag value after the payload transfer has completed.

For platforms where *inter-node* data delivery is known to be byte ordered, we provide the *use_canary* setting for our collective operations. With this setting, all buffers which are used, the base buffer and the work buffers, are initialized to hold only zero values. This is ensured by always zeroing out the buffers upon completion of a collective operation. Furthermore, we reserve an extra trailing byte for all the buffers. When performing an inter-node write operation during the execution of a collective, an image may then append a value of 1 to the write message which will occupy this last byte at the destination buffer. The receiver need only poll on this byte, waiting for it to become non-zero, and it can then be guaranteed that the entirety of the message was delivered.

E. Exploiting Shared Memory

The latency for inter-node remote memory accesses is typically significantly higher compared to intra-node memory accesses. Therefore, a reasonable strategy for collective operations that exhibit a fixed communication structure (which is the case our *reduce*, *allreduce*, and *broadcast* algorithms) is to restructure the communication in such a way that minimizes that required inter-node communication. This is especially important when dealing with collective

operations for a team of images, where member images may be distributed and fixed across the nodes in a non-uniform manner. We therefore developed 2-level algorithms for these collective operations which exploit the fact that the operations are presumed to be commutative and associative. Each compute node which has at least one image in the team has a designated leader image, and all the leaders in the team have a unique *leader index*.

When performing the *reduce* or *allreduce* operation, there are three phases. In the first phase, team members residing on the same compute node will reduce to the leader image. In the second phase, the leaders, will carry out either the *reduce* or *allreduce* operation among themselves. After the second phase, the first leader has the final result for *reduce* operation, and all leaders have the final result for the *allreduce* operation. In the third and final phase, for the *reduce* operation the first leader will write the final result to the root image, if it is not itself the root image. For the final phase of the *allreduce* operation, each leader image will broadcast its result to other images on its compute node. Depending on the particular topology of the compute node, this intra-node broadcast may be implemented using a binomial tree algorithm, or by simply having all the non-leaders on a node read the final result from the leader with the requisite synchronizations.

For the *broadcast* operation, the three phases are as follows. In the first phase, the source image will first write its data to the first leader image in the team. In the second phase, the leaders will collectively perform a binomial tree broadcast. In the final phase, each leader will broadcast its result to the non-leaders on its node.

F. Team Barriers

The `sync all` and `sync team` statements require that all images in a team, the current team for `sync all` and the specified team for `sync team`, wait for completion of any pending communication and then synchronize at a barrier. Moreover, if while executing the barrier an image is detected to have entered a termination state, then the other images must abort the barrier operation and will either enter error termination or return a `stat_stopped_image` status to the program.

To execute a barrier for all images in the initial team, the typical case when teams are not created by the program, we could simply use the barrier routines provided by the underlying runtime library (GASNet, ARMCI, or OpenSHMEM). However, these barriers would only permit the detection of stopped images prior to entering the barrier, but not during the execution of the barrier itself. For the more general case of executing a barrier among images in a non-initial team, we implemented a dissemination barrier using one-sided communication facilities provided by the underlying runtime. The dissemination barrier proceeds in $\log P$ steps for a team of P images, where on step k image i will send a notification to image $i + 2^{k-1} \bmod P$ and wait for a synchronization notification from image $i - 2^{k-1} \bmod P$ or a signal that some image in the team has stopped. If an image receives a notification from image $i - 2^{\log P - 1} \bmod P$ on the final step, it can be assured that all P images in the team have reached the barrier. Furthermore, if at least one image in the team terminates without reaching the barrier, then this information will propagate to all other images in the team executing the barrier before or at the final step.

We also enhanced our barrier implementation by taking advantage of node locality information provided by the underlying runtime layer, described in [12]. Within the runtime descriptor for a team there is a list of image indices for images that reside on the same node and a list of image indices for images in the team which are designated leaders of their respective nodes. Using this structure, we implemented a *2-level* team barrier algorithm as follows. When an image encounters the barrier, it will either notify the leader image of its node, or if it is itself a leader it will wait on notifications from the non-leaders. Once the leader image has collected notifications from all other images in the same node, it engages in a dissemination barrier with other leader images in the team, as described above. Finally, once the dissemination barrier completes the leaders will notify its non-leaders that all images in the team have encountered the barrier. Each non-leader image will atomically increment a shared counter residing on the leader image within the node to signal that it reached the barrier, and it will wait for a synchronization flag to be toggled by the leader to signal that the rest of the images have also reached the barrier.

