Final Report

September 2011

Center for Programming Models for Scalable Parallel Computing*

Rice University Subproject

Cooperative Agreement No. DE-FC02-06ER25754

John Mellor-Crummey

Principal Investigator at Rice University

Department of Computer Science, MS 132
Rice University
P.O. Box 1892
Houston, TX 77251-1892
Voice: 713-348-5179
FAX: 713-348-5930
Email: johnmc@cs.rice.edu

*Multi-institutional center Project Director: Rusty Lusk, ANL, lusk@mcs.anl.gov

Contents

1 Introduction

As part of the Center for Programming Models for Scalable Parallel Computing, Rice University collaborated with project partners in the design, development and deployment of language, compiler, and runtime support for parallel programming models to support application development for the "leadership-class" computer systems at DOE national laboratories.

A principal goal of work at Rice University was refining the Coarray Fortran (CAF) programming language, compiler and runtime so that it is expressive (natural for expressing a broad spectrum of algorithms, constructing efficient parallel data structures, and supporting both static and dynamic strategies for decomposing work); productive (programs are as easy to write as possible); high performance (across the spectrum of computing platforms ranging from commodity clusters to leadership-class systems); scalable to leadership-class computer systems with tens to hundreds of thousands of processors; portable; and interoperable with other programming models and program development tools. A secondary goal of this effort was to develop new compiler technology that support higher-level "global view" parallel programming models.

Over the term of the award, the work at Rice University proposed to focus on the following themes: refine language-based parallel programming models for emerging platforms; refine language-based programming models for higher performance and better expressiveness; investigate compiler technology for global view parallel programs; explore interoperability of parallel programming models; and investigate integration with programming environments and tools. Rice was to design, develop, and deploy open-source software that embodies the results of this research.

2 Summary Description of the Research Performed

Work over the course of this project has focused on the design, implementation, and evaluation of a second-generation version of Coarray Fortran. Research and development efforts of the project have focused on the CAF 2.0 language, compiler, runtime system, and supporting infrastructure. This work has involved working with the teams that provide infrastructure for CAF that we rely on, implementing new language and runtime features, producing an open source compiler that enabled us to evaluate our ideas, and evaluating our design and implementation through the use of benchmarks.

To provide context and motivation for our work, in Section 2.1, we describe the state of Coarray Fortran as this project began. Section 2.2 describes the emergence of a problematic effort to add coarrays to the Fortran Standard. Section 2.3 describes our new Coarray Fortran 2.0 language extensions; Section 2.4 describes our CAF 2.0 compiler. Section 2.6 explores the expressiveness of CAF 2.0 by using it to implement the HPC Challenge benchmarks. Section 2.7 describes the award-winning productivity we achieved with CAF 2.0 versions of the HPC Challenge benchmarks.

2.1 Deep Background: Coarray Fortran's Original Design

In 1998, Numrich and Reid proposed a small set of extensions to Fortran 95 to support parallel programming that they dubbed Coarray Fortran (CAF) [20]. 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 evaluating `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.

2.2 Motivation: Flawed Coarray Extensions in the Emerging Fortran Standard

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. Our detailed critique [17] of the coarray extensions proposed for Fortran 2008 and identified several shortcomings in emerging coarray extensions that limit their expressiveness:

- There is no support for processor subsets; for instance, coarrays must be allocated over all images.
- The coarray extensions lack any notion of global pointers, which are essential for creating and manipulating any kind of linked data structure.
- Reliance on named critical sections for mutual exclusion hinders scalable parallelism by associating mutual exclusion with code regions rather than data objects.
- 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.
- There is no support for hiding communication latency.

2.3 Coarray Fortran 2.0

The aforementioned shortcomings caused us to rethink the CAF model. Our aim was to develop an expressive set of parallel extensions for Fortran that map well onto parallel systems of all sizes, ranging from multicore nodes to petascale platforms. The product of our efforts was a set of language extensions to Fortran that we call Coarray Fortran 2.0 (CAF 2.0).

Our redesign of coarray extensions for Fortran has been largely driven by practical experiences gained while developing applications with it and analyzing their performance. Early application development focused on the HPC Challenge (HPCC) benchmark suite,[1] a well-recognized litmus

[1]http://www.hpcchallenge.org/

test for parallel programming models and languages in the HPC space. Subsequent effort has targeted DOE mission-critical codes including LANL's Parallel Ocean Program component of the Community Earth System Model, Sandia's S3D combustion code, and Heat — a model for LANL's SAGE code, which performs cell-by-cell adaptive mesh refinement.

Our design of new coarray extensions for Fortran focused on three core tenets: orthogonality, expressiveness, and simplicity. In a nutshell, CAF 2.0 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, collective communication, asynchronous operations for latency hiding, function shipping for latency avoidance, copointers to remote memory, and a memory model that enables one to trade ease of use for performance. We explore key features of our new CAF 2.0 extensions in the subsections that follow. Most of these ideas are inspired by features in MPI [26] and Unified Parallel C [10]. Here, we describe their realization as a cohesive whole to support parallelism in Fortran.

Process Subsets (Teams) Processor subsets is a useful abstraction for decomposing work in a parallel application. Processor subsets can be used in coupled applications (*e.g.*, *ocean* and *atmosphere* subsets in a climate application) as well as within dense matrix numerical computation such as matrix decomposition and solver for linear equations (*e.g.*, finding pivot element and exchange rows within *column* subsets and broadcast factored panel across *row* subsets). Earlier 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 Coarray Fortran 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 $r \in \{0..\texttt{team\_size(t) - 1}\}$, rather than an absolute image ID. As identified by Skjellum [25], 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 existing teams by using the collective`team_split(existing_team, color, key, new_team)`. The split operation was inspired by the functionality of MPI's `MPI_Comm_split` [18]. 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 subteam. 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, 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 team constructed with `team_split` can be dismissed with a call to `team_free` (similar to MPI's `MPI_Comm_free`) after it is no longer needed. When a team is freed, its allocated resources are released and the team no longer exists.

Data allocation Both Numrich and Reid's original CAF and Fortran 2008 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, known as `team_default`, is used. When a CAF program is launched, `team_default` is set to `team_world`. A `with team` statement (inspired by the `with` statement in PASCAL) is a block structured construct for setting the default team, `team_default` within its scope. Unlike PASCAL's `with`, CAF 2.0's `with team` has dynamic scope, meaning that its binding for the default team applies not only to the code lexically enclosed in the `with team` block, but also to any code called from within the block. We use LIFO semantics for dynamically nested `with team` statements. 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.

Topologies Fortran 2008 and earlier flavors of CAF only provide 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.

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. The programmer may either specify an ordered set of processor images to map onto the topology, or use an overloaded version of the function that asks the CAF runtime to bind the topology with a good mapping to the underlying processor fabric. Finally, `topology_get` extracts the topology associated with a team, or returns an error if there is none.

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. Note that not every image needs to specify every edge connection; it is sufficient to declare the edges attached to the image's node and have the CAF runtime construct the full topology from this distributed information.

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 $\mathtt{k}^{th}$ 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 parentheses 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`.

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 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 $(\mathtt{d}_1, \mathtt{d}_2, \mathtt{d}_3, ..., \mathtt{d}_\mathtt{n})$ where one would otherwise specify an image rank, e.g. `my_data(3)[(x+1, y+1)@team_grid]`. As with graph topologies, if the team is implicit, one may omit the tuple's parentheses; 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 placing the offsets in parentheses and prepending a `+` sign: `foo[+(3,-4)]` specifies an offset of `(+3, -4)` from the current image's position in a 2D Cartesian topology. Similarly, `foo[+(-1),0]` is the first column of the previous row in a 2D Cartesian topology; note that in this example, the first subscript is relative and the second is absolute.

Copointers CAF 2.0 adds global pointers to the Fortran language in support of irregular data decompositions, distributed linked data structures, and parallel model coupling. The definition and use of these new "copointers" is as similar as possible to ordinary Fortran pointers: they are declared with new attributes analogous to 'pointer' and 'target', manipulated with the existing `=>` pointer assignment statement, and inspected with the existing pointer intrinsics. Accessing data via copointers is as similar as possible to existing coarray accesses, with implicit access to the local image and explicit access to remote images using a square-bracket notation. CAF 2.0's copointers may point to values of any type, including coarrays; we believe that copointers to coarrays will be especially valuable for parallel model coupling in systems like the Community Earth System Model. Copointers can be implemented easily and efficiently in existing CAF compilers; as of this writing, copointers are not yet fully implemented in our CAF 2.0 prototype. A detailed note about our design for copointers was submitted to the Fortran Standards Committee for consideration and is publicly available as ftp://ftp.nag.co.uk/sc22wg5/N1851-N1900/N1856.txt from Working Group 5's web server.

Mutual exclusion Based in part on our feedback [17], locks were added to Fortran 2008 to support mutual exclusion. In CAF 2.0, we further support deadlock-free multi-lock synchronization by allowing the programmer to transparently acquire a set of locks as a single logical operation.

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

Table 1: Eventset API for manipulating events in the set.

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

Table 2: Eventset API for manipulating events.

CAF 2.0 provides three language constructs for mutual exclusion.

1. **Lock.** This is the standard mutual exclusion state variable; `lock_acquire` and `lock_release` 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.

Creating a lock or lockset is not a collective operation; neither is acquiring a lock, lockset, or critical section.

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 an available increment occurs (a `wait` operation). Our event implementation is thus essentially a user-mode local-spin implementation of a counting semaphore. 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.

Eventsets Because we expect to use events as the basis of our asynchronous point-to-point and collective communication operations, it is important that they be sufficiently flexible and expressive. In particular, we need to support multi-event manipulation functionality akin to the capabilities of the Berkeley sockets `select` statement [27] and the MPI `WAITANY` and `WAITSOME` functions [26] that await completion of asynchronous send or receive operations.

