2011 IEEE International Parallel & Distributed Processing Symposium

Implementation and Performance Evaluation of the HPC Challenge Benchmarks in Coarray Fortran 2.0

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

***Abstract*—Today's largest supercomputers have over two hundred thousand CPU cores and even larger systems are under development. Typically, these systems are programmed using message passing. Over the past decade, there has been considerable interest in developing simpler and more expressive programming models for them. Partitioned global address space (PGAS) languages are viewed as perhaps the most promising alternative.**

In this paper, we report on our experience developing a set of PGAS extensions to Fortran that we call Coarray Fortran 2.0 (CAF 2.0). Our design for CAF 2.0 goes well beyond the original 1998 design of Coarray Fortran (CAF) by Numrich and Reid. CAF 2.0 includes language support for many features including teams, collective communication, asynchronous communication, function shipping, and synchronization. We describe the implementation of these features and our experiences using them to implement the High Performance Computing Challenge (HPCC) benchmarks, including High Performance Linpack (HPL), RandomAccess, Fast Fourier Transform (FFT), and STREAM triad. On 4096 CPU cores of a Cray XT with 2.3 GHz single socket quad-core Opteron processors, we achieved 18.3 TFLOP/s with HPL, 2.01 GUP/s with RandomAccess, 125 GFLOP/s with FFT, and a bandwidth of 8.73 TByte/s with STREAM triad.

***Keywords*-Parallel programming; performance analysis**

I. Introduction

Today, scalable parallel systems are principally programmed using the Message Passing Interface (MPI) [1]. Over the past decade, there has been considerable interest in developing higher level models for parallel programming. Partitioned global address space (PGAS) languages based on one-sided communication, such as UPC [2], Coarray Fortran [3], X10 [4], Chapel [5], Titanium [6], and Fortress [7] are viewed as perhaps the most promising class of programming languages for scalable parallel systems. A recent addition to the PGAS family of languages is XcalableMP [8]; this directive-based programming model provides support for simple global view computations in addition to its support for local view computations.

In this paper, we focus on Coarray Fortran (CAF). Originally designed by Numrich and Reid [3], CAF is a small set of extensions to Fortran 95 to support parallel programming. They envisioned CAF as a model for SPMD parallel programming based on a static collection of asynchronous process images (images for short). In 2005, the Fortran Standards committee began exploring the addition of coarray constructs to the emerging Fortran 2008 standard [9]. Their design closely follows Numrich and Reid's original vision.

We have extended the CAF programming language to create Coarray Fortran 2.0 (CAF 2.0) [10]. CAF 2.0 is a refinement of coarray features proposed in the working draft for Fortran 2008. CAF 2.0 is expressive. It provides a natural way to express a broad spectrum of algorithms, to construct efficient parallel data structures, and to support both static and dynamic strategies for decomposing work. CAF 2.0 is designed to increase programmer productivity: programs can be easily written without sacrificing performance across the spectrum of computing platforms ranging from commodity clusters to leadership-class systems. Both the language and the runtime are also carefully designed to be portable and scalable to leadership-class computer systems with tens to hundreds of thousands of processors.

The High Performance Computing Challenge (HPCC) benchmark suite [11] was developed as part of DARPA's High Productivity Computer Systems (HPCS) program to measure the performance of high performance computer systems from multiple perspectives: processor, memory subsystem, and system interconnect. For this reason, the benchmarks have been widely used to evaluate the efficiency of a range of systems from hardware architecture [12], [13] to programming languages [14], [5], [4], [15]. To evaluate the suitability of CAF 2.0 for writing scalable high performance parallel programs, we used it to implement four of the seven HPCC benchmarks—High Performance Linpack (HPL), RandomAccess, Fast Fourier Transform (FFT), and STREAM triad—and measured the performance of our resulting implementations.

The contributions of this paper are threefold. First, we describe our efficient and scalable implementation of the CAF 2.0 runtime system. Second, we present the implementation of four HPCC benchmarks in CAF 2.0. Third, we evaluate the performance of these benchmarks running on thousands of cores on a leadership computing platform.

The remainder of this paper is organized as follows. In Section II, we briefly describe CAF 2.0 and our runtime system implementation to provide context for the rest of the paper. Section III introduces our implementations of HPCC benchmarks with CAF 2.0. In Section IV, we discuss the

1530-2075/11 $26.00 © 2011 IEEE
DOI 10.1109/IPDPS.2011.104

performance of our benchmark implementations. Section V presents related work. We conclude in Section VI and describe open issues for future work in Section VII.