V. Experimental Results

In this section, we describe an evaluation of the implementation and optimizations described in this paper, which we refer to here as UHCAF. Experimental results are from 3 benchmarks: (1) a team barrier microbenchmark used to assess the benefits of using a team-based barrier versus existing approaches available in Fortran 2008, (2) a reduction microbenchmark which contains an evaluation of the incremental performance gained by applying the optimization techniques described in the preceding section, and (3) the CG benchmark from NAS benchmark suite which is rewritten using new Coarray Fortran features discussed in this paper.

A. Experimental Setup

Stampede is a supercomputer at the Texas Advanced Computing Center (TACC). It uses Dell PowerEdge server nodes, each with two Intel Xeon E5 Sandy Bridge processors (16 total cores) and 32 GiB of main memory per node. Each node also contains a Xeon Phi coprocessor, but we did not use this in our experimentation. The PowerEdge nodes are connected through a Mellanox FDR InfiniBand network, arranged in a 2-level fat tree topology. We installed OpenUH

3.0.40, Rice CAF 2.0 (r4169), GASNet 1.22.4, and the latest GASNet 1.24.2 on Stampede for our evaluations. The MPI implementation we used was MVAPICH2, version 1.9a2.

Titan is a Cray XK7 supercomputer at the Oak Rdige Leadership Facility (OLCF). Each compute node on Titan contains a 16-core 2.2GHz AMD Opteron 6274 processor (Interlagos) and 32 GiB of main memory per node. Two compute nodes share a Gemini interconnect router, and the pair is networked through this router with other pairs in a 3D torus topology. Titan includes the Cray Compiling Environment (CCE) which includes full Fortran 2008 support, including Coarray Fortran. We used CCE 8.3.4. OpenUH 3.0.40 and GASNet 1.22.4 were installed on Titan for our experiments, where we compare the performance achieved with OpenUH to the performance using the Cray compiler.

B. Evaluation of Team Barriers

In Figure 4, we show timings for different use cases of team barriers. We arranged 4096 images to show 3 synchronization cases: 1) all participating images synchronize using `sync all`, 2) form teams and have images in each team synchronize using `sync team`, and 3) image subsets synchronize using `sync images`. Logically, the `sync images` statement with an image list consisting of all images in the same logical "team" will have the same effect as a `sync team` statement using a team variable representing a team consisting of the same images.

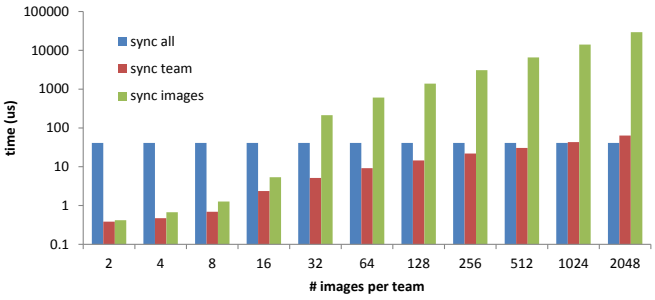


Figure 4: Barrier synchronization for groups of images (4096 total images), on Stampede.

The reader may notice that having all participating images execute `sync all` performs reasonably well until the number of images per team reduces past a certain threshold. This is because we utilized the barrier provided by GASNet to implement `sync all` for the initial team, and it happens to be well tuned for the InfiniBand interconnect used on Stampede. Before `sync team` was proposed, synchronization among a subset of images could be achieved alternatively using the `sync images` statement. The scalability for this statement, however, quickly became an issue as the participating images increase, since the semantics of this statement require that an image perform a point-to-point synchronization with each image in its specified image list. We observe here that `sync team` is a far more effective approach for synchronizing a subset of images compared to using `sync images`.