Logically, an eventset is a set of ordered tuples $\langle$`event`, `count`$\rangle$, where `count` is the number of times that `event` has been observed to trigger within an eventset API call, and is used for fairness purposes as detailed below. Table 1 details the API for initializing and destroying, and adding and removing elements from an event set. The parameter supplied to `eventset_init` receives a handle to the newly initialized eventset; this handle is then used as an input for the rest of the eventset API statements. It is an error to use an eventset handle after calling `eventset_destroy`: The eventset is no longer initialized.

Table 2 presents the portion of the eventset API that deals with notifying or waiting on sets of events. The statements `eventset_waitall` and `eventset_notifyall` wait for and signal, respectively, every event associated with the set. The `eventset_waitany` statement checks each event associated with the list to see if that event has triggered. When it finds one, it increments the trigger count associated with the event and resorts the list of tuples so as to avoid starvation even in the case where a single node is triggered with high frequency. Returning the ID of the triggered event allows the caller to know which event was triggered and react appropriately. `eventset_waitany_fair` behaves similarly to `eventset_waitany`; however, only those events with the fewest trigger counts are considered. This is useful in cases where at each stage of an algorithm, it is necessary to process each of several asynchronous events exactly once. Without this, the same effect could be obtained by maintaining a pair of eventsets, *current* and *next*, and explicitly migrating events from *current* to *next* as they come in and are processed. Once *current* is empty, swap the sets. Here, a small addition to the API dramatically increases programmer convenience and productivity for this case.

Collective Communication Collective subroutines are not new in Coarray Fortran; 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_max` and `co_product`. However, the emerging Fortran 2008 standard does not include these features even though collective operations are widely used in parallel applications. It is widely known that 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 some collective statements for Coarray Fortran 2.0 as shown in Table 3. These statements are mainly inspired by MPI's collectives based on two-sided synchronous communication. In the next section, we discuss asynchronous versions of these collectives.

Most collective statements require a local data variable source (`var_src`), a target variable (`var_dest`) and optionally a team where all image members will participate. If the team is not explicitly specified, then the current default team specified in an enclosing `with` statement or `team_default` will be used. Unlike the proposed collective statements by Reid and Numrich [22], all collectives in Coarray Fortran 2.0 do not require coarrays as its input/output data. As shown in Table 3, both `var_src` and `var_dest` variable can be local variables. For `team_broadcast`, the first argument (`var`) is a local variable that acts as the source variable for the root image, and as the target variable for other images.

When designing Coarray Fortran 2.0, we recognized that there are two types of collective reduction operations: those that are replication oblivious, where a value can be processed more than once by a reduction without changing the result, and those that are not. Some examples of replication oblivious operators include *min*, *max*, *and*, and *or*. We believe that it is worth identifying replication oblivious operators because reductions using them can be implemented efficiently by using a special communication pattern. The final optional `ro` boolean argument to a reduction operation indicates whether the reduction operator is replication oblivious.

In addition to the predefined collective operators shown in Table 3, Coarray Fortran 2.0 also supports user-defined operators for reductions. Instead of supplying a predefined operator to a reduce, a user can specify a subroutine that takes three arguments: two read-only inputs and the output. For Coarray Fortran 2.0, we introduce `team_sort` to sort arrays (whether they are coarrays or not) within a team. This operation requires a user-defined comparison function.

Statement	Description	Syntax
`team_broadcast`	broadcasts a data from an image to all images in a team	`team_broadcast(var, root_rank [, team ])`
`team_gather`	collects individual data from each image in a team at one image	`team_gather(var_src, var_dest, root_rank [, team ])`
`team_allgather`	gathers data from all images and distribute it to all images	`team_allgather(var_src, var_dest [, team])`
`team_reduce`	reduces data, the result is stored to an image of the team	`team_reduce(var_src, var_dest, root_rank, operator [, team] [, ro])`
`team_allreduce`	reduces data, the result is stored to all images of the team	`team_allreduce(var_src, var_dest, operator [, team] [, ro])`
`team_alltoall`	personalized all-to-all communication	`team_alltoall(var_src, var_dest, elt_size, num_elts`
`team_scan`	performs partial reduction (scan), each image store the result of reduction from its neighbor	`team_scan(var_src, var_dest [, team])`
`team_scatter`	distributes individual data from an image to each image in a team	`team_scatter(var_src, var_dest, root_rank [, team])`
`team_shift`	moves data from another image at an offset within a team	`team_shift(var_src, var_dest, image_offset [, team])`
`team_sort`	sorts arrays of the same size and type within a team	`team_sort(var_src, var_dest, comparison_function [, team])`

For most statements:

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

Table 3: Collective statements supported in Coarray Fortran 2.0

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

Alternatively, one can use the *implicit* model. Here, rather than signaling an event when the operation is complete, we provide synchronization to programmers via a dynamically scoped, nestable `finish` block. All asynchronous operations within a `finish` scope are guaranteed to be finished before exiting the `finish` block. We note that this concept of finish blocks is from the X10 programming language [24]. Each `finish` block is associated with a specific `team`, which may be omitted to have the runtime supply `TEAM_DEFAULT`. Note that only implicit operations are guaranteed to be complete at the close of a `finish` block; explicit operations are allowed to *escape*.

Predicated asynchronous copy On large-scale parallel systems, hiding communication latency is essential if a program is to achieve high performance. In studies with the HPC Challenge RandomAccess and FFT benchmarks, we quickly recognized the need for asynchronous data copies to overlap communication with computation. In the benchmark codes, we initially identified the `copy_async`(var_dest, var_src *[, ev_dw] [, ev_rdy] [, ev_sr]*)

var_dest	=	a coarray reference target
var_src	=	a coarray reference source
ev_dw	=	an optional event indicating that the write to var_dest is complete
ev_rdy	=	an optional event indicating that the copy can proceed
ev_sr	=	an optional event indicating that the read of var_src is complete

Figure 1: Asynchronous copy statement in CAF 2.0.

need to overlap computation with streaming writes of remote data. In particular, we wanted to issue a non-blocking PUT to remote data and notify the consumer awaiting the data when the PUT is complete; while this communication and synchronization are in flight, we want to continue with local computational work. Later, we identified a similar need to overlap a GET communication with computation and determine when the GET completes. We realized that under many circumstances a compiler might not be able to determine that it is safe to transform a GET or PUT into a non-blocking form and overlap it with computation. In particular, it can be very difficult to determine that a GET or PUT could be overlapped with a procedure call without changing a program's semantics in the case when code is being separately compiled. We realized that programmers know what communication can and should be overlapped with computation, and what CAF needs is a suitable language construct to make it possible to express such asynchronous communication.

As Coarray Fortran was originally conceived by Numrich and Reid, there were no language constructs that enable users to hide communication latency. In addition, their design included implicit memory fences at subroutine boundaries to avoid having a GET or PUT operation, in flight at a procedure call, cause data races with accesses by a callee to the target coarray [20]. A challenge was to create a design for adding asynchronous copies to CAF that enables application programmers to: (1) express that reading or writing of remote coarray data may be overlapped with computation; (2) make it possible to determine when such asynchronous operations are complete; (3) have the language constructs be sufficiently general to allow an application to await completion of pending asynchronous operations in any order; (4) make the completion of asynchronous operations orthogonal to program scopes, i.e. an application need not await completion of asynchronous operations within the same routine in which they are issued; completion may be requested at any point in the future by any routine whatsoever; and (5) provide a syntactic construct that is easy to use.

To satisfy these criteria, we designed the asynchronous copy primitive shown in Figure 1. In our design, `copy_async` is a statement rather than a subroutine in CAF 2.0. Both the source and destination of the copy must be coarray references. The source and destination may be scalar values, whole arrays, or array sections. Either may refer to local or remote coarray data. The events `ev_dw`, `ev_rdy`, and `ev_sr` respectively indicate that the write to destination is complete, that the source data is ready, and that the source data has been read (and may be safely updated concurrent with the `copy_async` operation). In the implicit asynchrony model, `ev_dw` is not specified and a subsequent `end finish` statement blocks until the write is complete.

The `copy_async` primitive is quite expressive. It can be used for local-to-local, remote-to-local, local-to-remote, or remote-to-remote copies. A local-to-local `copy_async` could be used to hide the latency of a local copy operation by having it executed asynchronously by another core or a DMA engine. A typical use of a remote-to-local copy would be for an asynchronous prefetch. If one specifies the optional `ev_rdy` event, the copy will not execute until `ev_rdy` is notified; this enables an application developer to specify a predicated prefetch that will not begin execution until the source data is available. A typical use of a local-to-remote `copy_async` is to export to a remote node

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

Argument descriptions:

`typedef::var_src`	local source variable	`team::team`	process subset (or default team)
`typedef::var_dest[*]`	target Coarray Fortran variable	`event::event`	event variable (if explicit asynchrony)
`integer::root`	the rank of the root image		

Table 4: Asynchronous collective operations in Coarray Fortran 2.0.

values that have just been computed, such as depositing boundary layer data into a ghost region on a remote processor. While `copy_async` supports remote-to-remote copies for completeness, we don't have a compelling case for this use at present.

Asynchronous Collective Operations When designing asynchronous collectives for a language with one-sided communication, one may ask why not to use a one-sided design. A two-sided design provides us two benefits. First, it enables each processor to have explicit control of how many collective operations may be pending on that processor at a time. Second, each processor has the flexibility to control buffer allocation and specify where incoming results should be placed. Table 4 shows the collective operations we are adding to Coarray Fortran 2.0.