II. CAF 2.0 Features and Implementation

A good programming language should be simple to use yet provide features that enable programmers to implement complex applications. CAF 2.0 was designed in this perspective. Coarray Fortran is simple: it enables a programmer to write parallel programs as modest code modifications to a serial program. With CAF 2.0, it is also possible to write a sophisticated program that hides communication latency by using asynchronous copies and collectives. CAF 2.0 includes critical features that are needed for productivity, such as process subsets (teams). Teams in our design serve as a domain onto which coarrays are allocated, provide a namespace within which coarray images can be indexed by rank, and define a process set for collective communication. CAF 2.0 provides a rich set of features that are useful for writing high performance codes. These include collective communication, lightweight synchronization, and asynchronous operations for latency hiding. Table I lists a collection of features that we have implemented in CAF 2.0.

We have constructed a prototype source-to-source compiler for CAF 2.0 using the ROSE compiler [16] developed by Lawrence Livermore National Laboratory. It translates CAF 2.0 code into Fortran code that can be compiled by native compilers. Advantages of using source-to-source compilation include portability and the ability to leverage machine-dependent optimizations available in native compilers. To support portability to many platforms, our runtime system uses the Global Address Space Networking (GASNet) library [17], developed by University of California at Berkeley, as a communications substrate. However, since our runtime library provides its own data structures to manage process image subsets, synchronization, and collective communication, it uses only a small fraction of the GASNet functionality — mainly get/put and active messages. As future work, we plan to extend our runtime implementation to use native communication libraries directly.

A. Teams

CAF 2.0 supports process subsets (*team*s), which are a useful abstraction for decomposing work in a parallel application. Teams can be used to partition and organize work among a collection of process images. For instance, teams can be used to organize subsets of process images performing a dense matrix computation calculation into row and/or column teams, or to map different calculations onto process image subsets in a coupled application.

In CAF 2.0, we have designed teams to scale to tens or hundreds of thousands of processor cores. We use a team representation that requires only $O(\log P)$ storage on each core of P processor cores (see Figure 1). If a process image

Table I
List of CAF 2.0 language features.

1-sided Support	Sync	Team	Collective	Asynchronous Support
`get`* `put`*	`barrier`* `event` `eventset` `lock`* `lockset`	`split` `free`	`broadcast` `gather` `allgather` `reduce` `allreduce` `scan` `scatter` `shift` `sort` `alltoall` `permute`	`copy` `spawn` `broadcast` `barrier`

* Primitives supported by CAF 1.0: 1-sided communication and some basic synchronization.

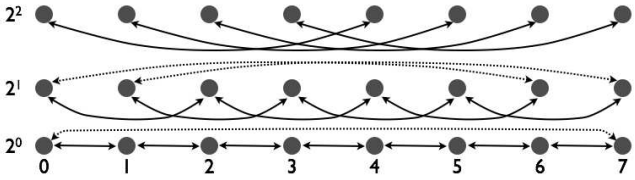

Figure 1. Members of a team of size S are linked in $\lceil \log S \rceil$ doubly-linked circular lists. In list i, $0 \leq i < \lceil \log S \rceil$, a team member at rank j is linked to team members $(j + S - 2^i) \bmod S$ and $(j + 2^i) \bmod S$, an organization inspired by pointer jumping [18].

j needs to contact a process image k ($k \neq (j \pm 2^i) \bmod P$), it determines the target process image with the help of tables distributed over members of the team and memoizes the result. (This approach is explained in Section II-B.) This strategy yields teams with scalable space requirements yet with performance comparable to a representation that includes information about each team member on every image using $O(P^2)$ space.

In CAF 2.0, when a program is launched, all process images belong to `TEAM_WORLD` (similar to `MPI_COMM_WORLD` in MPI). Process image subsets are formed using the `team_split` operation [10] inspired by `MPI_COMM_SPLIT`. Our approach enables us to form a team without using no more than $O(\log^2 P)$ space on any process image, where P is the number of process images in the team. Figure 2 shows an example of using the CAF 2.0 team and `with team` features, creating subteams with `team_split`, and allocating coarrays within a team. Rather than requiring a programmer to specify a team explicitly for each operation, we have introduced the concept of a default team. One can specify a new default team using a `with team` block. `with team` blocks are dynamically scoped and may be nested. The CAF 2.0 runtime maintains a stack of default teams that it pushes and pops at entry and exit of each `with team` block.

The default team is used any time a team is not specified; for example, the `team_size()` function in line 5 of

Table II
CAF 2.0 KEY FEATURES USED IN HPCC.

