An Open-Source Compiler and Runtime Implementation for Coarray Fortran

Deepak Eachempati Hyoung Joon Jun Barbara Chapman

Computer Science Department
University of Houston
Houston, TX, 77004, USA
{dreachem, hjun}@uh.edu chapman@cs.uh.edu

Abstract

Coarray Fortran (CAF) comprises a set of proposed language extensions to Fortran that are expected to be adopted as part of the Fortran 2008 standard. In contrast to prior open-source implementation efforts, our approach is to use a single, unified compiler infrastructure to translate, optimize and generate binaries from CAF codes. In this paper, we will describe our compiler and runtime implementation of CAF using an Open64-based compiler infrastructure. We will detail the process by which we generate a high-level intermediate representation from the CAF code in our compilers front-end, how our compiler analyzes and translate this IR to generate a binary which makes use of our runtime system, and how we support the runtime execution model with our runtime library. We have carried out experiments using both an ARMCI- and GASNet-based runtime implementation, and we present these results.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Programming Classifications—Concurrent, distributed, and parallel languages

General Terms Languages, Design

Keywords *Fortran, Coarrays, Compilers, PGAS*

1. Introduction

In the early 1990s, there were efforts to come up with a standard language for creating parallel codes on emerging parallel systems, including High Performance Fortran (HPF) [5, 1] and OpenMP [21]. In HPF, there was a single thread of control, a global address space, and directives for distributing data. But there were challenges to HPF's adoption. Too much burden was placed on the compiler to partition work and generate communication. Moreover, the complexity in implementing the compiler also resulted in inconsistent performance for different vendor implementations. Coarray Fortran (CAF) [20, 23], initially called F^{--}, was proposed in the late 1990s to address some of these challenges. It was a language-based extension to Fortran, defined on top of Fortran 95. The language was designed to be simple for compiler writers to use, in contrast to HPF. CAF is a Partitioned Global Address Space (PGAS) language that supports the SPMD programming style. There have been a few academic and commercial CAF compilers available to date [6, 3, 2, 16]. In CAF, all data and computations are local and must be explicitly decomposed in each executing process or thread. Remote accesses and stores to data are performed with the explicit use of co-subscripts.

In contrast to other prior and existing open-source implementation efforts, our approach is to use a single, unified compiler infrastructure to translate, optimize and generate binaries from CAF codes. We have also developed a runtime system for supporting 1-sided communication between executing processes. Our current approach is to provide this in a manner that is able to exploit both the ARMCI [18, 19] and GASNet [11, 10] communication subsystems. In this manner, we can facilitate a variety of different platform configurations. In this paper we describe our compiler and runtime implementation of CAF using OpenUH, an Open64-based compiler infrastructure. We detail the process by which we generate a high-level intermediate representation from the CAF code in our compilers front-end, how our compiler analyzes and translate this IR to generate a binary which makes use of our runtime system, and how we support the runtime execution model with our runtime library. We have carried out experiments using both an ARMCI- and GASNet-based runtime implementations, and we will present these results in the paper.

The rest of this paper is organized as follows. We briefly discuss the CAF language in Section 2. Then, we describe our CAF implementation using the OpenUH infrastructure in Section 3 and some experimental results presented in Section 4. Related work is described in Section 5, and we close with some concluding remarks and our planned future work in Section 6.

2. Coarray Fortran

A set of proposed language extensions to support CAF are expected to be adopted as part of the Fortran 2008 standard [23]. Its features promise to simplify the task of creating parallel programs by providing syntax with which a programmer can express communication at a high level. Programs containing coarrays will be executed SPMD fashion, where multiple copies of the same program are launched and are referred to as images. Regular arrays are local to an image, so that a Fortran program without coarrays is a sequential program with just one image; coarrays are distributed among the images, which may access elements that are local to other images. In CAF, coarrays are declared with use of square brackets. Remote accesses to coarrays are achieved with co-subscripts enclosed within square brackets which index into the logical space of images on which which the coarray is defined. A coarray reference without co-subscripts refers simply to the local data. Below shows a

PGAS '10 New York, New York USA

simple CAF code performing an all-reduce where each image gets the global maximum of array u across all images.

```
real :: rmax, max_u[*], u(N,N)
...
max_u = maxval(u)
sync all
do i=1,num_images()
  rmax = max_u[i]
  if (max_u < rmax) max_u = rmax
end do
if (this_image() == 1) print *, max_u
...
```

The major benefit of CAF is that the application developer simply uses the co syntax to mark non-local data (coarray element) accesses and thus need not be concerned with the details of data exchange between images. It is the task of the implementation to ensure that this is performed efficiently. The language also includes synchronization statements and intrinsic functions for providing the programmer information about the images and coarrays.

3. Implementation

OpenUH [15] is a branch of the open source Open64 compiler suite for C, C++, and Fortran 77/90/95, with support for the IA-64, IA-32e, and Opteron Linux ABI. Its major functional parts are the front-ends, the inter-language inter-procedural analyzer (IPA), and the back-end which is further subdivided into the loop nest optimizer (LNO), and auto-parallelizer for OpenMP, global optimizer (WOPT), and code generator (CG). OpenUH uses five levels of IR in its back-end, called WHIRL, for facilitating the implementation of different analysis and optimization phases. Most compiler optimizations are implemented on a specific level of WHIRL. OpenUH has also been enhanced to support the requirements of TAU, Kojak, and PerfSuite by supporting an instrumentation API for source code and OpenMP runtime library support [13, 12]. In related prior work, we have used OpenUH to translate OpenMP for clusters, and developed a supporting library based on global arrays and MPI [8].