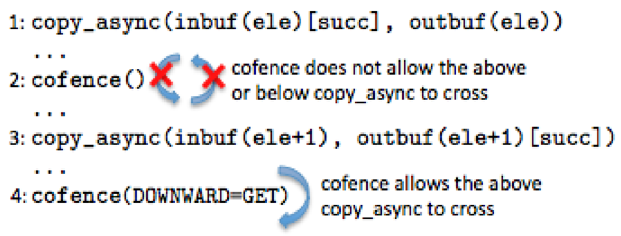
Function Shipping *Function Shipping* is another kind of asynchronous operation we newly integrated into CAF 2.0. It helps users avoid exposed communication latency by co-locating computation with remote data. One may argue this is redundant with transferring data to computation place, however, in practice moving data towards computation is difficult in certain circumstances. A simple example in this case is inserting an entry into a remote queue. Without function shipping, one needs to remotely acquire a lock, fetch the pointer location back, send the entry over to update the pointer, then release the acquired lock. This approach suffers from the overhead of four round-trips of communication, which makes it very undesirable.

Function Shipping makes this can be expressed fluently in CAF 2.0, by enabling programs to send computation along with necessary arguments to remote process for execution. As shown in Figure 2, replacing the Fortran keyword `call` with `spawn` causes the function to be executed on remote process image specified within the ending square bracket pair. Function shipping is made a

```
event :: ev
spawn(ev) foo(table(i,j)[p], n)[p]
call event_wait(ev)

finish (a_team)
  spawn foo(table(i,j)[p], n)[p]
  ...
end finish
```

Figure 2: Explicit and implicit model examples of CAF 2.0 function shipping



```
1: copy_async(inbuf(ele)[succ], outbuf(ele))
   ...
2: cofence()
   ...
3: copy_async(inbuf(ele+1), outbuf(ele+1)[succ])
   ...
4: cofence(DOWNWARD=GET)
```

cofence does not allow the above or below copy_async to cross
cofence allows the above copy_async to cross

Figure 3: `Cofence` and `copy_async`

asynchronous call by default in CAF 2.0. Like other asynchronous operations, its completion can be controlled explicitly by event variable provided after the `spawn` keyword, or managed implicitly by *finish* or *cofence* constructs when event variable is omitted.

Arguments passed into a shipped function are treated differently from normal function arguments. For coarray arguments, they are handled accordingly based on the dummy argument declaration within that function. Arguments that declared as non-coarray in dummy arguments are dereferenced on the call site, and copied to remote process, hence they act like implicitly marked with `VALUE` attributes - modifications on them will not be reflected back to the calling process. Coarray dummy argument give the function ability to access that array on any process image. If arguments are declared as copointers, image information from caller will be preserved as default image for that copointer, thus references followed by empty square brackets can access the array located on default image.

Shipped functions are usually executed on a different process than its origin. Thus, the context that shipped functions live is the one on the target process it is spawned to, so that they can access global data local to that process, even if the global data is not shared across images. We believe the change of execution context is necessary in making function shipping fully expressive. Manipulating many distributed data structures such as distributed hash tables, lists and graphs require shipped functions to have ability to operate on global data on target process.

Cofence *Cofence* construct allows programmers to control the local completion of put, get, and implicitly synchronized asynchronous operations. Cofence takes two optional arguments: first argument specifies which category of implicit asynchronous operations (i.e. `put/get`) to allow downwards, and second argument specifies which category of implicit asynchronous operations to allow upwards. Depending upon the argument value passed, the cofence allows puts, gets, or both to pass across the *cofence* in the specified direction.

Managing asynchrony with Cofence An asynchronous copy operation performing a `put` operation is locally complete at a process `p` after `p` initiates the `put` operation, and the buffer containing the data being copied to another process, is free to be modified by `p`. An asynchronous copy operation performing a `get` operation is locally complete at a process `p` after the buffer into which the data is brought into from another process, is ready to be consumed by process `p`. Figure 3 shows an example of using `cofence`, and `copy_async` constructs together. The asynchronous copy operation in line 1 shown in Figure 3 copies an element present in the coarray `outbuf` namely `outbuf(ele)` on the calling processor to the coarray `inbuf` on the `succ` processor. The operation is locally complete at the calling processor after it returns from initiating a one-sided communication call to `put` the element `(outbuf(ele))` on the `succ` processor. The point to note is that this completion does not include whether the `put` is complete on the `succ` processor i.e. `inbuf(ele)` is available for consumption by `succ` processor. The `cofence` (at line 2) takes the default argument (which stands for not allowing any `put/get` operation across) and hence does not allow the put operation to cross downwards (maybe it would have been advantages for the compiler/runtime from a performance perspective to move the copy operation downwards to a later point). The `cofence` (at line 4), however, allows the `copy_async` operation (a `get` at line 3) to pass downwards.

2.4 CAF 2.0 Compiler

When we began the project, we had an existing prototype compiler for Coarray Fortran that we had previously built using the Open64 compiler infrastructure. Our original intent was to explore some modest feature enhancements and harden this compiler to support experimentation and broader use. As the project unfolded, it became clear to us that the original design for Coarray Fortran needed significantly more refinement than we initially anticipated. As a result, we needed to change the compiler infrastructure we used for this work. The Open64 compiler infrastructure we were using was not readily modifiable to accommodate the significant changes to the language that we determined were necessary.

To support our redesign of the Coarray Fortran language extensions, we re-focused our implementation efforts to use LLNL's ROSE compiler infrastructure. ROSE was much more malleable and would make it easier to add new language features and syntax. Using ROSE, we constructed an open-source, source-to-source compiler for CAF 2.0. This compiler accepts Fortran with CAF 2.0 extensions and generates Fortran code in which the CAF 2.0 features have been lowered to Fortran. The translated programs make calls to a CAF 2.0 runtime library and support code that is generated based on the input program.

The CAF 2.0 compiler parses Fortran programs decorated with CAF 2.0 extensions. The CAF 2.0 program is translated into the ROSE intermediate form. CAF 2.0 constructs are then lowered by the CAF 2.0 compiler. In the course of lowering coarray variable declarations and uses, the CAF 2.0 compiler generates auxiliary modules that correspond to the coarray type $\times$ the dimensionality of the coarray. These helper modules are necessary to support Fortran's name equivalence model for types. Local coarray accesses are translated into conventional Fortran variable references; remote coarray accesses are translated into calls to the CAF 2.0 runtime system, which will interact with one or more remote nodes as necessary.

Most of the project's effort on the CAF 2.0 compiler was spent on enhancing ROSE support for Fortran instead of analysis and optimization. We describe the work on ROSE in more detail in Section 5.2.2.

2.5 CAF 2.0 Runtime System Innovations

We designed and implemented a novel algorithm for forming processor subsets and a novel representation for processor subsets as well [16]. Our algorithm for assembling teams has better asymptotic time and space complexity than all prior algorithms used for constructing communicators in MPI and outperforms them as well on large numbers of processors [19].

2.5.1 Scalable Teams

Representation As part of the CAF 2.0 runtime, we developed a scalable representation of teams that based on the concept of pointer jumping (Figure 4). Each image in a team of size S has $\lceil \lg S \rceil$ 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 $(i + S - 2^k) \bmod S$ and $(i + 2^k) \bmod S$.

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 $\lg S$ 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

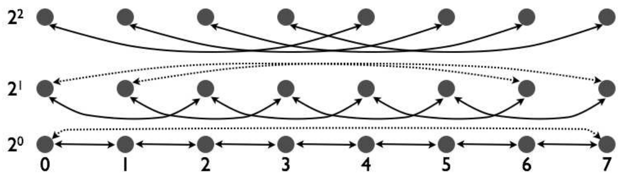


Figure 4: Members of a team of size S are linked in $\lceil \lg S \rceil$ doubly-linked circular lists. In list i, $0 \le i < \lceil \lg S \rceil$, a team member at rank j is linked to team members $(j + S - 2^i) \bmod S$ and $(j + 2^i) \bmod S$, an organization inspired by pointer jumping.

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 $(i-j) \bmod S \neq (j-i) \bmod S$. For performance, we have images cache information about how to directly communicate with a fixed modest number of frequent communication partners within one's team.

Formation To assemble teams using the aforementioned representation, we developed an innovative scalable implementation of `team_split` that takes $O(\lg^2 P)$ time on P processors. We perform a `team_split` by sorting (color, key, rank) tuples using parallel bitonic sort; performing left and right shift operations to determine team boundaries; using 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 are 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 $O(\lg^2 P)$ space on any image; we use this much space as a scratch buffer for parallel bitonic sort.

This algorithm yields lower asymptotic time and space complexity than an algorithm by Sack and Gropp in a 2010 paper entitled "A Scalable `MPI_Comm_split` Algorithm for Exascale Computing" [23]. A 2011 evaluation of scalable algorithms for `MPI_Comm_split` by Moody, Ahn, and de Supinski showed that our algorithm and an alternative hash-based approach they introduce offer the best scalability and performance [19].

Collective operations Our team representation makes for a natural expression of collective communication. All of the pairwise communications for common collective communication patterns such as binomial trees for broadcast and reduction, the dissemination pattern for barriers, power-of two circular shifts, and algorithms based on recursive doubling embed naturally in this structure.

Point-to-point communication As described in the paragraph about our team representation, one can map the from the rank of a communication partner in a point-to-point communication to the destination process using at most $\lceil \lg S \rceil$ steps.

2.5.2 Pervasive Support for Asynchrony

The CAF 2.0 runtime system contains pervasive support for asynchrony. The CAF 2.0 language design includes asynchronous copies, function shipping, and asynchronous collective communication. All asynchronous activities are managed in the CAF 2.0 runtime by an asynchronous progress engine. The progress engine provides generalized support for one-sided and two-sided collective and communication operations by maintaining a queue of in-flight operations that need to be tracked and managed. The progress engine is invoked implicitly at each communication operation. It can also be invoked explicitly directly from a CAF 2.0 program. Each time the progress engine is invoked, any activities that are no longer blocked waiting for an enabling event to occur are advanced until they complete or block again.