	STREAM	RandomAccess	FFT	HPL
Team, team formation	team*	team*	team*	team,split,free
Synchronization	barrier	event,barrier	event,barrier	event,barrier
One-sided communication		get,put	get,put	get,put
Collective communication		reduce	alltoall	reduce,broadcast
Async. communication		copy	copy	broadcast
Function shipping		spawn		

* STREAM, RandomAccess, and FFT use only the default team, TEAM_WORLD.

```
 1 team :: rowteam, colteam
 2 integer :: myrow, mycol, me, nprocs, nprow, ierr
 3 double precision, allocatable :: w(:)[*]
 4
 5 nprocs = team_size() ! get the number of images
 6 me = team_rank()     ! get my image ID
 7 mycol = me / nprow
 8 myrow = me - mycol * nprow
 9
10 ! create row and column teams
11 call team_split(TEAM_WORLD,myrow,mycol,rowteam,myrow)
12 call team_split(TEAM_WORLD,mycol,myrow,colteam,mycol)
13
14 ! allocating array w within colteam
15 with team colteam
16   allocate(w(2*BLKSIZE+4)[])
17 end with team
```

Figure 2. HPL code snippet that shows teams primitives team_size, team_rank, and team_split; and coarray allocation within team.

Figure 2 retrieves the number of image processes in the default team which is equivalent to TEAM_WORLD. However, in line 16 of the same figure, the allocation of coarray w is carried out by images within the column subteam (colteam) which in turn was set as the default team by the with team block on line 15.

As shown in Table II, our implementations of four HPCC benchmarks all require access to team properties such as the number of images and the rank of image within a team. In Table II, we list key features of CAF 2.0 that we have used in our implementations of the four HPCC benchmarks. We discuss features of collective operations and asynchronous operations later in this section.

B. Mapping team ranks to process images

For scalability, our implementation of teams and coarrays avoids replicating information unnecessarily. Each member of a team only has direct knowledge of $O(\log P)$ team members at distance 2^i, where $i \geq 0$ and P is the number of process images in the team. If a process image j needs to contact a process image k ($k \neq (j \pm 2^i) \bmod P$), it does this by inspecting a chain of pointer-jumping links. The number of pointer-jumping lookups in the worst case is the number of '1' bits in $(j - k + P) \bmod P$.

Using the minimum length chain of pointer-jumping lookups to find the communication partner of a process image within a team is important for minimizing communication overhead. We implemented an algorithm for computing

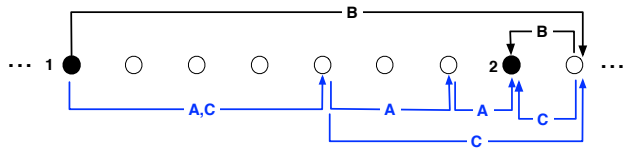

Figure 3. Three alternative lookup chains for locating a remote image at distance seven.

the minimum length chain of *forward* and *backward* pointer-jumping lookups to reach a target. A forward (or backward) lookup by a process image requires 1-sided communication between the process image and its communication partner at a positive (negative) relative distance around a ring. This is accomplished by recursively choosing to move to a successor or a predecessor on the ring at some power-of-two distance that is closer to the target rank. At each level of the recursion, we explore both forward and backward jumps. Our algorithm uses dynamic programming to select the shortest chain of jumps to reach the target.

Figure 3 shows three possible lookup chains A, B, and C from node 1 to node 2 at distance seven. Chain A consists of three forward lookups with distances four, two, and one. Chain C uses two foward lookups of distance four and a backward lookup of distance one. Chain B, which uses a forward lookup at distance eight followed by a backward lookup at distance one, requires the fewest lookups.

C. Coarray caching for efficient lookup

To cache addresses of coarrays on remote process images, we implemented a two-level data structure with splay trees [19] at each level. A splay tree is a data structure that supports fast access to elements that are recently accessed, with $O(\log n)$ amortized time for basic operations such as insertion, lookup, and removal in an n-element tree. In our data structure, the first level splay tree maps a coarray handle to a second level tree and the second level tree maps a process image to a coarray address. A lookup function for coarray c is initiated before coarray operations such as coarray read or write; the coarray handle for c, H_c, and remote image rank I are arguments to the lookup. If a process image cannot find a two-level mapping for H_c in its data structure, an active message is initiated to to request $A_{c,I}$ (the coarray's address on partner image I) from partner

image I, where the association of the coarray's handle and its local address was created at the time the coarray was allocated. Upon receiving a reply to a remote lookup, a process image then caches the mapping $H_c \rightarrow (I \rightarrow A_{c,I})$ locally to avoid future remote inquiries.