CAF support in the OpenUH compiler/runtime system comprises three areas, illustrated in Figure 1.

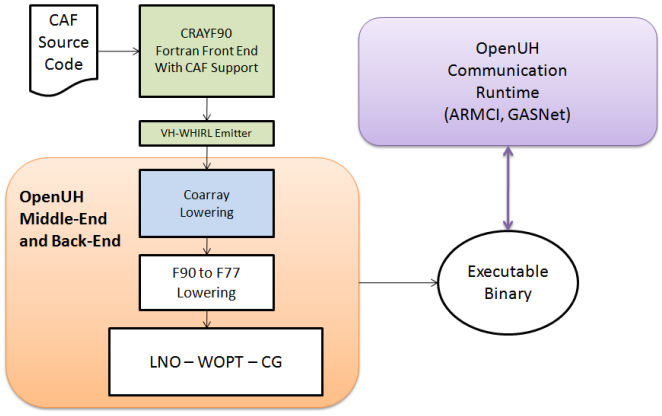


Figure 1. Framework of OpenUH CAF Compiler/Runtime System

First, we extended the OpenUH Fortran front-end to accept coarray syntax and generate intermediate code (IR). Next, we added a translation phase in our compiler that can accept this IR, analyze it, and translate the coarray references to corresponding buffering and communication code. Third, we developed a portable, extensible runtime library that provides the necessary buffering and communication facilities.

3.1 Front-end Processing

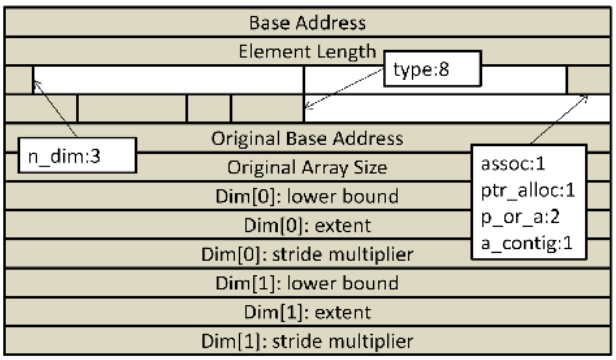
We modified the Cray Fortran 95 front-end used by OpenUH to support our coarrays implementation. Cray had provided some support for CAF syntax, but its approach was to perform the translation to the underlying runtime library in the front-end. It accepted the [] syntax in the parser, recognized certain CAF intrinsics, and it targeted a SHMEM-based runtime with a global address space. Our implementation strategy, looking forward, is to eventually take advantage of the robust static analysis and optimizing capabilities in the OpenUH back-end, so we needed to preserve the coarray semantics into the back-end. To accomplish this, we adopted a similar approach to that used in Open64/SL Fortran front-end from [6], where co-subscripts are preserved in the IR as extra array subscripts. In the `TYPE_TBL` generated by the front-end, we mark arrays declared with the [] syntax as coarrays and distinguish the coarray dimensions from the local array dimensions. We also added support for CAF intrinsic functions such as `this_image`, `num_images`, `image_index`, and more as defined in [24].

In our implementation, all coarrays are allocated at runtime on the heap, even if they are declared explicitly as *allocatable*. A coarray that does not have the `allocatable` attribute exists from the beginning of the program until then end, while an allocatable coarray may be allocated and deallocated during runtime. By default, allocatable arrays are transformed into dope vectors by the front-end, and runtime calls to dynamically allocate or deallocate data for the array are generated based on the `allocate` and `deallocate` statements. We disable this transformation for allocatable coarrays in the front-end, and defer their translation into dope vectors until the back-end where all coarrays are converted to this representation in a single pass.

Special care needs to be taken when a call statement is encountered for a procedure with a coarray argument. We consider two cases for the coarray argument: (1) it has an *explicit shape* declaration, (2) it has an *assumed shape* declaration. For the first case, the front-end will ordinarily assume that the compiler treats explicit shape array argument as a contiguous array. Therefore, if a non-contiguous array section is passed, the front-end uses a temporary buffer and inserts copy-in, copy-out statements before and after the subroutine call. This approach will not work for subroutines with explicit shape *coarray* arguments, because the passed in argument must also be a coarray. In our implementation, an explicit shape coarray argument need not be contiguous, since all coarrays are represented using a dope vector which allows for non-contiguous shapes (see Section 3.2). We added support in the front-end for generating call statements with non-contiguous actual arguments, since ordinarily only a base address is used. In the second case, the front-end will ordinarily represent all assumed shape arrays as dope vectors. Therefore, it will create a dope vector in the calling function, initialize it to point to the section of data that is being passed, and replace the actual argument in the call statement with it. But since we defer the translation of coarrays into dope vectors until the back-end translation, we disabled this transformation as well.

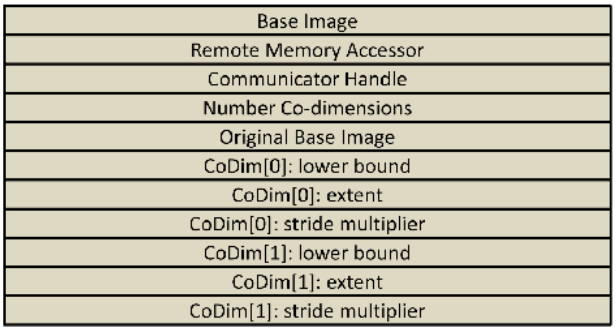
3.2 Coarray Lowering in Back-end

We have currently implemented a basic translation strategy for coarrays in our back-end, with plans to add an analysis/optimization phase prior to the lowering of the coarray representation. Fairly early in the back-end processing, a F90 lowering phase is carried out in which F90-supported elemental array operations are translated into loops. We make use of the higher-level F90 array operations for generating block communications in our translation, so we perform Coarray Lowering prior to F90 lowering. The translation strategy consists of four parts. (1) Each declared coarray is represented by corresponding “dope” and “codope” vectors. The coarrary’s dope vector points to and describes the local coarray data,



...