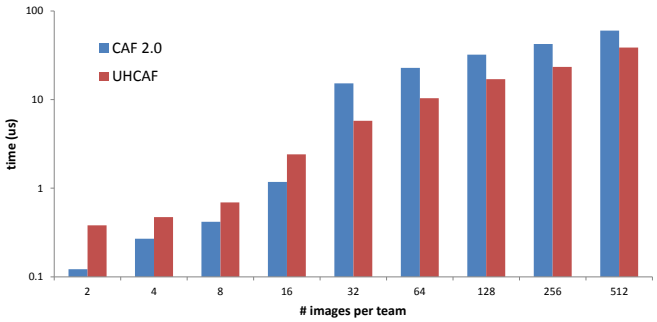
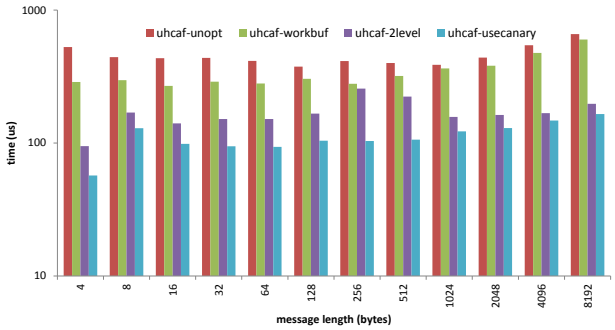


Figure 5: Comparison of team barrier between UHCAF and CAF 2.0 (1024 total images), on Stampede.

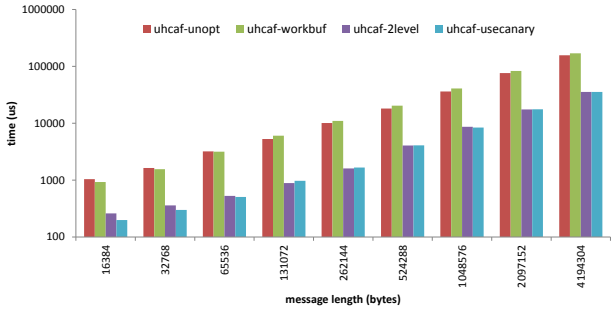
We also compared the performance of our team barrier implementation with the equivalent `team_barrier()` routine available in the Rice CAF 2.0 implementation, shown in Figure 5. For this comparison, we used the most recent GASNet version, 1.24.2 (all other experiments described in this section used GASnet 1.22.4). The result shows that the CAF 2.0 barrier implementation was more efficient when the team size was less than or equal to 16, where all images in a team reside within the same compute node. On the other hand, our barrier implementation was more efficient when each team spans multiple compute nodes. We attribute this result to the 2-level barrier algorithm we've implemented, while there is evidently some improvements to be made in our intra-node barrier implementation.

C. Evaluation for Reductions

In Figure 6, we show timings for the `co_sum` operation on Stampede with various optimizations incrementally applied. *uhcaf-baseline* refers to the basic scheme for our collectives implementation including the double-buffering technique for hiding startup synchronization costs. *uhcaf-workbuf* shows the performance when each image additionally uses $\log P$ work buffers. *uhcaf-2level* furthermore adds the shared memory support within each node. Finally, *uhaf-usecanary* incorporates our optimization for write completion synchronization, described above, which we verified as safe for the InfiniBand network on this system. We observe that adding the shared memory ("2-level") support to our implementation had the biggest impact, consistently across all message lengths. For smaller message lengths (less than 64K), the use of additional work buffers provided some improvement, but for larger message lengths the overhead of managing multiple work buffers offsets the reduction in synchronization. The *use_canary* setting for reducing write completion synchronization also provided benefits for smaller message sizes, but this advantage vanished when dealing with larger messages.