Farreras, Almási, Cascaval and Cortes developed a similar strategy for achieving scalable RDMA performance [20]. They used a remote address cache to enable small message transfers via RDMA for a shared variable directory in their UPC implementation.

D. Collective communication

Collective communication operations, known simply as *collectives*, are one of the most common communication patterns in parallel programming. Using collectives, one can perform complex communication and synchronization patterns such as broadcast, reductions, barrier synchronization, scatter, and gather in $\lceil \log P \rceil$ rounds on P process images. As shown in Table I, CAF 2.0 provides a broad set of collective operations that are carefully designed for efficiency. Our implementation of `team_barrier` uses a dissemination algorithm [21]; our implementations of most other collectives, at present, are based on binomial trees, though other approaches are possible.

E. Asynchronous operations

To hide communication latency, CAF 2.0 includes support for asynchronous point-to-point and collective operations. We have implemented an asynchronous progress engine that provides generalized support for one-sided and two-sided communication (including collectives) by maintaining a queue of "in-flight" operations that need to be tracked and managed. Checking whether asynchronous operations can progress can be done using interrupts, polling, or a combination thereof. Our current implementation uses a polling mechanism. By sprinkling periodic calls to a progress function throughout the runtime code, we enable asynchronous operations to progress within their state machines.

Using our progress engine, we have implemented an asynchronous copy primitive (`copy_async`) which consists of an asynchronous read/write of data followed by calling notify on an associated event to alert the receiver once the data has been transmitted. `copy_async` is particularly useful in pipelined operations where the transmission of one chunk of data can be overlapped with computation on the next chunk. Without using a non-blocking read/write together with a notify operation, the entire pipeline would stall on waiting for the transmission to complete; this would expose the latency of the read/write operation. We use `copy_async` to overlap communication and computation in the main computational loop of the HPCC FFT benchmark.

We have also implemented asynchronous collective operations including barriers, broadcast, and function shipping. To give a sense how these work, we describe our implementation of asynchronous broadcast. An asynchronous broadcast operation is initiated by a call to `team_broadcast_async` and is completed by a call to `event_wait`. After an asynchronous broadcast operation is initiated, each process image starts from an initial state, allocates buffer space if needed, and sends the buffer address to its parent (if any). It then enters a receive state waiting for data to arrive. As soon as the process image receives the data from its parent, it initiates an asynchronous operation for each of its children in the binomial tree. The process image then switches states to wait for completion of these suboperations. In parallel, each suboperation is choreographed by a separate, simple state machine, waiting to receive a child's buffer address and then relaying data to it. Returning to the main state machine, the process image enters a reply state when the suboperations have all completed. It then sends an acknowledgement to its parent. Next, the process image enters another wait state in which it waits for messages from its children indicating that they have finished relaying the broadcast data. Finally, it transitions to a completion state, copies data to its local destination, and calls `event_notify` to signal the end of the operation. We use `team_broadcast_async` to hide latency of panel broadcast in our HPL implementation. Descriptions of the implementation of other asynchronous collectives and function shipping are not included in this paper due to space limitations.

III. 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. In this paper, we evaluate four of seven HPC Challenge benchmarks[1]:

- 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.

Since the CAF 2.0 prototype only recently became operational, achieving high performance is a work in progress. To date, we have been experimenting with the HPC Challenge

[1]We have focused on these four because they are the ones measured for the HPCC performance awards.

benchmarks and using their performance issues to help focus our implementation efforts. Below comments about our experiences thus far in implementating these benchmarks.

A. 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 [22] 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 [23], [24], [25]. In this section, we highlight features of our implementation in CAF 2.0.

1) 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. As shown in Figure 2, to support row- and column-wise communication in HPL, we created both row and column teams. Subteams are further created when it is necessary.

2) 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 4 shows a code snippet for the LU factorization.

```
 1 integer,target,allocatable :: panelinfo(:,2)[*]
 2 double precision,target,allocatable :: panel(:,2)[*]
 3 event,allocatable,dimension(:) :: delivered[*]
 4
 5 allocate(delivered(1:NUMPANELS)[])
 6 event_init(delivered, NUMPANELS)
 7 ...
 8 do j = pp, PROBLEMSIZE - 1, BLKSIZE
 9  cp = j / BLKSIZE + 1
10  cp = mod(cp - 1, 2) + 1
11  event_wait(delivered(3-cp))
12  if (mycol == cproc) then
13   ncol = localsize(min(BLKSIZE,PROBLEMSIZE-j), &
14                    j, BLKSIZE, NPCOL, mycol)
15   if (ncol > 0) then
16    if (NPCOL==1) call update(m,n,BLKSIZE,ncol,3-cp)
17    if (NPCOL/=1) call update(m, n, 0, ncol, 3 - cp)
18   end if
19   call fact(m, n, cp)
20   col = col + ncol
21  end if
22  if (mycol == pproc) col = col + BLKSIZE
23  ub = (panelinfo(5,cp)+BLKSIZE+1)*BLKSIZE
24  call team_broadcast_async(panel(1:ub,cp), &
25                            panelinfo(8,cp), &
26                            delivered(cp),rteam)
27
28  if (nn-ncol>0) call update(m,n,col,nn-ncol,3-cp)
29
30  if (mycol == cproc) nn = nn - BLKSIZE
31  pproc = cproc
32  cproc = mod(cproc+1, NPCOL)
33 end do
```