(a) Dope vector is a local coarray data descriptor



...

(b) Codope vector is a global coarray data descriptor

Figure 2. Dope vector and codope vector for coarray representation

while the coarray's codope vector provides the location of the coarray on other images and describes the cobounds. (2) The parameter list of procedures with coarray arguments is altered to receive a copy of the coarray's dope and codope vectors. (3) We generate communication for any remote coarray references. (4) We translate CAF intrinsics based on the dope and codope vectors.

3.2.1 Coarray Data Representation

Coarrays are represented by the compiler using two data structures: (1) a dope vector for describing the coarray on the local image, (2) a codope vector for describing the coarray across all images, shown in Figure 2. We use the compiler's native dope vector format, defined by Cray, for describing local coarray information. This contains the base address, the base element type, rank, and the array bounds information. There are also flags for indicating whether the array is allocatable or non-contiguous. We add the codope vector in our implementation which, in contrast to the dope vector, contains requisite information for accessing elements of the coarray *across* images.

3.2.2 Communication Generation

We generate communication based on remote coarray references. Suppose the Coarray Lowering phase encounters the following statement:

$$A(i,j,1:n)[q] = B(1,j,1:n)[p] + C(1,j,1:n)[p] + D[p] \quad (1)$$

This means that array sections from coarrays B and C and the coarray scalar D are brought in from image p. They are added together, following the normal rules for array addition under Fortran 90. Then, the resulting array is written to an array section of coarray A on process q. To store all the intermediate values used for communication, temporary buffers must be made available. Our translation creates 4 buffers t1, t2, t3, and t4 for the above statement. We can represent this statement in the following way:

$$A(i,j,1:n)[q] \leftarrow t1 = t2 \leftarrow B(1,j,1:n)[p] + t3 \leftarrow C(i,j,1:n)[p] + t4 \leftarrow D[p] \quad (2)$$

For each expression of the form $t \leftarrow R(...)[...]$, the compiler generates an allocation for buffer t with the same shape as the array section $R(...)$. Next, we create a temporary dope vector describing $R(...)$ and a codope vector describing $R[...]$, based on the R's dope and codope representation. The dope vector for t, along with the dope/codope vectors for $R(...)[...]$ are then passed to the generated `GET` runtime call. This call will retrieve the data into the buffer t using an underlying communication subsystem (either ARMCI or GASNet, as specified by the user). The final step is for the compiler to generate a deallocation for buffer t. An expression of the form $L(...)[...] \leftarrow t$ follows a similar pattern, except the compiler generates a `PUT` runtime call.

```
! omitted creation and
! initialization of dope vectors
GET( t2, B(1, j, 1:n), [p] )
GET( t3, C(i, j, 1:n), [p] )
GET( t4, D, [p] )
t1 = t2 + t3 + t4
PUT( t1, A(i, j, 1:n), [q] )
```

The above pseudo-code depicts the communication pattern generated for the statement representation given in (2). Currently, all generated communication is blocking, and we have not yet implemented optimizations for the buffering.

3.3 Supporting CAF in Communication Runtime System

The implementation of our supporting runtime system relies on an underlying communication subsystems provided by ARMCI or GASNet. We have adopted both the ARMCI and GASNet libraries for most communication and synchronization operations required by the CAF execution model. We describe in this section memory management for coarray data, communication facilities provided by the runtime, and support for synchronizations specified in the CAF language.

3.3.1 Coarray Memory Management

All coarray data is dynamically allocated at runtime, regardless of whether it is defined with the `allocatable` attribute. Coarrays not declared as allocatable, which we call non-allocatable coarrays, must be resident in memory throughout the run of the program. Coarrays that are declared as allocatable may be allocated and deallocated at any time during the run of the program. The function of the runtime's memory management is to (1) allocate memory regions, in synchrony with other images, for holding coarray data in a globall accessible "PGAS memory", and (2) assign coarrays to contiguous locations in these memory regions. The runtime tracks collectively allocated memory regions reserved for coarrays using

a *remote memory descriptor*. Each image has a copy of the same memory descriptor, and it provides the base address for the memory region on each process, the total size of the memory region, and the *next offset* for the memory region which indicates the starting address of its unassigned portion minus the base address (Figure 3). Coarrays are always assigned to memory regions in a collective fashion, and the size of the coarray is identical on each image. Consequentially, *next offset* will also always be the same on every image. This means that the offset of a coarray relative to the base address of its memory region will also be the same on all images. Thus, given a coarray's starting address and its remote memory descriptor, the runtime can compute the starting address for this coarray on any other image.