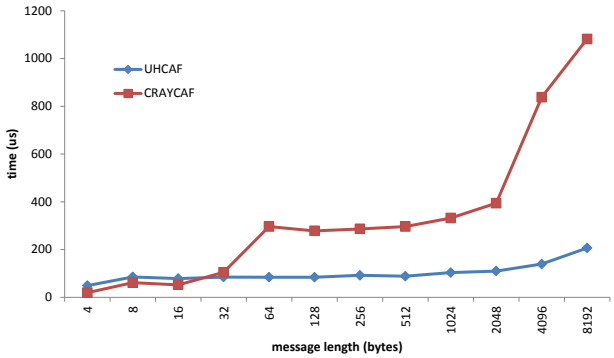


(a) small message lengths

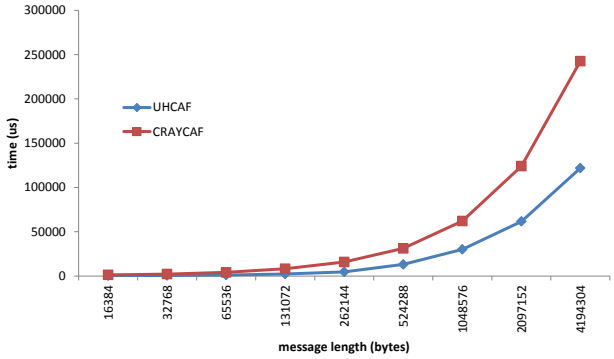


(b) large message lengths

Figure 6: co_sum (allreduce) performance on Stampede using 64 nodes, 1024 images



(a) small message lengths



(b) large message lengths

Figure 7: co_sum (allreduce) performance on Titan using 64 nodes, 1024 images

Figure 7 shows timings for the allreduce `co_sum` operation using 1024 images over 64 compute nodes on Titan (so, 16 images per node). We compare our implementation with the Cray compiler's CAF implementation. Cray's implementation is well-tuned for reductions with very small message lengths ($\leq$ 16 bytes), but for larger messages performs poorly relative to our implementation. The OpenUH implementation benefits here from the use of double-buffering to hide startup synchronization, multiple work buffers, and using shared memory for reduction within each compute node. On this platform, the optimization for write completion synchronization was not used, as Titan's Gemini interconnect does not guarantee byte ordering of writes to remote memory.

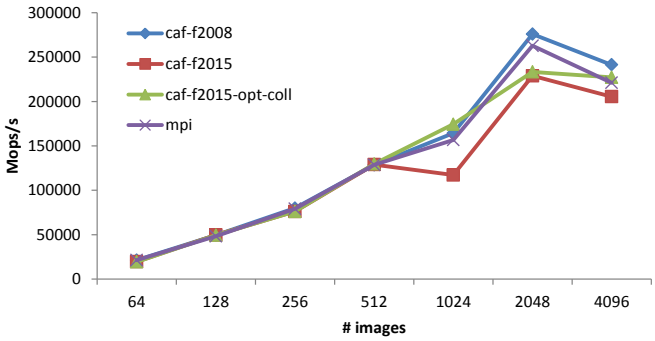


Figure 8: CG benchmark (class D) on Stampede, using 16 images per node

D. Using Team-based Collectives for CG

To assess the potential benefits of using the teams and collectives features, we updated our CAF implementation of the CG benchmark from the NAS Parallel Benchmarks (NPB) suite (available in [1]). The CG benchmark uses the *conjugate gradient* method to approximate the eigenvalue of a sparse, symmetric positive definite matrix, and makes use of unstructured matrix vector multiplication. We first ported this benchmark to use Fortran coarrays, adhering to the Fortran 2008 specification. For the extended version, we grouped the images into *row teams*, and during the execution of the conjugate gradient method we performed the sum reductions with respect to these teams. In this way, we were able to assess the utility of both the teams and reduction features that are expected to be included in Fortran 2015.