Figure 4. LU factorization in the HPL benchmark.

B. 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.

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 5.

C. 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

```
 1 double precision, allocatable :: a(:)[*]
 2 double precision, allocatable :: b(:)[*], c(:)[*]
 3 ! allocate with the default team
 4 allocate(a(ndim)[], b(ndim)[], c(ndim)[])
 5 ...
 6 do round = 1, rounds
 7   do j = 1, rep
 8     call triad(a,b,c,n,scalar)
 9   end do
10   call team_barrier()
11 end do
12 ...
13 subroutine triad(a, b, c, n ,scalar)
14   double precision a(n), b(n), c(n), scalar
15   a = b + scalar * c
16 end subroutine triad
```

Figure 5. Implementation of the STREAM benchmark.

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.

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 [26]. A sketch of the implementation is shown in Figure 6. 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 [27]. IBM also explored a software routing strategy for this benchmark on Blue Gene systems [28].

D. 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:

$$Z_k \leftarrow \sum_j^m z_j e^{-2\pi i \frac{jk}{m}}; 1 \leq k \leq m$$

Performance of the FFT benchmark is measured in GFLOP/s, with calculated performance defined as

```
 1 event,allocatable :: delivered(:)[*],received(:)[*]
 2 integer(8),allocatable :: fwd(:,:,:)[*]
 3
 4 do i = world_logsize-1, 0, -1
 5   ...
 6   call split(ret(:,last), retsizes(last), &
 7        ret(:,current), retsizes(current), &
 8        fwd(1:,out,i),fwd(0,out,i),bufsize,dist)
 9
10   if (i < world_logsize-1) then
11     event_wait(delivered(i+1))
12     call split(fwd(1:,in,i+1), fwd(0,in,i+1), &
13          ret(:,current), retsizes(current), &
14          fwd(1:,out,i),fwd(0,out,i),bufsize,dist)
15     event_notify(received(i+1)[from])
16   endif
17
18   copy_async(fwd(0:outgoing_size,in,i)[partner], &
19              fwd(0:outgoing_size,out,i), &
20              delivered(i)[partner], received(i))
21   ...
22 end do
23
24 ! each process image applies its local updates
25 ......
```

Figure 6. Implementation of a routing algorithm in RandomAccess.

$5\frac{m\log_2 m}{t}10^{-9}$, 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 [29], [30], [31]. The reference FFT implementation of the HPC Challenge benchmarks uses a 1D algorithm based on [29].

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 7 shows the main loop body of the remote FFT computation using asynchronous copy.

IV. Evaluation

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.