2.6 HPC Challenge benchmarks in CAF 2.0

The HPC Challenge suite consists of seven benchmarks organized into four categories characterized by memory access pattern that span high and low spatial and temporal locality. To evaluate the expressiveness and performance of the CAF 2.0 languages, compiler and runtime system, we implemented four of seven HPC Challenge benchmarks in CAF 2.0:

- Fast Fourier transform (FFT) measures the floating-point rate of execution of double precision complex for 1D discrete Fourier transform (DFT) on a vector of pseudo-random values.
- High performance Linpack (HPL) computes the floating point rate of execution for solving a system of linear equations, and potentially has high temporal and spatial locality.
- STREAM measures the sustainable memory bandwidth for a simple vector kernel.
- RandomAccess computes pseudo-random updates to a large distributed table of 64-bit integer values. It measures interconnect bandwidth in a system with low temporal and spatial locality.

We focused on these four benchmarks because they are the ones measured for the HPC Challenge Class II Awards Competition, which is a competition to evaluate the productivity of parallel programming models. Our implementation and evaluation of these HPC Challenge benchmarks in CAF 2.0 helped us identify shortcomings in both the programming model's expressiveness and evaluate the effectiveness of our strategies for transforming CAF 2.0 into efficient and scalable executables for supercomputers. Below, we begin with a description of these benchmarks and our experiences implementing them in CAF 2.0. In Section 2.7.1, we evaluate the productivity and performance of these benchmarks on up to 4K nodes of a Cray XT.

```
integer,target,allocatable :: panelinfo(:,2)[*]
double precision,target,allocatable :: panel(:,2)[*]
event,allocatable,dimension(:) :: delivered[*]

allocate(delivered(1:NUMPANELS)[])
event_init(delivered, NUMPANELS)
...
do j = pp, PROBLEMSIZE - 1, BLKSIZE
 cp = j / BLKSIZE + 1
 cp = mod(cp - 1, 2) + 1
 event_wait(delivered(3-cp))
 if (mycol == cproc) then
  ncol = localsize(min(BLKSIZE,PROBLEMSIZE-j), &
                   j, BLKSIZE, NPCOL, mycol)
  if (ncol > 0) then
   if (NPCOL==1) call update(m,n,BLKSIZE,ncol,3-cp)
   if (NPCOL/=1) call update(m, n, 0, ncol, 3 - cp)
  end if
  call fact(m, n, cp)
  col = col + ncol
 end if
 if (mycol == pproc) col = col + BLKSIZE
 ub = (panelinfo(5,cp)+BLKSIZE+1)*BLKSIZE
 call team_broadcast_async(panel(1:ub,cp), &
                           panelinfo(8,cp), &
                           delivered(cp),rteam)

 if (nn-ncol>0) call update(m,n,col,nn-ncol,3-cp)

 if (mycol == cproc) nn = nn - BLKSIZE
 pproc = cproc
 cproc = mod(cproc+1, NPCOL)
end do
```

Figure 5: LU factorization in the HPL benchmark.

2.6.1 HPL

The HPL benchmark measures the ability of a system to deliver fast floating point execution while solving a system of linear equations. Performance of the HPL benchmark is measured in GFLOP/s, with the calculated performance defined as $\frac{\frac{2}{3}n^3+\frac{3}{2}n^2}{t_s}10^{-9}$, where n is the order of the system of linear equations, and t_s is the time to solution (in seconds). The actual solution is done by first computing an LU factorization of the matrix corresponding to the n linear equations, using row partial pivoting of the n by $n+1$ coefficient matrix $P[A,b] = [[L,U],y]$. Then, the solution is obtained by solving the upper triangular system $Ux = y$. The lower triangular matrix L is left unpivoted and the array of pivots is not returned.

Parallel LU factorization has been studied for many years. As one of the most common implementations of parallel LU factorization for distributed memory machines, HPL [1] has been used as a reference code for HPC Challenge competition and for determining the Top 500 list. Different implementations of HPL using PGAS and other parallel languages have been developed and studied in the past [5, 6, 9]. In this section, we highlight features of our implementation in CAF 2.0.

Creating teams for block-cyclic distribution Our CAF 2.0 implementation of HPL implements a sophisticated tiling of the computation, capable of varying both the arrangement of processor cores for the overall computation as well as the width of each panel of data that each core blocks data into. The entire matrix is distributed in block-cyclic fashion to a processor grid in one or two dimensions. The block size of the data distribution is determined by the width of the panel. Our team representation for process subsets provides a general and efficient method of synchronization and communication between processes. To support row- and column-wise communication in HPL, we created both row and column teams. Subteams are further created when it is necessary.

Hiding latency with asynchronous broadcast To improve the performance of HPL, we used a dual panel structure for factorization. This not only increases parallelism in factorization but also overlaps communication latency with computation when we use asynchronous split-phase panel broadcast. An asynchronous broadcast of a panel is first initiated after the panel is factored. Processors in the next column team can start their updates and factor the next panel as soon as they receive the previous panel. The broadcast of the second panel is overlapped with the update of the trailing matrix from the previous factored panel and with the computation of the next panel factorization. An `event_wait` is performed before the data communicated is used. Figure 5 shows a code snippet for the LU factorization.

2.6.2 STREAM

The HPC Challenge STREAM benchmark evaluates the extent to which a parallel system can deliver and sustain peak memory bandwidth by performing a simple vector operation that scales and adds two vectors: $a \leftarrow b + \alpha c$. Performance of the STREAM benchmark is measured in GByte/s, with the calculated performance defined as $24\frac{m}{t_{min}}10^{-9}$, where m is the size of the vectors, required to be at least a quarter of system memory; and t_{min} is the minimum execution time over at least 10 repetitions of the benchmark kernel. The STREAM benchmark is embarrassingly parallel: the work performed on any one node is independent of that performed on others.

```
double precision, allocatable :: a(:)[*]
double precision, allocatable :: b(:)[*], c(:)[*]
! allocate with the default team
allocate(a(ndim)[], b(ndim)[], c(ndim)[])
...
do round = 1, rounds
  do j = 1, rep
    call triad(a,b,c,n,scalar)
  end do
  call team_barrier()
end do
...
subroutine triad(a, b, c, n ,scalar)
  double precision a(n), b(n), c(n), scalar
  a = b + scalar * c
end subroutine triad
```

Figure 6: Implementation of the STREAM benchmark.

Since the STREAM benchmark does not require communication between processes, we have implemented the Coarray Fortran version exactly like the sequential one with the exception that all arrays are allocated with our Coarray Fortran 2.0 allocator as shown in Figure 6.

2.6.3 RandomAccess

The HPC Challenge RandomAccess benchmark evaluates the rate at which a parallel system can apply updates to randomly indexed entries in a distributed table. Performance of the RandomAccess benchmark is measured in Giga Updates Per second (GUP/s). GUP/s is calculated by identifying the number of table entries that can be randomly updated in one second, divided by 1 billion (10^9). The term "randomly" means that there is little relationship between one table index to be updated and the next. An update is a read-modify-write operation on a 64-bit word in the table. A table index is generated, the value at that index is read from memory, modified by an integer operation (xor) that combines the current value of the table entry with a literal value, and the resulting value is written back to the table entry.

```
event,allocatable :: delivered(:)[*],received(:)[*]
integer(8),allocatable :: fwd(:,:,:)[*]

do i = world_logsize-1, 0, -1
  ...
  call split(ret(:,last), retsizes(last), &
       ret(:,current), retsizes(current), &
       fwd(1:,out,i),fwd(0,out,i),bufsize,dist)

  if (i < world_logsize-1) then
    event_wait(delivered(i+1))
    call split(fwd(1:,in,i+1), fwd(0,in,i+1), &
         ret(:,current), retsizes(current), &
         fwd(1:,out,i),fwd(0,out,i),bufsize,dist)
    event_notify(received(i+1)[from])
  endif

  copy_async(fwd(0:outgoing_size,in,i)[partner], &
             fwd(0:outgoing_size,out,i), &
             delivered(i)[partner], received(i))
  ...
end do

! each process image applies its local updates
......
```

Figure 7: Implementation of a routing algorithm in RandomAccess.

On distributed-memory parallel systems that lack hardware support for shared memory, fine-grain operations on remote data are expensive. To develop a high performance implementation of RandomAccess in CAF 2.0, we exploit the "1024 element look ahead and storage" allowed by the problem specification [12]. A sketch of the implementation is shown in Figure 7. First, each process image generates a batch of 1024 indices of table locations to be updated. Next, the code uses a hypercube-based pattern of bulk communication to route updates to the process image co-located with the table index being updated. Finally, each process image applies updates locally.

Similar software routing strategies have been used before, though never with CAF. Researchers at Sandia studied a different but related strategy for RandomAccess using all-to-all communication based on a hypercube communication pattern [21]. IBM also explored a software routing strategy for this benchmark on Blue Gene systems [11].

2.6.4 FFT

The HPC Challenge *FFT* (Fast Fourier Transform) benchmark measures the ability of a system to overlap computation and communication while calculating a very large Discrete Fourier Transform of size m with input vector z and output vector Z:

$$Z_k \leftarrow \sum_{j}^{m} z_j e^{-2\pi i \frac{jk}{m}}; 1 \le k \le m$$

Performance of the FFT benchmark is measured in GFLOP/s, with calculated performance defined as $5\frac{m\log_2 m}{t}10^{-9}$, where m is the size of the DFT and t is the execution time (in seconds). The number of processors for this benchmark may be implementation-specific; in particular, it is allowed to be an integral power of 2. Parallel FFT algorithms have been well studied in the past [28, 4, 15]. The reference FFT implementation of the HPC Challenge benchmarks uses a 1D algorithm based on [28].

```
complex, allocatable :: c(:,2)[*]
event, allocatable :: ready(:)[*], copied(:)[*]
event, allocatable :: prefetch(:)[*]

......
do l = lcomm, levels
  ......
  event_notify(ready(l-lcomm)[partner])
  event_wait(ready(l-lcomm))

  ! prefetch blocks
  do outer = 0, (n_local_size/2)-1, blksize
    copy_async(buf(lo:hi),c(lo:hi,last)[partner],&
           prefetch(outer/blksize),copied(l-1-lcomm))
  end do

  do outer = 0, (n_local_size/2)-1, blksize
    event_wait(prefetch(outer/blksize))
    ! perform computation
    ......
    ! send result to who next needs it
    copy_async(c(lo:hi,curr)[partner], &
               buf(lo:hi), copied(l-lcomm)[partner])
  end do
enddo
```

Figure 8: Using `copy_async` for hiding communication latency in FFT.