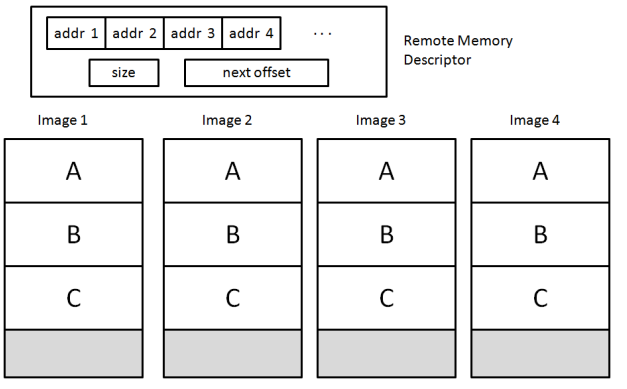


Figure 3. Coarray data allocation across images

In our runtime, all non-allocatable coarrays use the same remote memory descriptor; that is, they belong to the same memory region allocated at the beginning of the program. On the other hand, allocatable coarrays each have a unique remote memory descriptor, allowing for independent allocation and deallocation. Unlike ARMCI, GASNet only allows the allocation of global-accessible memory to occur once. In order to support allocatable coarrays in GASNet, we allocated a large chunk of the PGAS memory at the start of the program and used our own storage allocator for dynamic allocation and deallocation of space for coarray data.

ARMCI does not guarantee aligned memory segment across processes running, while GASNet does so on systems with a small number of nodes. In addition, GASNet is able to initiate the aligned offset addresses of PGAS memory across processes with the *disable-aligned-segments* option, which is desirable for systems with a large number of nodes [11]. For instance, using this option, runtime system developers do not need to compute the differences of these addresses to access coarray data on remote addresses.

3.3.2 Strided 1-sided Communication for Coarray

Our runtime provides 1-sided, strided *get* and *put* routines [4], necessary for supporting basic cosubscripted coarray accesses. These routines make use of the non-contiguous data transfer facilities provided by the underlying communication layer. The runtime requires Cray dope vectors describing the source and destination, either of which may point to a non-contiguous section of PGAS memory. Additionally, the runtime requires a codope vector which provides the remote image to be accessed and the remote memory descriptor for retrieving the address on the remote image. ARMCI provides middleware developers with two non-contiguous transfer modes: generalized I/O vector and strided [18]. The strided data transfer mode, which is intended to reduce the description of storage for dense multi-dimensional arrays and is optimized for block-wise data transfers, is the one we use in our current implementation. As of now, we have not explored the use of the generalized I/O vector mode for coalescing multiple block-wise communications, but this is a potential enhancement. GASNet also supports the same strided transfer interface as ARMCI and is implemented using Active Messages [9].

3.3.3 Supporting Image Synchronization

CAF defines several synchronization constructs which we support, including `sync all`, `sync images`, `critical`, and `notify/ query` (note: `notify/query` is not part of the Fortran 2008 core set for coarrays). The `sync all` statement synchronizes all images collectively. For our ARMCI implementation, we simply invoke *ARMCI_Barrier*, which in turn is a wrapper for *MPI_Barrier*. GASNet provides split-phased barriers which may be used to increase communication overlap with computation, though at present we do not take advantage of such potential benefits. The `sync images` statement, which takes in an array of image IDs as an argument, is used to specify a set of images which the executing image will synchronize with. The runtime maintains distributed array *syncSet*, where if *syncSet[i]==1* that means image *i* is attempting to synchronize with the executing image. So for instance, if image i is to synchronize with image j, it will set *syncSet_j[i]* (*syncSet[i]* on image *j*) to 1. The `sync images` statement is implemented, as shown in 1 using the *Lock* and *Unlock* mechanisms provided by ARMCI and GASNet. *Lock(p,q)* means acquire lock for mutex *p* on image *q*, and similary *Unlock(p,q)* means release lock for mutex *p* on image *q*.

SYNC_IMAGES($images$):
{Tell each images that this image is waiting to synchronize}
for each image i in $images$ **do**
  $Lock(this\_image, i)$
  $syncSet_i[this\_image] \leftarrow 1$
end for
{Wait to receive notices from each image that they are waiting to synchronize with this image}
while $AllSync(syncSet, images)$ = $false$ **do**
  $sleep(n)$
end while
for each image i in $images$ **do**
  $Unlock(this\_image, i)$
end for
{Synchronization complete. Reset entries in syncSet array corresponding to images.}
for each image i in $images$ **do**
  $Lock(i, this\_image)$
  $syncSet[i] \leftarrow 0$
  $Unlock(i, this\_image)$
end for

Algorithm 1: Implementing `sync images` statement using the array *syncSet*

The `notify/query` statements allow users to do point-to-point sychronization in their programs. Each image maintains an array *syncNotify* which indicates the number of outstanding notifications it has received from every other image. So, when image *p* notifies *q*, it will acquire a lock on the pth entry of *syncNotify_q* (the array *syncNotify* on image *q*), increment it, and then release the lock (see Algorithm 2). When image *q* issues a query on image *p* it simply waits for at least one notification from image *p* to be received and then acknowledges it by decrementing the associated entry in its *syncNotify* (see Algorithm 3).

3.3.4 Coarray Reductions

We support some coarray reduction operations such as `comaxval` and `cominval` in our implementation, following the CAF exten-

NOTIFY(q):
$p \leftarrow this\_image()$
$Lock(p, q)$
$syncNotify_q[p] \leftarrow syncNotify_q[p] + 1$
$Unlock(p, q)$

Algorithm 2: Notify image q by incrementing syncNotify[this_image] on image q

QUERY(p):
while $syncNotify[p] = 0$ **do**
  $sleep(n)$
end while
$q \leftarrow this\_image()$
$Lock(p, q)$
$syncNotify[p] \leftarrow syncNotify[p] - 1$
$Unlock(p, q)$

Algorithm 3: Query image p for notifications by waiting for syncNotify[p] to be non-zero and then decrementing the notification count

sion set of Fortran 2008. Our implementation works in two modes, depending on whether the reduction it is carriny out a "coscalar" reduction (one element per image) or a non-scalar coarray reduction. As shown in Figure 4, the runtime uses an *all-reduce* algorithm when performing a coscalar reduction. For instance, the reduction `cominval()` returns the minimum value of the variable on all the running images using 1-sided communication. Initially the variable is stored locally, the runtime then copies the value of the variable to a globally shared memory to perform the data transfer between images using 1-sided strided data transfer operations. Whenever the image encounters each barrier, the value of the minimum variable is updated by comparing it with the data on the remote images.