A. Productivity

One of the key criteria used in the annual HPC Challenge awards competition [11] is the number of source lines in the implementation of each benchmark. Table III 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.

```
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 7. Using `copy_async` for hiding communication latency in FFT.

Table III
SOURCE LINES OF CODE (SLOC) OF HPC CHALLENGE BENCHMARKS IN CAF 2.0, AND THEIR COMPARISON WITH CHAPEL [32] AND REFERENCE MPI IMPLEMENTATION OF HPCC [33]. A RATIO BEING LESS THAN 1 MEANS THAT THE CAF 2.0 IMPLEMENTATION IS SMALLER.

	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 III also compares the total source lines of code between CAF 2.0, the Chapel [32] implementations, and the reference MPI implementations of HPCC [33]. 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 productivity 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.

B. 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".

1) 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.

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 8.

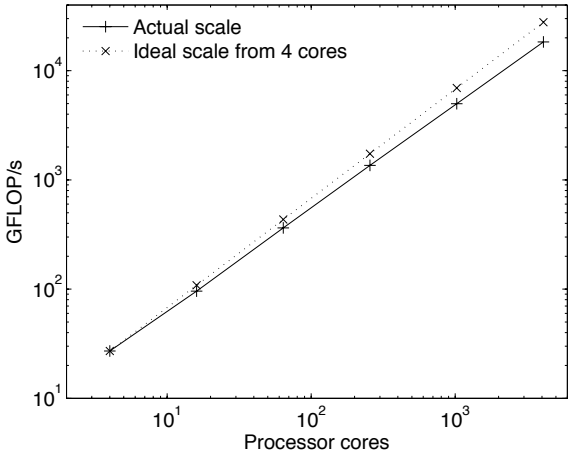

Figure 8. Performance results of HPL.

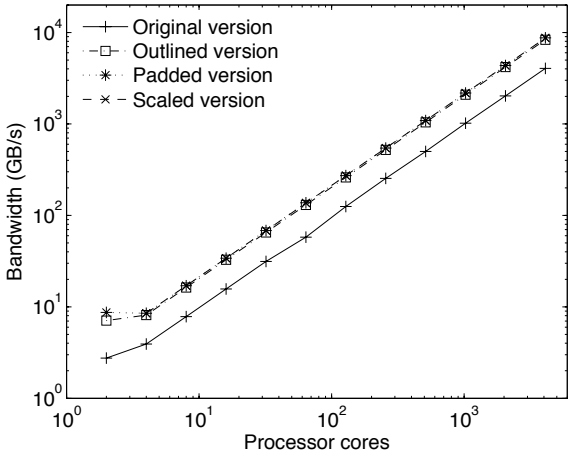

Figure 9. Performance results of STREAM.

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 [34] 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.

2) 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 9. 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 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.

3) RandomAccess: The performance of our CAF 2.0 implementation of RandomAccess running on a Cray XT4 is shown in Figure 10. 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.

4) FFT: When we analyzed our initial implementation of FFT with Rice University's HPCToolkit performance tools [34] 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

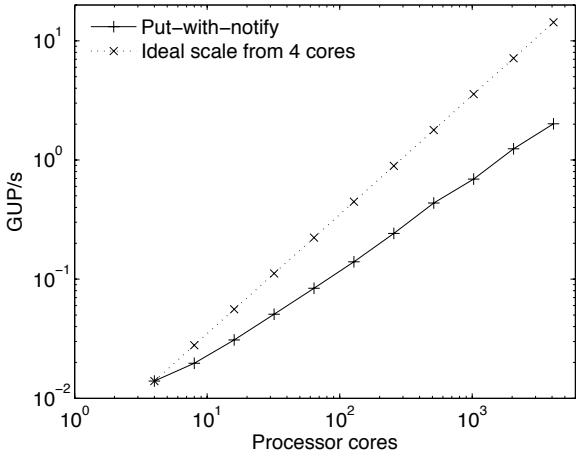

Figure 10. Performance results of RandomAccess.

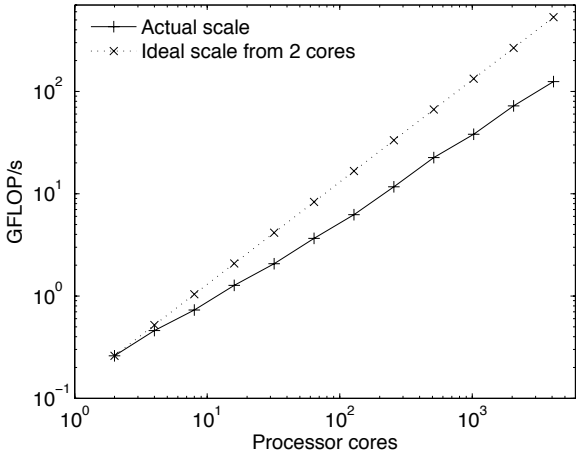

Figure 11. Performance results of FFT.

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 [35] 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 elementwise 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 11, 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.

C. Comparison with other PGAS implementations

1) 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 [11].[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.

2) 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

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

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) [32]. 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 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.

3) 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 much 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 also performed more efficiently per node than the Class 1 result, which achieved 398 TByte/s on 223112 cores.

V. Related work

The HPC Challenge benchmark suite has been implemented in many languages, such as MPI [1], Cilk [36], Chapel [5], [32], pMatlab [14], XcalableMP [8], Unified Parallel C (UPC) [37] and X10 [4].

Cilk is an extension of C and is a programming model for shared-memory hardware. It was awarded for best combination of elegance and performance in 2006. Although its HPCC implementation is concise and simple, at the time its scaling was limited by the lack of a shared-memory system with more than 512 cores. XcalableMP is a directive-based programming language for C and Fortran. The number of source lines of code is very small compared to others. XcalableMP received an honorable mention even though its performance was reported only up to 32 nodes.

Chapel, UPC, and X10 are other PGAS languages competing in the HPC Challenge competition. X10 and Chapel are both new programming language as part of the DARPA-led HPCS program. Chapel HPCC's implementation was awarded as the most elegant in 2009 while Almási et al were awarded best performance for their implementation in UPC and X10 on an IBM Blue Gene/P system [23].

Nishtala, Almási, and Cascaval [38] introduced the notion of instant teams, defined by data-centric operations, which allow for dynamic team construction in UPC. In CAF 2.0, the processor dimension is explicit for coarrays; thus, a data-centric specification would require explicitly enumerating processors, which is awkward or worse in the face of complex user-defined mappings. Hence, CAF 2.0 only supports teams that are assembled in advance.