Our CAF 2.0 FFT implementation uses a radix 2 binary exchange formulation that consists of three parts: permutation of data to move each source element to the position that is its binary bit reversal; local (in-core) FFT computation for as many layers of the DFT calculation as all fit in the memory of a single processor; and remote DFT computation for the layers that span multiple processor images. Figure 8 shows the main loop body of the remote FFT computation using asynchronous copy.

2.7 Evaluation of CAF 2.0 Using HPC Challenge Benchmarks

Our CAF 2.0 design and implementation has been carefully crafted to deliver scalable high performance on parallel systems with thousands processor cores. In this section, we evaluate both productivity and performance.

2.7.1 Winner: HPC Challenge Class II *Most Productive Language*

In November 2010, at the SC10 conference in New Orleans, CAF 2.0 was awarded the HPC Challenge Class II Award for *Most Productive Language.*

One of the key criteria used in the annual HPC Challenge awards competition [13] is the number of source lines in the implementation of each benchmark. Table 2.7.1 shows the count of source lines in each of benchmark implementation in CAF 2.0. The table breaks source lines into four categories: computation, communication and synchronization, declarations, and comments/white space. As the table shows, communication and synchronization only account for a very small proportion of the entire implementation.

Table 2.7.1 also compares the total source lines of code between CAF 2.0, the Chapel [8] implementations, and the reference MPI implementations of HPCC [14]. Chapel is a good point of comparison because it was the winner of the prize for most elegant implementation of HPC Challenge benchmarks. In our comparison, we exclude comments and spaces. The CAF 2.0 STREAM code is shorter than the Chapel code. For the other benchmarks, the CAF 2.0 implementations are a factor of 2 to 3 times larger. In comparison with the reference implementation of HPCC, CAF 2.0 implementations are significantly shorter. We believe that the best way to measure pro-

	STREAM	RA	FFT	HPL
Computation	30	170	188	532
Communication & sync	0	13	15	50
Declaration	15	121	98	109
Total (Benchmark)	**45**	**304**	**301**	**691**
Comments & spaces	13	95	138	95
Total (Program)	58	399	439	786
$\frac{SLOC(\text{CAF2.0 benchmark})}{SLOC(\text{Chapel benchmark})}$	0.38	1.88	1.37	2.99
$\frac{SLOC(\text{CAF2.0 benchmark})}{SLOC(\text{MPI HPCC})}$	0.10	0.18	0.21	0.06

Table 5: Source lines of code (SLOC) of HPC Challenge benchmarks in CAF 2.0, and their comparison with Chapel [8] and reference MPI implementation of HPCC [14]. A ratio being less than 1 means that the CAF 2.0 implementation is smaller.

ductivity is the performance divided by the number of lines. In our implementations, we favored performance over brevity. In the next section, we argue that the performance benefits we reap from more sophisticated implementations outweigh the increase in code size.

2.7.2 Performance

To evaluate the performance of our CAF 2.0 implementations of the RandomAccess, FFT, and HPL HPC Challenge benchmarks, we ran them on up to 4096 cores of Franklin, a Cray XT4 system at the National Energy Research Scientific Computing Center, and the Cray XT 4 partition of Jaguar, a machine at Oak Ridge National Laboratory. Each node in Franklin contains a 2.3 GHz single socket quad-core AMD Opteron processor (Budapest) (theoretical peak performance of 9.2 GFLOP/s per core) and 2GB of memory per core. The memory speed is 800 MHz. Each node is connected to a dedicated SeaStar2 router through Hypertransport. The SeaStar2 interconnect is arranged as a 3D torus. Jaguar contains 7,832 compute nodes in its XT4 partition. Each node contains a quad-core AMD Opteron 1354 (Budapest) processor running at 2.1 GHz. Some nodes use 8 GB of DDR2-800 memory; others use DDR2-667 memory. Jaguar nodes are connected with a SeaStar2 router.

Our runtime library is built on the UC Berkeley's GASNet communication library version 1.14.2 with its portal conduit implementation for communication. We used Cray's PrgEnv-pgi/2.248B and the Portland Group's PGI 10.0 Fortran compiler for compiling the Fortran 90 codes generated by our CAF translator with compiler option "-fastsse".

HPL In our experiments with HPL, we allocate the matrix as a coarray with $12K \times 12K$ double precision array elements on each core to meet the requirement of the HPC Challenge specification. We used the Cray Scientific Libraries package, LibSci 10.4.3, for matrix multiply operations in updating the trailing matrix.

An important parameter for HPL is the block size of the block-cyclic data distribution. We also used this block size as the panel width for factorization. As demonstrated by a performance study we conducted (not included in this paper), the choice of block size has significant impact on the performance of the benchmark. An ideal block size is large enough to achieve high performance in updating the trailing matrices, but small enough to maintain a good load balance for scalability. Another factor is the topology of processor cores along the two dimensions. We see great potential here for a future auto-tuning study of parallel performance.

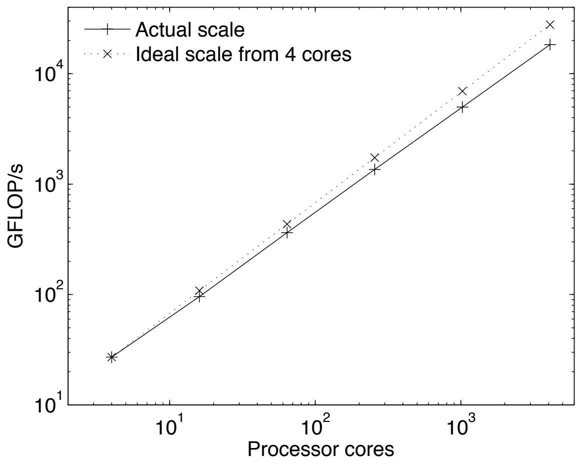


Figure 9: Performance of HPL.

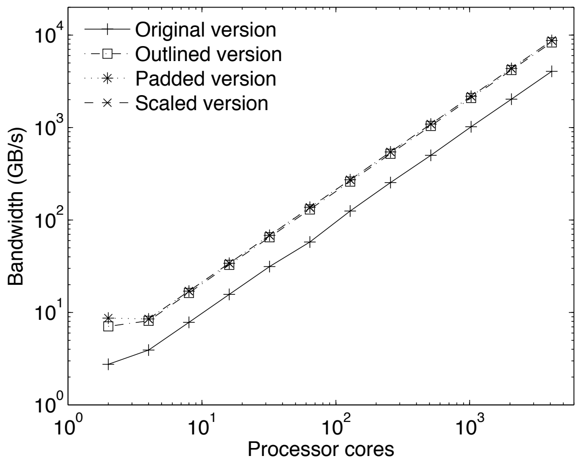


Figure 10: Performance of STREAM.

An important issue we addressed in the course of implementing HPL is how to improve node performance by avoiding data copying. Passing array sections or whole arrays in procedure calls may incur extra data copying if the compiler cannot determine the data to be contiguous. We carefully combined array pointer assignments and parameter passing of the array locations and their leading dimensions to avoid unnecessary data copying.

To achieve scalable high performance on a large parallel machine, we also need to reduce communication overhead. We implemented asynchronous broadcast in CAF 2.0 and applied it to the panel broadcast of our HPL implementation. The overall performance of HPL is shown in Figure 9. This shows our current implementation of HPL scales very well up to thousands of processor cores. We achieved 18.3 TFLOP/s on 4096 cores. However, the roughly 10% difference on the 4096 core run compared with the linearly scaled performance indicates that there is still room to improve. Our analysis with the Rice HPCToolkit [2] shows that our implementation of asynchronous operations incurs unnecessary overhead in each advance step. Carefully removing this overhead and fine tuning the overlapping of broadcast overhead with computation is future work.

STREAM In CAF 2.0, coarray data is represented as a Fortran 90 pointer within the generated Fortran code. This could cause trouble for the backend Fortran compiler when it tries to generate prefetch instructions for the STREAM kernel. Although vectors a, b and c in the stream equation are disjoint, the fact that they are allocated with our CAF allocator makes them opaque to the underlying Fortran compiler. This decreased STREAM's performance by 50% as shown in Figure 10. We resolved this performance gap by wrapping the STREAM kernel within a subroutine `triad` that declares a, b and c as regular Fortran array, giving the underlying Fortran compiler a chance for optimization.

Initially, our implementation gave performance 70% of sequential Fortran. While working to get STREAM node performance up to that of sequential Fortran, we identified an alignment issue that causes corresponding vector elements to map to the same cache lines, resulting in conflict misses. The K&R malloc that we used in our implementation of the CAF 2.0 runtime system aligned the allocation of large memory regions on page boundaries; this caused corresponding vector elements to be exactly spaced at a multiple of the page size. To address this issue, for larger memory allocations, we adjusted the CAF 2.0 memory manager to insert a small, variable amount of padding for large blocks of allocated memory; this caused them to be differently aligned and closed the remaining performance gap in this benchmark. With allocation padding, single thread

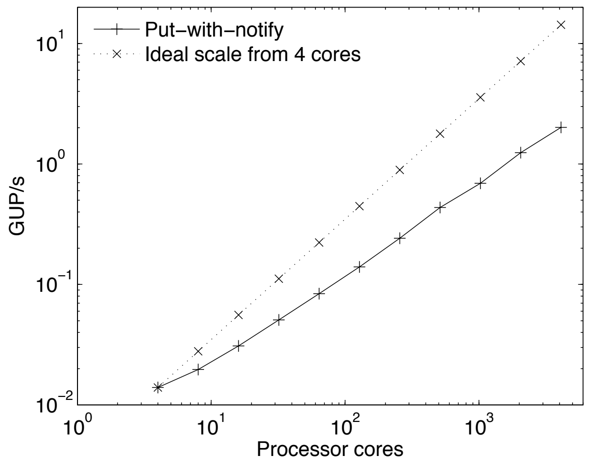


Figure 11: Performance of RandomAccess.