In Figure 8, we compare the results achieved with this new implementation on Stampede using class D problem size with our original Fortran 2008 version of the benchmark. We also show the results executing the original MPI version of CG using MVAPICH2. The baseline collectives implementation resulted in regressed performance relative to the Fortran 2008 version. Through synchronization-hiding and locality-aware optimizations of these collective operations,

as described in Section IV, we were able to improve on the baseline performance. However, we observed scalability issues even when using our optimized reductions when running with 2048 and 4096 images. We believe the issue originates from the need to add the `change team` and `end team` statements before and after calls to `co_sum` for performing the row-based reductions. The `end-team` statement, in particular, entails a barrier synchronization for all images in the initial team. One way around this would be to surround the entire iterative loop executed in *conj_grad* inside a team block, and utilize the new image selection syntax (e.g., $a(j)[i, team{=}init\_team]$) to perform the necessary communication and synchronization operations across teams (e.g. for the transpose operation). However, we have not yet implemented this image selection feature.

VI. Related Work

There are variety of existing Fortran coarray implementations which offer varying degrees of support for the CAF syntax in the current standard and proposed technical specification (TS). OpenCoarrays [7] is an open-source software project for developing, porting and tuning transport layers that support CAF compilers. This project provides near full support for Fortran 2008 Coarrays, and it also supports the collective operations defined in the TS. To our knowledge, support for teams has not been initiated yet in this project, and the collectives support are at present unoptimized. CAF 2.0 [15], an alternative Fortran extension for coarrays proposed by Rice University, included teams as first-class objects since its inception. It provided several features which are being, in part, adopted in the proposed TS, including teams, an expansive set of collectives, and events for point-to-point synchronization. CAF 2.0 specified a more MPI-like way to arrange subset of images. The collectives also including a team argument (just as MPI collective routines accept a communicator argument), which allow images to execute a collective operation as part of a specified team without changing the current team. While this feature can be quite useful from a programmer's perspective, it also requires implementations based on 1-sided communication to perform additional startup synchronization to ensure all images are executing the operation as part of the team.

Collective algorithms have been widely explored and are fundamental to distributed parallel programming. We implemented Bruck's algorithm using 1-sided communication in our implementation of `form team` to exchange image index information among images [2]. A number of algorithms and optimizations have been proposed, many of which are implemented in MPI, which we based some of our work on. These include the classical recursive-doubling algorithm for allreduce operations, described in [20], and the dissemination barrier, described in [9]. Since communication and synchronization are decoupled when relying on 1-sided remote memory access, the optimizations for collectives should focus on eliminating unneccessary synchronizations as much as possible, in addition to reducing message length or number of communications. In previous work for enabling RDMA in implementing MPI [8] [14] [18], the authors discussed several differences between classic MPI send/recv and RDMA. Similarly, we implemented several buffering and point-to-point synchronization schemes to enhance our 1-sided implementation of collectives.

VII. Conclusions and Future Work

In this paper, we presented the support of new Coarray Fortran features, namely teams and collective intrinsics, which are expected to be part of an upcoming Fortran technical specification for inclusion in the next Fortran standard. For teams, we detailed the design and implementation of symmetric data management, since the team construct relaxes the symmetric nature of coarrays found in Fortran 2008. For collectives, we have implemented a number of classical algorithms using 1-sided communication to carry out this work. We describe a number of optimization techniques applied to achieve better performance for the algorithms, specifically focusing on hiding synchronization costs and exploiting node locality. We implemented most of the constructs and intrinsics being drafted in the forthcoming TS, except image selector syntax and support for image failure reporting. In our evaluation, we assessed the potential advantages of using teams for programs exhibiting a divide-and-conquer parallel pattern – specifically, the performance benefit of `sync team` over an implementation that does not form teams and uses `sync all` or `sync images`. We also show how the optimizations we have applied for collectives helped to achieve better performance for the sum reduction operation. We concluded our evaluation by showing results for one of the NAS Benchmarks, CG, which we ported to assess the usefulness of the new features. Using teams and collectives, we obtained a 6.2% performance improvement compared to the original Fortran 2008 version when running on 1024 images on Stampede with the class D problem size. However, for a larger number of images, we found the synchronization costs required for executing the statements for changing teams degraded performance.

So far, we have concentrated on developing an early implementation and focused our optimization efforts on collectives when using 1-sided communication. We intend to improve the efficiency of our teams implementation in order to make it more scalable and memory-efficient. We will also explore opportunities at compile-time to optimize the performance for collectives, synchronization barriers, and implicit barrier synchronization when changing teams. Finally, we intend to soon complete our implementation of the expected CAF extensions and release this to the public.