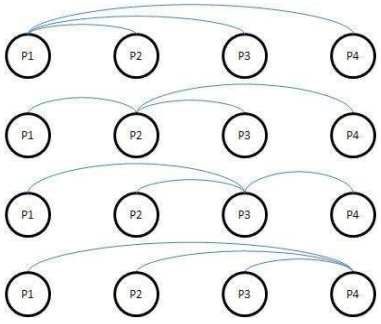
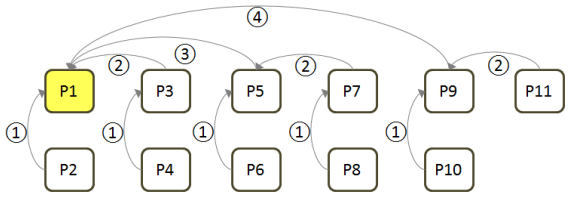
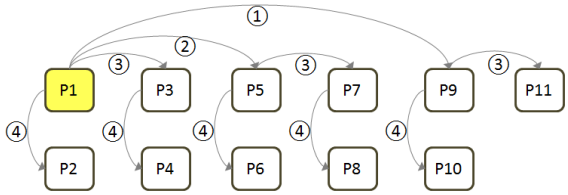


Figure 4. Use of *all-reduce* algorithm for a coscalar reduction across 4 images

For the non-scalar coarray reduction, the CAF runtime supports a few collective operations such as *cominval* and *comaxval* which are based on the binomial tree reduction and broadcast algorithms [22]. A binomial tree reduction is based on recursive distance doubling with copy-in and -out between local and global shared memory for coarray variable (optimization is still underway). As shown in Figure 5.a, first images P_1, P_3, P_5, P_7, P_9 and P_{11} receive data from their remote images at distance 1(=2^0) through 1-sided strided data transfer operations. In the second stage, images P_1, P_5,and P_9 receive data from their remote images at distance 2(=2^1). In the third stage, P_1 receives data from remote image at distance 4(=2^2). Finally, P_1 receives data from P_9 whose distance is 8(=2^3). The result is available only to the root image in this algorithm. Therefore the runtime needs to broadcast this result to other images. Broadcasts are implemented by recursive distance halving [22], that is the



(a) Reduction by recursive distance doubling



(b) Broadcast by recursive distance halving

Figure 5. Illustration of *binomial tree reduction* algorithm for non-scalar coarray reduction on 11 images.

reverse of the reduction operation in Figure 5.b. Due to native characteristics of binomial tree algorithms, non power-of-two number of images may have additional communication overhead with an extra barrier required for reduction and broadcast.

4. Evaluation

All experiments were performed on two cluster systems, called *Medusa* and *Maxwell*. The *Medusa* cluster has 20 nodes connected with InfiniBand (Mellanox MT23108) cards on a single IB switch. Each node contains two 2GHz Opteron processors (246) with 4GB of RAM. The *Maxwell* cluster contains 247 nodes connected with multiple Gigabit Ethernet (Broadcom BCM5715) cards that connect to 6 HP Gigabit switches. Each node contains two or eight 2.2-2.3GHz Opteron processors with 8-16GB of RAM. The operating systems of both clusters are SMP-capable GNU/Linux kernel (Redhat v2.6.18). We used one processor per node on both cluster systems to avoid memory contention between InfiniBand or Gigabit. We used the Open64-based *uhf90* Fortran compiler for the kernel codes that have been re-written in CAF and Fortran (Hand-coded), as well as MPI with optimization level 3. All executable binaries were linked with OpenMPI 1.3.2 library, running on InfiniBand and Gigabit Ethernet systems.

4.1 2D Finite Difference Kernel

We present a preliminary evaluation of our compiler/runtime implementation using a simple 2D Finite Difference kernel which we rewrote in CAF. Finite Difference methods are common numerical methods that allow approximation of solutions to differential equations by replacing the derivatives with approximately equivalent Finite Differences. The algorithm of 2D Finite Difference Kernel is based on a 17-star Finite Difference Stencil in space, solving a typical wave equation with a single source term, constant velocity field, and absorbing boundary condition as illustrated in Figure 7(a) . The kernel is used to perform the forward modeling of the reverse time migration, a popular method used by seismic depth imaging simulation.

As shown in Figure 6a, the sequential code has a subroutine, *cg_fwd_2d* that receives two arrays as arguments, *u* and *v* . In the main loop, a result from the *csource* function is added to the center of array *u*, then *cg_fwd_2d* is invoked. Each element of the *u* array is updated based on the corresponding element of the *v* array and

the four elements that fall above, below, to the left, and to the right of the corresponding element of array *v*. The next value from the source is added to the center of the array *v*. Finally *cg_fwd_2d* is called again and array *v* is updated based on *u*. This step is repeated 500 times.

```
real :: u(xmin-lx:xmax+lx, zmin-lz:zmax+lz)
real :: v(xmin-lx:xmax+lx, zmin-lz:zmax+lz)
...
do  it=1,nt, 2
  print *, maxval(u), minval(u), source(it)
  u(xsource,zsource) = u(xsource,zsource) + &
                       source(it)
  call cg_fwd_2d( u, v, ... )
  v(xsource,zsource) = u(xsource,zsource) + &
                       source(it+1)
  call cg_fwd_2d( v, u, ... )
end do
```