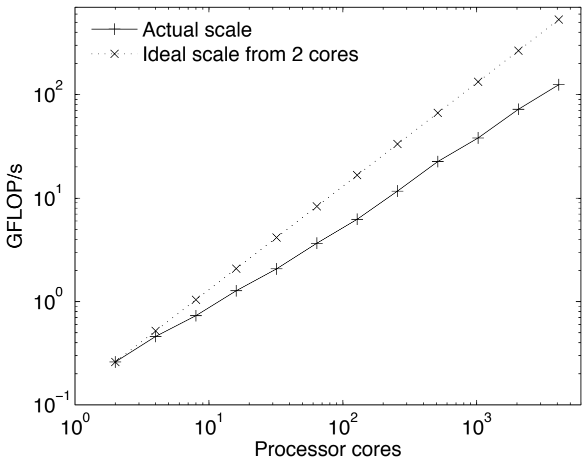


Figure 12: Performance of FFT.

performance of STREAM increased from 4.2 GByte/s to 5.5 GByte/s on a Cray XT4 system. Our implementation of STREAM achieves 8.73 TByte/s on 4096 cores.

RandomAccess The performance of our CAF 2.0 implementation of RandomAccess running on a Cray XT4 is shown in Figure 11. The reported results use a 1GB table per core, consistent with the benchmark specification that the program use half of each node's memory for the table. The graph shows two lines: ideal relative scaling starting at 4 processors and actual scaling. The gap between actual and ideal performance reflects two issues. First, each doubling of the number of processors adds another stage to our software routing protocol. Second, as the number of processors gets larger, congestion for links and limited bisection bandwidth begin to come into play. Our largest run on 4096 cores achieved 2.01 GUP/s, a very respectable result.

FFT When we analyzed our initial implementation of FFT with Rice University's HPCToolkit performance tools [2] and found that our loop for the permutation step was consuming 75% of execution time on a 32-processor core run. The initial loop simply walked through each processor's data array, performed a bit reversal on the calculated index, and shipped the datum to the appropriate remote location. This gave poor performance and poor scalability because we were flooding the memory interconnect with many tiny messages, which leads to high overhead.

To resolve this situation, we decomposed the permutation into three stages: packing the data into a series of blocks, one per processor; transmitting the blocks to the correct destination processor with an all-to-all collective operation; and then unpacking the data. On a machine with a multi-level memory hierarchy, one must take considerable care when packing and unpacking the communication buffers as part of the bit reversal. We performed a study that showed that performance of the packing loop improves by over a factor of 10 by blocking the packing loop to pack a subset of the data for a subset of the processors before moving on rather than simply performing a strided gather for each processor's buffer. Similarly, careful unpacking was faster than naive unpacking by roughly a factor of eight. For the all-to-all step, we developed a prototype implementation that uses a hypercube routing protocol to route data to its destination processor in $\log P$ steps; however, this implementation works only for integral powers of two. We plan to implement Bruck's algorithm [7] in the near future to support all-to-all collective operations on arbitrarily-sized collections of processors. While the original elementwise bitreversal consumed 75% of the original running time, our optimized bitreversal reduced the cost to about 6% of the reduced running time.

With the permutation problem solved, we then focused our energies on improving the performance of the remote DFT step. We arranged to overlap communication and computation by strip mining the main loop performing the element-wise calculations within each "butterfly" involving remote data. While one chunk of butterfly is being calculated, the previous chunk is available for communication. We leverage the CAF 2.0 `copy_async` to transfer the chunk asynchronously to our partner for calculating a butterfly. Then, to transfer the data asynchronously from our new partner to ourselves at the beginning of the next round of calculating butterflies, we use a series of predicated `copy_async` operations to prefetch the data conditioned on its availability. As soon as the first chunk reaches us, we can immediately start processing it instead of waiting for the rest to arrive.

After optimizing the bit reversal and employing asynchronous copies to overlap communication with computation, we obtained the scalable performance shown in Figure 12, peaking at 125 GFLOP/s with 4096 processor cores. Although the performance of FFT did not improve much with the addition of asynchrony, we note that pipelining the DFT computations has us set up for further improvements. Rather than sending a processed chunk of data back to one's partner so that it can be communicated to the partner's next partner image in the processing of the next butterfly, we can send it directly to that image and save an entire copy. This should improve the performance of FFT further. Finally, the absolute performance of FFT could be improved by using a higher radix DFT computation.

2.7.3 Comparison with other PGAS implementations

Comparison with IBM's UPC implementation The performance of our CAF 2.0 implementations of RandomAccess and HPL on quad-core nodes of a Cray XT is comparable to the performance that IBM achieved using UPC on a 4096 core Blue Gene/P system [13].[2] The IBM UPC implementation relied on their UPC compiler to automatically transform element-wise table updates to use function shipping. On a 4096-processor rack, IBM's HPL implementation achieved 8.12 TFLOP/s and their RandomAccess implementation achieved 1.21 GUP/s. Their FFT implementation used all-to-all collective communication instead of pairwise communication. Based on a projection from their two rack result, the combination of these factors yielded performance more than twice what we achieved.

Comparison with Cray's Chapel implementation Unlike Cray's HPCC benchmark implementations in Chapel, our approach to productivity was to focus on achieving high performance rather than keeping the number of source code lines to a minimum. Our approach to implementing RandomAccess underscores this point. While our RandomAccess implementation using software routing of updates was nearly twice the size of Cray's element-wise updates in Chapel (304 vs. 162 lines of code), our more sophisticated approach is a factor of 32 faster than the implementation in Chapel on a Cray XT4, which achieved only .0612 GUP/s on 4096 cores (1024 quad-core processors, 4 active cores each) [8]. The Cray implementation is in part much slower because it uses element-wise remote table updates. For HPL, we have a scalable parallel implementation of HPL, whereas the Chapel implementation of HPL ran on a single locale only. Even our implementation of the STREAM triad was affected by our performance centric approach. Outlining the triad into a separate procedure enabled us to communicate the lack of aliases to the backend compiler, which boosted performance by roughly 50%. As a result, our STREAM triad implementation netted 8.73

[2]Cray XT and Blue Gene/P systems have some significant differences; hence, our comparison is qualitative rather than quantitative.

TByte/s on 4096 cores, whereas the Chapel implementation of EP STREAM triad ran at 6.26 TByte/s on a comparable system configuration (1024 quad-core processors, 4 active cores each). Finally, even with significant delays due to an interaction between asynchrony and the GASNet Portals conduit, our CAF 2.0 implementation of FFT outperforms the Chapel implementation by a factor of over 24,000 on 16 quad-core processors with four active cores each—the largest system size for which Chapel FFT results are reported.

2.7.4 Comparison with Class 1 award winner implementation

The 2010 HPC Challenge Class 1 award winner implementation from the Oak Ridge National Laboratory was run on the XT5 partition of Jaguar. The XT5 partition contains 18688 compute nodes, each contains dual hex-core AMD Opteron 2435 (Istanbul) processors running at 2.6 GHz, 16 GB of DDR2-800 memory and a SeaStar 2+ router. The implementation uses a combination of C, MPI, and multithreading. Their HPL implementation achieved 66% of the peak performance on 224220 cores which is higher than our 49% of the peak on 4096 cores. Their FFT entry achieved 10.7 TFLOP/s on 196608 cores using Cray modified FFTW 3.2 based implementation. However, our RandomAccess is approximately 20% slower than the Class 1 result, which achieved 37.7 GUP/s on 223112 cores. (Assuming logarithmic scaling, we project that our approach would achieve 30.9 GUP/s on 223112 cores.) Our STREAM performed more efficiently per node than the Class 1 result, which achieved 398 TByte/s on 223112 cores.

3 Research Objectives Remaining

Over the course of this project, we have shifted our focus from broad application of the Coarray Fortran language to a more narrow, but much more extensive reimagining of the Coarray Fortran language that we have designed, prototyped, and evaluated. We thus dove more deeply into our goals of refining language-based parallel programming models for emerging platforms and for performance and expressiveness, with a consequent lessening of focus in other areas of the proposal.

The need for a flexible compiler substrate for supporting new language features in the CAF 2.0 programming model led us away from the Open64 compiler and towards the LLNL ROSE infrastructure. However, the relative immaturity of ROSE required us to spend more effort on "hardening" it than on investigating compiler technologies surrounding it.

Although we lacked time to explore this issue, we believe that global view arrays would add value to CAF. The Japanese XScalableMP effort combines global view arrays with local view programming. We believe that this is an important direction to explore in the future. It may lead to a more natural expression of irregular problems with access patterns like A(B(I)) = C(D(I), described by Rich Barrett as critical to Sandia in his talk at the ASCR Pogramming Models Workshop in August, 2011.

Although we have not directly explored interoperability of CAF 2.0 with other programming models, we have been conscious throughout this effort that interoperability is a key requirement for the success of any extension to a mainstream programming language. We have therefore been very careful to ensure that nothing we do precludes interoperability with other programming models.

We have not published any results of integration of CAF 2.0 with programming environments and tools; however, in order to evaluate the performance of applications written in CAF 2.0, we have made extensive use of the HPCToolkit suite developed at Rice with DOE support through the Performance Engineering Research Institute and the Center for Scalable Application Development Software.

4 Findings

In brief, significant findings of the project were the following:

- Numrich and Reid's original design for Coarray Fortran had significant flaws that merited a complete redesign. For instance, the lack of support for processor subsets made it unsuitable for use in writing coupled applications. In addition, the lack of built-in support for collective communication meant that users would end up writing their own non-scalable implementations. Furthermore, the lack of adequate mechanisms for latency hiding meant that it would not scale well to large systems.