Acknowledgments

This work was supported by TOTAL and access to the computing facilities used were provided through project allocations on Titan and Stampede from OLCF and TACC, respectively. We would like to thank the participants in the Fortran standards WG5 working group for their on-going work in defining new language features for Fortran coarrays. Special thanks are also due to Pierre Jouvelot for providing valuable feedback on this evaluation.

References

[1] CAF Test Suite. https://github.com/uhhpctools/caf-testsuite.

[2] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, Eli Upfal, and Derrick Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *Parallel and Distributed Systems, IEEE Transactions on*, 8(11):1143–1156, 1997.

[3] Barbara Chapman, Deepak Eachempati, and Oscar Hernandez. Experiences Developing the OpenUH Compiler and Runtime Infrastructure. *Int. J. Parallel Program.*, 41(6):825–854, December 2013.

[4] Fortran Standards Committee. TS 18508 Additional Parallel Features in Fortran (WG5/N2074). Technical report, August 2015.

[5] Deepak Eachempati, Hyoung Joon Jun, and Barbara Chapman. An open-source compiler and runtime implementation for Coarray Fortran. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 13:1–13:8, New York, NY, USA, 2010. ACM.

[6] Deepak Eachempati, Alan Richardson, Siddhartha Jana, Terrence Liao, Henri Calandra, and Barbara Chapman. A Coarray Fortran implementation to support data-intensive application development. *Cluster Computing*, 17(2):569–583, 2014.

[7] Alessandro Fanfarillo, Tobias Burnus, Valeria Cardellini, Salvatore Filippone, Dan Nagle, and Damian Rouson. Opencoarrays: open-source transport layers supporting coarray fortran compilers. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, page 4. ACM, 2014.

[8] Rinku Gupta, Pavan Balaji, Dhabaleswar K Panda, and Jarek Nieplocha. Efficient collective operations using remote memory operations on via-based clusters. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 9–pp. IEEE, 2003.

[9] Debra Hensgen, Raphael Finkel, and Udi Manber. Two Algorithms for Barrier Synchronization. *Int. J. Parallel Program.*, February 1988.

[10] Guohua Jin, John Mellor-Crummey, Laksono Adhianto, William N Scherer III, and Chaoran Yang. Implementation and performance evaluation of the hpc challenge benchmarks in coarray fortran 2.0. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1089–1100. IEEE, 2011.

[11] John Mellor-Crummey, Laksono Adhianto, William Scherer. CAF: A Critique of Co-array Features in Fortran 2008, Feb. 10. 2008 2008. available at: www.j3-fortran.org/doc/meeting/183/08-126.pdf (Feb. 10. 2008).

[12] Dounia Khaldi, Deepak. Eachempati, Shiyao Ge, Pierre Jouvelot, and Barbara Chapman. A team-based methodology of memory hierarchy-aware runtime support in coarray fortran (accepted). In *IEEE International Conference on Cluster Computing (IEEE CLUSTER)*, Sept 2015.

[13] Doug Lea. A memory allocator called doug leas malloc or dlmalloc for short. *Available Online: April*, 18, 2010.

[14] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K Panda. High performance rdma-based mpi implementation over infiniband. *International Journal of Parallel Programming*, 32(3):167–198, 2004.

[15] John Mellor-Crummey, Laksono Adhianto, William N. Scherer, and Guohua Jin. A New Vision for Coarray Fortran. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, PGAS '09, pages 5:1–5:9, New York, NY, USA, 2009. ACM.

[16] Robert W Numrich and John Reid. Co-array fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM, 1998.

[17] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.

[18] Hari Subramoni, Ammar Ahmad Awan, Khaled Hamidouche, Dmitry Pekurovsky, Akshay Venkatesh, Sourav Chakraborty, Karen Tomko, and Dhabaleswar K Panda. Designing non-blocking personalized collectives with near perfect overlap for rdma-enabled clusters. In *High Performance Computing*, pages 434–453. Springer, 2015.

[19] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[20] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of Collective Communication Operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.