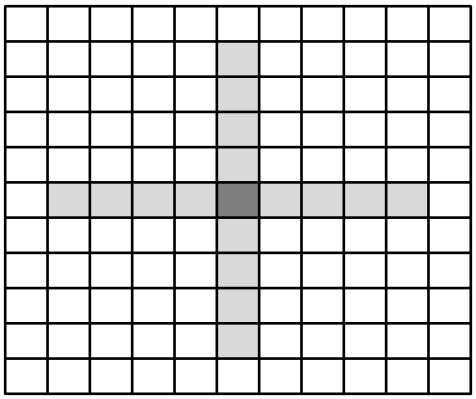
(a) Sequential version

```
real :: u(xmin-lx:xmax+lx,zmin-lz:zmax+lz)[npx,*]
real :: v(xmin-lx:xmax+lx,zmin-lz:zmax+lz)[npx,*]
...
do  it=1,nt, 2

sync all
call co_maxval( maxval(u), u_max)
call co_minval( minval(u), u_min)

if (this_image()==1)  print *, u_max, u_min
if (is_center_image())  &
   u(xsource,zsource) = u(xsource,zsource) + source(it)
call cg_fwd_2d( u, v, ...)

! get data from top neighbor
if (px>1)
  u(xmin-lx:xmax-1,:) = u(xmax-lx+1:xmax,:)[px-1,pz]
! get data from bottom neighbor
if (px<npx)
  u(xmax+1:xmax+lx,:) = u(xmin:xmin+lx-1,:)[px+1,pz]
! get data from left neighbor
if (pz>1)
  u(:,zmin-lz:zmin-1) = u(:,zmax-lz+1:zmax)[px,pz-1]
! get data from rgiht neighbor
ub = ucobound(v, 3)
if (pz<ub)
  u(:,zmax+1:zmax+lz) = u(:,zmin:zmin+lz-1)[px,pz+1]

sync all
...
if (is_center_image()) &
  v(xsource,zsource) = v(xsource,zsource) + source(it+1)

call cg_fwd_2d( v, u, ...)
sync all

! get data from top neighbor
if (px>1)
  v(xmin-lx:xmax-1,:) = v(xmax-lx+1:xmax,:)[px-1,pz]
! get data from bottom neighbor
if (px<npx)
  v(xmax+1:xmax+lx,:) = v(xmin:xmin+lx-1,:)[px+1,pz]
! get data from left neighbor
if (pz>1)
  v(:,zmin-lz:zmin-1) = v(:,zmax-lz+1:zmax)[px,pz-1]
! get data from rgiht neighbor
vb = ucobound(v, 3)
if (pz<vb)
  v(:,zmax+1:zmax+lz) = v(:,zmin:zmin+lz-1)[px,pz+1]
...
end do
```

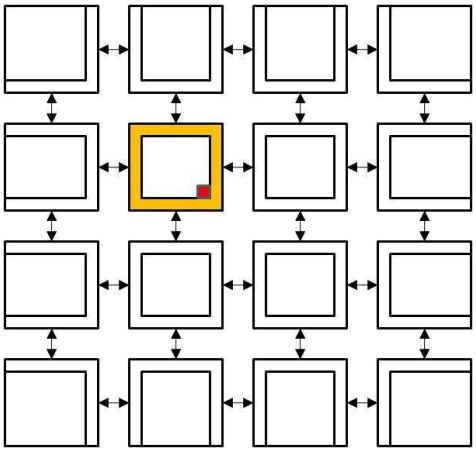
(b) CAF version

Figure 6. 2D Finite Difference Kernel Code

For our CAF parallelization (shown in Figure 6b), we decompose the arrays, *u* and *v* into equal- and fixed-sized coarrays on each image. The values of the functions, *maxval* and *minval*



(a) Update by 17-star Finite Difference Stencil in space



(b) Overall communication pattern with 16 images

Figure 7. Parallelization of 2D Finite Difference Kernel

are replaced with two "co-reduction" operations: *co_maxval* and *co_minval* (which we implement using `all-reduce` algorithm similar the one showed in Section 2). After every invocation of *cg_fwd_2d*, each image sends and receives boundary data to and from neighbouring images and updates its coarray data locally. The center image is where the source is added.We can illustrate the communication pattern that is carried out by each image when invoking *cg_fwd_2d* in Figure 7(b). The center image is where the source is added. The kernel will then propagate the values outward.

4.2 Performance Measurement

The kernel code was compiled with optimization level 3 by our CAF compiler with runtime based on ARMCI or GASNet. For MPI, the code is also re-written in Fortran 90/95 and compiled with the same optimization level by OpenUH-Fortran compiler, and linking with OpenMPI 1.3.x. Instead of measuring total time of the entire program execution, to check the efficiency of compiler/runtime for CAF- and hand-codes, we mainly observed the computation part (500 iterations) in where each image communicates with others to update the coarray data of the kernel program. We extended the original kernel code into both CAF and hand-coded versions to compare the performance of our CAF compiler/runtime on the two cluster systems.