- Our new Coarray Fortran 2.0 design is expressive and admits high performance implementations, as shown by our experience with the HPC Challenge benchmarks. At SC10, CAF 2.0 was awarded *Most Productive Language* at the HPC Challenge Awards.

- Interoperability with other programming models can't be achieved at the programming language level. For multiple programming models to interoperate, there must be a common runtime layer below them all. In particular, there must be an agreed upon scalable representation for data descriptors that can be exchanged between runtime systems for the different languages. Addressing this issue was part of the scope of the UNISTACK proposal to the DOE SciDAC-3 program.

- The GASNet substrate is presently not ideal as a low-level infrastructure suitable as a compiler target for PGAS languages. In particular, some of the operations in the GASNet interface bundle too much semantics into its interface operations. For instance, when calling a GASNet routine to issue a non-blocking PUT, the call does not return until the source data can be overwritten. On a modern system with RDMA support, it should be possible to simply queue the operation for service, return immediately and check for completion later. Similarly, there is no public interface to GASNet operations that don't invoke the GASNet progress engine as a side effect. This led to a problem when implementing an asynchronous progress engine for CAF 2.0. For the future, there should be a unified progress engine in GASNet that provides extension hooks for use by hosted language runtime systems. Addressing these issues were part of the scope of the UNISTACK proposal to the DOE SciDAC-3 program.

- The only way to move the computational science community to new programming models is through language standards. Computational scientists won't switch to a programming model unless it is codified in a language standard, guaranteeing them longevity and portability for their code for years to come. Thus, our effort to engage the Fortran standards committee is an essential step to transferring our ideas into common use.

- Changes in emerging architectures require that programming models for HPC platforms address the issues of dynamic multithreading, load balance, and accelerators. We began work on multithreading, exploring both function shipping and work stealing to address this issue. Language support that will naturally enable application developers to harness accelerators is also essential for many emerging systems.

- Scalable algorithms will be essential to meet the challenges of next-generation systems. We expect that topology awareness will play an important role as well. Our work on scalable team construction, which served to catalyze follow on work in the MPI community was the first step in this direction. More work is necessary.

- Asynchronous algorithms will play an increasingly important role for future systems. At present, implementing asynchronous algorithms in the runtime is somewhat difficult. Coordination between our CAF 2.0 runtime, the GASNet communication substrate, the underlying

OS, and hardware support is somewhat problematic. We believe that overhauling support for asynchrony will require significant attention for future systems.

- Fault tolerance concerns, expected to be significant at the exascale, can't be handled by a programming language alone. The underlying communication infrastructure must provide appropriate support. For instance, in our CAF 2.0 implementation using GASNet, there is presently no way to deal with failing processes or nodes at the GASNet level. This must be the subject of future work.

5 Products of the Research

5.1 An Open Source Compiler and Runtime for CAF 2.0

In addition to feature development for CAF 2.0, we have expended considerable energy on the usability of our CAF 2.0 compiler and runtime system and on making it publicly available. To this end, we have written a sophisticated compiler driver script that automatically combines our runtime library, coarray type wrapper modules, and proper runtime initialization procedures with translated CAF 2.0 source code. The script supports multiple back-end Fortran 90 compilers, including GNU `gfortran`, the PGI group's `pgf90`, and Intel's `ifort` compiler. CAF 2.0 users do not need to manually compile the disparate intermediaries created in the process of translating CAF 2.0 source code to standard Fortran 90. This in turn vastly simplifies development of Makefiles for CAF-based projects.

In addition to the CAF compiler driver script, we have developed extensive autoconfig and automake scripts that allow the CAF 2.0 language to be built on a wide array of host hardware, and with multiple compilers. In particular, we have used these scripts to support CAF 2.0 on Cray XT4 and XT5 machines, and with both the GNU and PGI back-end Fortran 90 compilers.

We are dedicated to the proposition of open-source software, and to this end we have provided direct (read-only) access to our svn source control repository for anyone interested in CAF 2.0. This pre-alpha release of the CAF 2.0 software is being evaluated by groups all over the world. We consider our release preliminary because the ROSE Fortran infrastructure does not yet provide sufficiently robust support for Fortran to enable our compiler to be used with arbitrary programs. Full support for Fortran in ROSE is still a work in progress. Instructions for downloading and using the CAF 2.0 software are described in formal release notes served from a website (`caf.rice.edu`) dedicated to the CAF 2.0 project.

5.2 Contributions to Community Open-source Infrastructure

5.2.1 GASNet

Over the course of this project, we coordinated with the GASNet team at LBNL to improve their communication library for general usability and for CAF 2.0 needs in particular. In evaluating the scalability of executables generated from CAF 2.0 programs, we identified an issue with the GASNet startup routines that required every pair of processors to effect a pairwise exchange of data. The quadratic overhead of this exchange was not scalable and led to increasingly high overhead as we experimented with larger and larger groups of processor cores. Based on our feedback, the LBNL group implemented a more scalable initialization sequence that led to dramatic improvements in the GASNet startup time. We also tracked down the root cause of a bug that caused GASNet to fail with more than 4096 processor cores, enabling it to be resolved.

5.2.2 ROSE

Over the course of this project, we worked closely with both LANL and LLNL to enhance support for Fortran in the LLNL's ROSE compiler infrastructure, a well respected and widely used tool for HPC programming model research. We focused on ROSE's Fortran language support, which is both necessary for our project and in demand by the HPC community at large.[3] Our work has resulted in measurably improved Fortran language coverage, more comprehensive test suites, and better design and implementation quality.

Why does so highly regarded a system need so much work? The answer is historical. ROSE's support for C++ is quite mature (over a decade of steady development), whereas Fortran support is still in its infancy. Moreover, ROSE's C++ front end is based on a robust commercial compiler, whereas its Fortran front end is a newly hand written one based on an open source Fortran grammar still being developed. Consequently, ROSE Fortran is incomplete and buggy and needs a lot of work to become as useful as ROSE C++.

We found problems with ROSE's implementation of Fortran in all areas of operation, including lexical behavior (line endings, line continuation, constant handling); syntactic behavior (unimplemented and misparsed constructs); semantic behavior (name scoping, type checking, intrinsic functions and modules); unparsing behavior (invalid Fortran output, output not equivalent to input); and pragmatic behavior (configuration, option handling, failed assertions, crashes).

We worked closely with the group at LANL that produces the OpenFortranParser being used as the Fortran front-end for ROSE. We helped them identify and resolve errors in the Fortran grammar. In addition, we devised a set of CAF 2.0 extension hooks that can folded back into the main parser branch. These hooks enable the parser to be used both with CAF 2.0 (by matching parsing code to the hooks so that CAF 2.0 syntax is recognized and processed) and with standard-issue Fortran (by leaving the hooks unmatched).

In September 2010 we began a serious effort to improve the robustness of ROSE Fortran. We established a close collaboration with the ROSE team, including internal developer privileges. We focused on three widely used benchmark applications. First, the GFortran test suite, contains 1801 files with roughly 79,000 lines of code, excluding negative, Gnu-specific, and multi-language tests. Second, LANL's Parallel Ocean Program (POP) code consists of 63 files and 68,000 lines of code in a representative build. Finally Sandia's S3D combustion code contains 65 files and about 45,000 lines of code in a representative build.

When we began our "hardening", ROSE failed on 44% of the files in the GFortran test suite and on 50% each of POP and S3D. As of this writing, we have fixed about a hundred bugs in ROSE and redesigned and reimplemented several key parts of the ROSE Fortran front end. We have also contributed bug fixes to the open source ANTLR parser generator project. As a result of our efforts, ROSE failures are down to 21% in the GFortran test suite and zero failures in POP and S3D. We have added both POP and the GFortran test suite to the ROSE automatic test suite.

5.3 Contributions to Community Standards

5.3.1 Fortran 2008

Beginning with our critique of proposed coarray and synchronization features for the Fortran 2008 standard [17], we have had a long interaction with Working Group 5 of the standards committee. Based in part on our input, these features were deferred from the initial standard into a TR for further consideration. After a period of feedback on the TR, we submitted another critique [3] on the proposed modifications to the TR. Because of our experience with advanced CAF 2.0 features, we have been asked to participate directly in the upcoming (as of this writing) standards meeting in October 2011 in Las Vegas to advise the standards committee.

5.3.2 MPI

Our algorithm for team creation, which we wrote in 2009, yields lower asymptotic time and space complexity than an algorithm by Sack and Gropp in a 2010 paper entitled "A Scalable

[3]Our work on ROSE at Rice was jointly supported by the Center for Scalable Application Development Software.

`MPI_Comm_split` Algorithm for Exascale Computing" [23]. A 2011 evaluation of scalable algorithms for `MPI_Comm_split` by Moody, Ahn, and de Supinski showed that our algorithm and an alternative hash-based approach they introduce offer the best scalability and performance [19]. We thus expect this algorithm to be adopted into the MPI standard shortly.