Hoefler, Lumsdaine, and Rehm [39] explored implementations of non-blocking collective operations for MPI. For this purpose, they developed libNBC—a library that implements asynchronous collective operations using non-blocking point-to-point operations. libNBC provides primitives to schedule a set of non-blocking send and/or receive events to be performed by a processor into a sequence of "rounds". Each round consists of a set of operations that can be performed in parallel. A round is complete when all of its component operations are complete. A schedule for a non-blocking collective is stored as a linear array and the schedule is interpreted by a progress engine. Unlike libNBC, our implementation of asynchronous operations allows us to organize independent sequences of component operations with each communication partner; once these threads are created, they run independently to completion, which avoids unnecessary serialization.

Hoefler and Lumsdaine evaluate various implementation strategies for managing the progress of asynchronous operations [40]. They found that using a separate progress thread works best when a dedicated core is available. Without a dedicated core, they found that a separate progress thread provides only limited benefits and only if real-time scheduling of the progress thread is used.

VI. Conclusions

Our experiences developing and evaluating implementations of the HPC Challenge benchmarks show that CAF 2.0 is a viable PGAS programming model for scalable parallel computing. CAF 2.0 produces scalable performance comparable to the best PGAS language implementations

showcased in the HPC Challenge awards competition. We have successfully scaled CAF 2.0 to thousands of processor nodes with a rich programming model. This achievement has two components. First, our scaling results show that our language constructs and runtime implementation support implementation of scalable programs. Second, the total performance of our codes, which reflects not only scalability but also node performance, shows that our source-to-source translation of CAF 2.0 enables us to achieve a significant fraction of peak node performance with our codes.

VII. Future work

There is much remaining to do in the implementation of CAF 2.0. After adding full support for coarrays of user-defined types, other major tasks include finishing support for asynchronous collective operations, adding support for multithreading, and defining a precise memory model.

This will yield a design that is not only simpler, but that also moves each independent communication sequence along as expeditiously as possible.

Currently, the progress engine in the CAF 2.0 runtime is invoked whenever a program initiates a communication or synchronization operation. As future work, we plan to experiment with alternative strategies for invoking the CAF 2.0 progress engine more eagerly. Three possibilities include: (1) having the compiler sprinkle calls to the progress engine throughout CAF 2.0 programs; (2) using the interval timer to invoke the progress engine whenever it has not been invoked sufficiently recently; and (3) using a dedicated progress thread. Potential drawbacks of a naive implementation of the first strategy include having no progress engine invocations during calls to long-running library operations or having excessively frequent invocations of the progress engine add overhead. The third strategy may be attractive in cases where extra cores are available that could not be profitably tasked with additional computation threads; past work has shown that without a spare core, this approach may not be the best [40]. A combination of the first and second approaches seems most promising. For developers who desire finer-grain control than offered by our automatic management of progress engine invocations, we plan to add a directive to allow users to explicitly enable/disable automatic invocation of the progress engine during important code regions.

Finally, though our CAF 2.0 runtime uses the GASNet communication interface directly, we intend to introduce an intermediate abstraction layer in our implementation of the runtime system. This will enable us to port our runtime system directly to low-level communication layers built directly upon hardware support.

Acknowledgments

Development of Coarray Fortran 2.0 is supported by the Department of Energy's Office of Science under cooperative agreements DE-FC02-07ER25800 and DE-FC02-06ER25754. This research used resources of the National Energy Research Scientific Computing Center (NERSC) at Lawrence Berkeley National Laboratory and the National Center for Computational Sciences (NCCS) at Oak Ridge National Laboratory. Both facilities are supported by the Office of Science of the U.S. Department of Energy. NCCS is supported under Contract No. DE-AC05-00OR22725; NERSC is supported under Contract No. DE-AC02-05CH11231.

References

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