All communication operations used in CAF and the hand-coded version are native blocking strided *put* or *get* which are supported by both ARMCI and GASNet. Due to potential deadlock during communication in the kernel logic we used non-blocking communication operations between processes in the MPI version. The

# images	xmax*zmax	OpenUH-ARMCI time	OpenUH-GASNet time	MPI time	Hand-ARMCI time	Hand-GASNet time
1	2000*1000	20.43	26.39	20.52	20.77	26.82
2	1000*1000	10.4	13.42	10.18	10.82	22.86
4	1000*500	7.21	7.72	7.14	8.06	12.18
8	500*500	3.85	3.80	3.48	4.09	14.25
16	500*250	3.27	2.32	2.11	3.7	8.39

Table 1. Execution time(sec) of 2D Finite Difference kernel on InfiniBand: 19.9 sec (baseline)

# images	xmax*zmax	OpenUH-ARMCI time	OpenUH-GASNet time	MPI time	Hand-ARMCI time	Hand-GASNet time
1	2000*1000	19.18	20.17	18.62	19.73	24.31
2	1000*1000	10.01	14.29	9.74	11.65	20.02
4	1000*500	6.14	8.96	5.82	6.24	11.80
8	500*500	3.45	7.82	3.40	3.52	9.65
16	500*250	2.00	8.85	1.92	2.05	10.38

Table 2. Execution time(sec) of 2D Finite Difference kernel on Gigabit Ethernet: 19.33 sec (baseline)

speedup in Figure 8 illustrates that the 2D Finite Difference kernel programs deliver comparable performance to the MPI version on both InfiniBand (Medusa)- and Gigabit Ethernet (Maxwell)-based cluster systems.

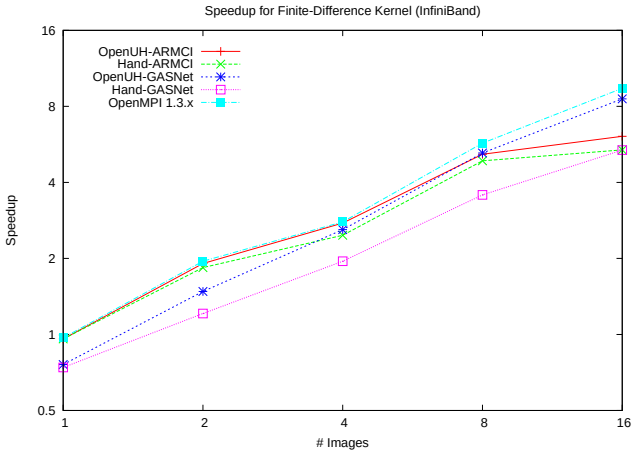
In Figure 8.a we see that on the InfiniBand cluster system, the OpenUH-ARMCI (CAF) and Hand-ARMCI version show that there is some communication overhead when 16 images exchange 2066 or 4066 bytes of data. Unlike ARMCI, the OpenUH-GASNet (CAF) and Hand-GASNet (non-CAF) kernels generally present a linear speedup with increasing number of images. GASNet-based CAF runtime runs well on InfiniBand NIC by allowing efficient packing and unpacking of data for non-contiguous data transfer. When using blocking strided *get* operation on both runtime systems with 16 images, we found that the average bandwidths for ARMCI and GASNet were 76MB/sec and 86MB/sec respectively.

On the Gigabit Ethernet-based cluster (Maxwell) 8.b, OpenUH-ARMCI and Hand-ARMCI have scalability equivalent to the MPI version. OpenUH-GASNet and Hand-GASNet shows less speedup than the other three implementations. In our experiments we observed that GASNet does not efficiently handle non-contiguous data transfers with other small-sized data transfers (which is `all-reduce`) on Gigabit Ethernet NIC. The cluster system may have some OS noises with memory latencies for communications on GASNet; but we noticed that when OpenUH-GASNet runs with two images, its execution time and speedup are less sensitive to such factors than when executing with 1 image.

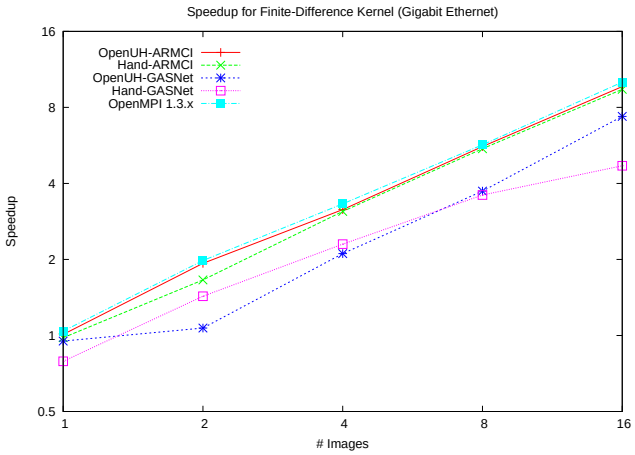
The ARMCI- and GASNet-based hand-coded kernels show lower speedup on both cluster systems. The hand-coded kernels were written in such a way as to mimic a source-to-source translation approach, though the translation strategy employed is identical to that of our implementation (that is, unoptimized). The difference in execution time can be attributed to overhead from explicitly associating dope vectors to F90 pointers in the source code itself.

5. Related Work

There have been few public Coarray Fortran implementations to date. Dotsenko et al. developed CAFC [6, 7], a source-to-source implementation based on Open64 with runtime support based on ARMCI and GASNet. They used Open64 as a front-end and implemented enhancements in the IR-to-source translater to generate Fortran source code to be compiled using GNU compilers. In contrast, we use OpenUH as an end-to-end compiler and have implemented the CAF-to-runtime translation phase in our compiler's back-end. While as of yet our compiler translation is not guided by analysis or augmented with compiler-optimizations, we plan to exploit the robust optimization framework provided by Open64 to this end. More recently, Rice University has presented a critique of proposed coarray features in Fortran 2008 [14] and have a new vision in CAF 2.0 [16]. CAF 2.0 offers a number of desirable features for a more expressive parallel programming language, including process subsets, topologies, team-based coarray allocation/deallocation, enhanced synchronizations and collectives, and asynchronous communication support. An implementation based on the ROSE source-to-source compiler infrastructure and GASNet communication library has been released.