5.4 Technical Communications

5.4.1 Presentations

- John Mellor-Crummey. Coarray Fortran: An emerging language for parallel programming. Future Technologies Colloquium Series, ORNL, July 2006.
- John Mellor-Crummey, Programming Models for Scientific Computing on Leadership Computing Platforms: The Evolution of Coarray Fortran, CScADS Workshop on Leadership-class Machines, Petascale Applications, and Performance Strategies. Snowbird, UT, July 2008.
- William Scherer, CAF 2.0: A Next-generation Coarray Fortran, International Workshop on Peta-Scale Computing Programming Environment, Languages and Tools. Tsukuba, Japan, March 25, 2009.
- John Mellor-Crummey, Coarray Fortran: Past, Present, and Future. Workshop on Workshop on Leadership-class Machines, Petascale Applications, and Performance Strategies. Center for Scalable Application Development Software Summer Workshop Series, Tahoe City, CA, July 2009.
- John Mellor-Crummey, Laksono Adhianto, and William Scherer III, A New Vision for Coarray Fortran, Los Alamos Computer Science Symposium, Santa Fe, New Mexico, October 14, 2009.
- John Mellor-Crummey, Coarray Fortran: Past, Present, and Future. Workshop on Workshop on Leadership-class Machines, Petascale Applications, and Performance Strategies. Center for Scalable Application Development Software Summer Workshop Series, Snowbird, UT, July 2010.
- John Mellor-Crummey, Laksono Adhianto Mark Krentel, Guohua Jin, William Scherer III, Chaoran Yang. 2010 HPC Challenge Class II Submission: Coarray Fortran 2.0. HPC Challenge Awards Competition, SC10. New Orleans, LA, November 2010.
- John Mellor-Crummey. Coarray Fortran 2.0: A productive language for scalable scientific computing. Keynote Address, 16th International Workshop on High-Level Parallel Programming Models and Supportive Environments. Held in conjunction with IPDPS 2011, Anchorage, AK, May 2011.
- John Mellor-Crummey. Coarray Fortran for Exascale Systems? Workshop on Future Approaches to Data Centric Programming for Exascale. Held in conjunction with IPDPS 2011, Anchorage, AK, May 2011.
- Brad Chamberlain, Barbara Chapman, Daniel Chavarria, Laxikant Kale, John Mellor-Crummy, DK Panda. Programming Models at Exascale: Are we ready for the Challenges? Panel presentation. Workshop on Future Approaches to Data Centric Programming for Exascale. Held in conjunction with IPDPS 2011, Anchorage, AK, May 2011.
- John Mellor-Crummey, Coarray Fortran 2.0: A Productive Language for Scalable Scientific Computing. HIPS 2011: Workshop on Leadership-class Machines, Extreme Scale Applications, and Performance Strategies. Center for Scalable Application Development Software Summer Workshop Series, Tahoe City, CA, July 2011.

- John Mellor-Crummey, Lessons from the Past, Challenges Ahead, and a Path Forward. DOE ASCR Workshop on Exascale Programming Challenges, Marina del Rey, CA, August 2011.

5.4.2 Papers

- John Mellor-Crummey, Laksono Adhianto, Guohua Jin, and William Scherer. A New Vision for Coarray Fortran, *The 3rd Conference on Partitioned Global Address Space (PGAS) Programming Models (PGAS 2009)*, Oct., 2009, Ashburn, Virginia.

- W. N. Scherer III, L. Adhianto, G. Gin, J. Mellor-Crummey, and C. Yang. Hiding Latency in Coarray Fortran 2.0. *The 4th Conf. on Partitioned Global Address Space (PGAS) Programming Models (PGAS 2010)*, Oct. 2010, New York, New York.

- G. Jin, L. Adhianto, J. Mellor-Crummey, W. N. Scherer III, and C. Yang. Implementation and Performance Evaluation of the HPC Challenge Benchmarks in Coarray Fortran 2.0. *The 25th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS 2011)*, May 2011, Anchorage, Alaska.

5.4.3 Reports

- John Mellor-Crummey, Laksono Adhianto, and William 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.

- Laksono Adhianto, John Mellor-Crummey, Guohua Jin, Karthik Murthy, Dung Nguyen, William N. Scherer III, Scott Warren, and Chaoran Yang. A Critique of ISO/IEC JTC1/SC22/WG5 N1835 (Addition/Modification of CAF Features). Available from the Fortran Standards Committee Working Group 5's Document Repository. URL: ftp://ftp.nag.co.uk/sc22wg5/N1851-N1900/N1856.txt, June 2011.

5.5 Awards

- Most Productive Language, HPC Challenge Class II Award. HPC Challenge Awards Competition, SC10. New Orleans, LA, November 2010.

6 Conclusions of the Project

When the project began, Coarray Fortran was widely regarded as an up and coming partitioned global address space model for parallel programming. As we explored the semantics and expressiveness of this model, we discovered numerous problems and limitations. We reported these to the Fortran 2008 Standards Committee and were able to significantly influence the standardization process to avoid having the wrong set of coarray features added to Fortran 2008.

Our subsequent exploration of these issues led us to conclude that a new set of coarray constructs was the only solution to the raft of problems we had discovered. We realized our solutions to these problems in a new set of extensions for Fortran that we call Coarray Fortran 2.0 (CAF 2.0).

Over the course of this project, we demonstrated that our new CAF 2.0 language extensions are useful for expressing applications with a wide range of characteristics. As shown by our receipt of the 2010 HPC Challenge Award for the Most Productive Language, CAF 2.0 applications are not only highly performant, but compact as well. The relative simplicity of writing applications in CAF 2.0 yields a language that is highly productive and expressive. We attribute this success in large part to careful selection of features most needed for a range of HPC applications, and to keeping these features orthogonal and easy to use.

However, more work remains to be done. In particular, the CAF 2.0 programming model needs further refinement to better support coupled applications. Additionally, renewed focus on interoperability with other programming models and environments will greatly simplify adoption of CAF 2.0 for many users. As we look forward to the exascale, more research is needed into managing massive multithreading in the presence hierarchical locality and deep memory hierarchies. Furthermore, language and compiler support is necessary for hardware accelerators such as GPUs, which are becoming common in HPC; and APUs (e.g., AMD's Fusion product line which combines CPU and GPU logic on the same chip), which are emerging. It is imperative that language, compiler, and runtime support for accelerators enable programmers to develop codes using a natural programming style that is independent from the characteristics of today's accelerators; otherwise, we will be faced with an awful porting task as processor designs evolve in the future.

For the coming exascale era, programming models will also need to provide support for fault tolerance; something better than simple program-wide checkpoint/restart is clearly needed if the expected interval between processor failures is on the order of hours rather than weeks. We would like to investigate using teams (process subsets) and coarrays mapped across their members as a mechanism for implementing components for finer grain checkpointing and recovery.

Our overarching goal has been and continues to be to develop a suitable model for writing scalable and high performance programs that is useful for programmers at all points on the application spectrum, from desktop workstations to supercomputers. This requires attracting application programmers, who have been historically skeptical about adopting anything not a standard. To have a practical impact, we have focused our efforts on Fortran, which is the dominant programming language today for HPC. It is thus crucially important to prove the viability of CAF 2.0 features with further language prototyping and evaluation to show that (1) it is practical to implement the rich set of features in CAF 2.0, (2) these features are necessary and sufficient for expressing a wide variety of important algorithms, and (3) these features can be implemented in a way that delivers high performance across the entire range of computer sizes. Without that we will be unable to provide an authoritative argument that they should be incorporated into the Fortran standard. Having CAF 2.0 features adopted into the Fortran standard will be critical in moving the community forward.

References

[1] A. Petitet and R.C.Whaley and J.Dongara and A.Cleary. HPL - A Portable Implementation of the High-Performance Linkpack Benchmark for Distributed-Memory Computers, September 10, 2008. `http://www.netlib.org/benchmark/hpl`.

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

[3] L. Adhianto, J. Mellor-Crummey, G. Jin, K. Murthy, D. Nguyen, W. N. Scherer III, S. Warren, and C. Yang. A critique of ISO/IEC JTC1/SC22/WG5 N1835 (addition/modification of CAF features). Fortran Standards Technical Committee Document N1856, June 2011. ftp://ftp.nag.co.uk/sc22wg5/N1851-N1900/N1856.txt.

[4] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A high performance parallel algorithm for 1-D FFT. In *SC '94: Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, pages 34–40, Washington, D.C., 1994. IEEE Computer Society Press.

[5] G. Almási et al. IBM's 2009 submission to the HPC Challenge class 2 competition Unified Parallel C(UPC) and X10, 2009.

[6] G. Bikshandi, G. Almási, S. Kodali, I. Peshansky, V. Saraswat, and S. Sur. A comparative study and empirical evaluation of global view high performance Linpack program in X10. In *PGAS '09:*

Proceedings of the Third Conference on Partitioned Global Address Space Programing Models, pages 1–9, NY, NY, USA, 2009. ACM.

[7] J. Bruck, C.-T. Ho, S. Kipnis, and D. Weathersby. Efficient algorithms for all-to-all communications in multi-port message-passing systems. In *Proceedings of the* 6^{th} *Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 298–309, NY, NY, USA, 1994. ACM.

[8] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, and D. Iten. HPC Challenge benchmarks in Chapel, 2009. `http://chapel.cray.com/hpcc/hpcc09.pdf`.

[9] B. L. Chamberlain, S. J. Deitz, S. A. Figueroa, D. M. Iten, and A. Stone. Global HPC Challenge benchmarks in Chapel, 2008. `http://chapel.cray.com/hpcc/hpccOverview-2.1.pdf`.

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

[11] R. Garg and Y. Sabharwal. Software routing and aggregation of messages to optimize the performance of HPCC randomaccess benchmark. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, NY, NY, USA, 2006. ACM.

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

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

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

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

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

[17] 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.

[18] 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.

[19] A. Moody, D. H. Ahn, and B. R. de Supinski. Exascale algorithms for generalized `mpi_comm_split`. In *EuroMPI 2011*, 2011.

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

[21] S. J. Plimpton, R. Brightwell, C. Vaughan, K. D. Underwood, and M. Davis. A simple synchronous distributed-memory algorithm for the HPCC RandomAccess benchmark. In *Proceedings of the 2006 IEEE International Conference on Cluster Computing*, Sept. 2006.

[22] J. Reid and R. W. Numrich. Co-arrays in the next Fortran standard. *Sci. Program.*, 15(1):9–26, 2007.

[23] P. Sack and W. Gropp. A scalable `mpi_comm_split` algorithm for exascale computing. In R. Keller, E. Gabriel, M. Resch, and J. Dongarra, editors, *Recent Advances in the Message Passing Interface*, volume 6305 of *Lecture Notes in Computer Science*, pages 1–10. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-15646-5_1.

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

[25] 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.

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

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

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