[2] The UPC Consortium, "UPC language specification," http://www.gwu.edu/~upc/publications/LBNL-59208.pdf, May 2005.

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

[4] X10 HPC Challenge, http://x10.codehaus.org/HPC+Challenge, Apr. 2010, last accessed.

[5] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the Chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, 2007.

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

[7] Project Fortress Community, http://projectfortress.sun.com, 2010.

[8] XcalableMP, http://www.xcalablemp.org, 2010.

[9] Fortran J3 Committee, "F2008 Working Document, J3/10-007r1, Nov. 24, 2010." [Online]. Available: http://www.j3-fortran.org/doc/standing/links/007.pdf

[10] 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*. NY, NY, USA: ACM, 2009, pp. 1–9.

[11] HPC Challenge Awards Competition, http://www.hpcchallenge.org, 2010.

[12] S. Alam, R. Barrett, M. Bast, M. R. Fahey, J. Kuehn, C. McCurdy, J. Rogers, P. Roth, R. Sankaran, J. S. Vetter, P. Worley, and W. Yu, "Early evaluation of IBM BlueGene/P," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.

[13] S. Alam *et al.*, "Cray XT4: an early evaluation for petascale scientific simulation," in *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. NY, NY, USA: ACM, 2007, pp. 39:1–39:12.

[14] N. Travinin Bliss and J. Kepner, "pMatlab parallel Matlab library," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 336–359, 2007.

[15] P. Husbands and K. Yelick, "Multi-threading and one-sided communication in parallel LU factorization," in *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. NY, NY, USA: ACM, 2007, pp. 31:1–31:10.

[16] D. J. Quinlan, "ROSE: Compiler support for object-oriented frameworks," *Parallel Processing Letters*, vol. 10, no. 2/3, pp. 215–226, 2000.

[17] D. Bonachea, "GASNet specification, v1.1," U.C. Berkeley, Technical Report UCB/CSD-02-1207, Oct. 2002.

[18] W. D. Hillis and G. L. Steele, Jr., "Data parallel algorithms," *Commun. ACM*, vol. 29, pp. 1170–1183, Dec. 1986.

[19] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *J. ACM*, vol. 32, pp. 652–686, July 1985.

[20] M. Farreras, G. Almási, C. Cascaval, and T. Cortes, "Scalable RDMA performance in PGAS languages," in *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12.

[21] D. Hensgen, R. Finkel, and U. Manber, "Two algorithms for barrier synchronization," *Int. J. Parallel Program.*, vol. 17, pp. 1–17, Feb. 1988.

[22] 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, Sept. 10, 2008," http://www.netlib.org/benchmark/hpl.

[23] G. Almási *et al.*, "IBM's 2009 submission to the HPC Challenge class 2 competition Unified Parallel C(UPC) and X10," 2009. [Online]. Available: http://dist.codehaus.org/x10/hpcc09/IBM_HPCC_2009_submission.pdf

[24] 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*. NY, NY, USA: ACM, 2009, pp. 1–9.

[25] 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.

[26] "HPC Challenge Awards: Class 2 Specification," http://www.hpcchallenge.org/class2specs.pdf, Jun. 2005.

[27] 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.

[28] 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: ACM, 2006.

[29] 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.*, vol. 15, no. 2, pp. 207–228, 2000.

[30] 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*. Washington, D.C.: IEEE Computer Society Press, 1994, pp. 34–40.

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

[32] 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.

[33] "HPC challenge benchmark," http://icl.cs.utk.edu/hpcc, FAQ: http://www.hpcchallenge.org/faq/index.html, last accessed July 13, 2010.

[34] 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.*, vol. 22, no. 6, pp. 685–701, 2010.

[35] 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*. NY, NY, USA: ACM, 1994, pp. 298–309.

[36] B. C. Kuszmaul, "Cilk provides the "best overall productivity" for high performance computing: (and won the HPC challenge award to prove it)," in *Proceedings of the* 19^{th} *Annual ACM Symposium on Parallel Algorithms and Architectures*, San Diego, California, USA, Jun. 2007, pp. 299–300.

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

[38] R. Nishtala, G. Almási, and C. Cascaval, "Performance without pain = productivity: Data layout and collective communication in UPC," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, Feb. 2008.

[39] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and performance analysis of non-blocking collective operations for MPI," in *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. NY, NY, USA: ACM, 2007, pp. 1–10.

[40] T. Hoefler and A. Lumsdaine, "Message Progression in Parallel Computing - To Thread or not to Thread?" in *Proceedings of the 2008 IEEE International Conference on Cluster Computing*. IEEE Computer Society, Oct. 2008.