(a) InfiniBand cluster (Medusa)



(b) Gigabit Ethernet cluster (Maxwell)

Figure 8. Speed-up for finite difference kernel on InfiniBand and Gigabit Ethernet clusters

G95 [3] provides a coarray implementation (with closed-source runtime support). G95 allows coarray programs to run on a single machine with multiple cores, or on multiple images across homogeneous networks. In this second mode, images are launched and managed via a *G95 Coarray Console*. There has been a recent effort to implement coarrays in GFortran [17], and an updated de-

sign document for this implementation is maintained online. As of this writing, the gfortran implementation does not yet support for multi-image execution, and coarray intrinsics are not supported for coarrays with bounds that are determined at runtime.

6. Conclusions and Future Work

In this paper, we have presented a basic implementation of Coarray Fortran in the OpenUH compiler. We extended our Fortran front-end, added a Coarray Lowering phase to our compiler's back-end, and implemented a communication runtime system. We evaluated this implementation using both the ARMCI and GASNet communication subsystem and observed significant performance differences. At present, our system is using blocking 1-sided communication. We plan to explore compiler techniques for overlapping communication with computation using the non-blocking interfaces provided by ARMCI and GASNet. We support global synchronizations (`sync all`) as well as more fine-grained synchronizations (`sync images`, `notify/query`). Our future plans include exploring compiler and runtime optimization strategies for these synchronizations. Another enhancement we are exploring is improving performance on an SMP node. Currently, our implementation does not distinguish between images executing on separate nodes versus the same node. But when images have access to the same physical memory, we should consider using load and store operations rather than generating communications. This will additionally help remove unnecessary buffering and copying that comes with our communication generation strategy.

Acknowledgments

This work is supported in part by Total .S.A. We would like to thank Henri Calandra and Terrence Liao for their helpful feedback on our CAF implementation based on their application needs.

References

[1] High Performance Fortran language specification. version 2.0. Technical report, Rice University, Rice University, Houston, TX, January 1997.

[2] R. Barrett. Co-Array Fortran Experiences with Finite Differencing Methods . http://users.nccs.gov/ rbarrett/PAPERS/cug06_caf.pdf, 2006.

[3] A. Beddall. The g95 project. url:http://www.g95.org/coarray.shtml.

[4] D. Buntinas. Designing a common communication subsystem. In *In Proceedings of the 12th European Parallel Virtual Machine and Message Passing Interface Conference, Euro PVM MPI*, pages 156–166. Springer Verlag, 2005.

[5] R. S. S. Charles H. Koelbel, David B. Loveman. *The High Performance Fortran Handbook*. The MIT Press, November 1993.

[6] Y. Dotsenko, C. Coarfa, and J. Mellor-Crummey. A multi-platform co-array fortran compiler. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 29–40, Washington, DC, USA, 2004. IEEE Computer Society.

[7] Y. Dotsenko, C. Coarfa, J. Mellor-crummey, and D. Chavarra-mir. Experiences with co-array fortran on hardware shared memory platforms. In *In Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, 2004.

[8] D. Eachempati, L. Huang, and B. M. Chapman. Strategies and implementation for translating OpenMP code for clusters. In R. H. Perrott, B. M. Chapman, J. Subhlok, R. F. de Mello, and L. T. Yang, editors, *HPCC*, volume 4782 of *Lecture Notes in Computer Science*, pages 420–431. Springer, 2007.

[9] T. V. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *In Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, 1992.

[10] GASNet. Global-address space networking. home page. url: http://gasnet.cs.berkeley.edu (Nov. 2009).

[11] GASNet. GASNet: GASNet Specification Version 1.8, October 11th 2008. url: http://gasnet.cs.berkeley.edu/dist/docs/gasnet.html (Oct. 2008).

[12] K. A. Huck, O. Hernandez, V. Bui, S. Chandrasekaran, B. Chapman, A. D. Malony, L. C. McInnes, and B. Norris. Capturing performance knowledge for automated analysis. Submitted to SC'08.

[13] O. H. H. Jin, , and B. Chapman. Compiler support for efficient instrumentation. In *PARCO*, 2007.

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

[15] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng. OpenUH: An optimizing, portable OpenMP compiler. In *12th Workshop on Compilers for Parallel Computers*, January 2006.

[16] J. Mellor-Crummey, L. Adhianto, G. Jin, and W. N. S. III. A New Vision for Coarray Fortran. In *The Third Conference of Partitioned Global Address Space Programming Models*, 2009.

[17] T. Moene. Towards an implementation of Coarrays in GNU Fortran. http://ols.fedoraproject.org/GCC/Reprints-2008/moene.reprint.pdf.

[18] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 533–546. Springer-Verlag, 1999.

[19] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. K. Panda. High performance remote memory access communication: The armci approach. *Int. J. High Perform. Comput. Appl.*, 20(2):233–253, 2006.

[20] R. W. Numrich and S. Graphics. Co-array fortran for parallel programming. *ACM Fortran Forum*, 17:1–31, 1998.

[21] OpenMP: Simple, Portable, Scalable SMP Programming . url: http://www.openmp.org, 2006.

[22] R. Rabenseifner. Optimization of collective reduction operations. In *Computational Science - ICCS 2004, Springer-Verlag LNCS 3036*, pages 1–9. Springer Berlin, Heidelberg, 2004.

[23] J. Reid. The new features of Fortran 2008. *SIGPLAN Fortran Forum*, 27(2):8–21, 2008.

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