# Co-Array Fortran Experiences with Finite Differencing Methods*

Richard Barrett
Future Technologies Group
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN 37931

May 26, 2006

**Abstract**

Partial differential equations are used to describe physical phenomena in many science and engineering fields. Finite differencing methods map these continuous equations into discrete space so that they may be solved on computers. Co-Array Fortran provides the means for implementing such solution methods in parallel processing computing environments. In this report we describe our experiences with several different implementations using Co-Array Fortran, supported by experiments run on the Cray X1E at ORNL.

## 1 Introduction

A broad range of physical phenomena in science and engineering can be described mathematically using partial differential equations. Determining the solution of these equations on computers is commonly accomplished by mapping the continuous equation to a discrete representation. One such solution technique is the finite differencing method, which lets us solve the equation using a difference stencil, updating the grid as a function of each point and its neighbors, presuming some discrete time step. The algorithmic structure of the finite difference method maps naturally to the parallel processing architecture and single-program multiple-data (SPMD) programming model. For example, on a regular, structured grid, $O(n^2)$ computation is performed, with nearest neighbor $O(n)$ inter-process communication requirements.

Co-Array Fortran [13] (CAF) has emerged as an important programming model for the development of scientific applications designed to execute on high performance, parallel processing platforms. This model provides the programmer with semantics and syntax that can be used to hide data movement between distributed memory processes. Specifically, it provides a remote process "load/store" capability, allowing a process to perform computations which interleave local and remote data accesses.

In this report we show that the CAF programming model provides special opportunities for achieving strong performance for finite differencing techniques. Toward that end we report on the performance of three co-array based implementations, as well as a point-to-point message passing (MPI[16]) implementation, as realized on the Cray X1E computer located at Oak Ridge National

*This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

```
REAL, DIMENSION(:,:)    ::  A ! Private array.
REAL, DIMENSION(:,:)[*] ::  B ! Co-array

ALLOCATE ( A( M, N ), STAT = IERR )      ! Allocating private array
ALLOCATE ( B( M, N )[*], STAT = IERR )   ! Allocate co-array.

A(1) = B(2)[7]   !  All images load B(2) from image 7 into their A(1).
A(5) = B(3)      !  All images load their B(3) into their A(5).
```

Figure 1: Declaring and allocating Fortran co-arrays, and the associated load models.

Laboratory (ORNL). More broadly we seek to understand how CAF may be used in a more general sense in scientific computing applications, beginning with unstructured or semi-structured mesh based applications.

This paper is organized as follows. In section 2, we describe the syntax and semantics of Co-Array Fortran. In section 3 we describe the finite difference algorithm used for our experiments, describe various implementations of the algorithm in section 4 , and report on experiment performance on the Cray X1E in section 5. In section 6 we summarize our experiences, offer our conclusions, and describe future work.

Finally, note that although the focus of this report is on the performance potential of the Co-Array Fortran programming model, throughout we offer some comments regarding the ease-of-use aspects of our experiences programming using co-arrays.

# 2 Co-Array Fortran

A proposed small syntactic extension allows the Fortran programming language to be used to create programs that execute on parallel processing computers. This proposal was first made in the form of F$^{--}$[15]. Renamed Co-Array Fortran, the extension is scheduled to be formalized into the next update to the Fortran Standard[14, 1].

This single-program multiple data (SPMD) programming model[1] requires the user to confront the parallel processing environment, but provides tools that are natural to the Fortran programmer (or procedural language programmer for that matter) which eases the burden usually required to create effective distributed memory parallel programming applications.

As with regular Fortran arrays, co-arrays are indexed from a local perspective, with access to off-process elements (or off-image, in co-array terminology) accomplished using a bracket notation. In other words, Fortran arrays are private to the image, while elements in co-arrays are shared across all images, via loads and stores, However, there is well-defined affinity between an image and its resident co-array data. Figure 1 illustrates the syntactic difference.

Regular arrays operate as usual, and thus a Fortran program operates in a parallel processing environment as a collection of asynchronous serial programs, sharing data via loads and stores among declared co-arrays, or other communication protocols, such as message passing via MPI functionality, and coordinated access to co-array data via synchronization procedures (both global and amongst subsets of images).

The independence of images means standard Fortran optimization techniques still apply. Additionally, in order to achieve strong performance, the compiler must not only support off-image loads and stores, but should attempt to schedule them so that the extra time needed for data movement be overlapped with local computations, resulting in hidden latencies. In other words, off-process

[1] Although technically SPMD, the independence of the individual images implies that MIMD programs could be configured, although startup and termination would follow the SPMD model.

```
DO I = 1, N
   A(I) = B(I)[LEFT_IMAGE] + C(I) + D(I)[RIGHT_IMAGE]
END DO
```

Figure 2: This DO-loop mixes computation with local and remote stores. We view this as the "natural" use of co-arrays, with our goal of achieving high performance.

locations should be viewed as simply another memory level in the on-process hierarchy. This leads to our notion of the "natural" use of Co-Array Fortran, based on the "load it as you need" it model, illustrated in Figure 2. We expect a quality Co-Array Fortran compiler to attempt to hide the latencies required for the off-image loads from co-arrays B and D by overlapping access with local computation, pre-fetching, and perhaps other optimization techniques.

## 2.1 The Cray X1E

The Cray X1E at Oak Ridge National Laboratory (ORNL), named Phoenix, consists of 1024 multi-streaming vector processors (MSP), each capable of performing 18 GFLOPS. For non-vectorizable computation, far less powerful scalar processors (400 MHz) are employeed. Memory bandwidth is very high, roughly half the cache bandwidth. The interconnect functions as an extension of the memory system, offering each node direct access to memory on other nodes at high bandwidth and low latency. The basic building block of the X1E is a compute module, consisting of MCM, memory, routing logic, and external connectors. Further, four MSPs behave like a traditional SMP. An MCM contains two MSPs, meaning each module contains eight MSPs. Thus two four-MSP-way SMPs comprise a module. Remote memory accesses are issued directly from the processors as load and store instructions, transparently executed over the X1E interconnect to the target processor, bypassing the local cache. This provides a favorable environment for partitioned global address space languages like Co-Array Fortran; however, it does not for shared-memory programming models, like OpenMP, outside of a given four-MSP node, due to lack of inter-node cache-coherency.

Further raising our expectations regarding the performance of co-arrays on the X1E is Cray's long commitment to this model, including native support within the Fortran compiler.

## 2.2 Beyond the X1E

Beyond Cray, many others have recognized the value of Co-Array Fortran. Of particular interest is the work of the compiler group at Rice, where an open source, portable CAF compiler[10] is under development. The result of this broad interest is the inclusion of co-arrays on the agenda for the Fortran 2008 standardization effort.

Throughout this report we will refer to CAF as a language (Co-Array Fortran). Likewise we will refer to it as a programming model, since its syntax and semantics define it as such. We will also refer to "Fortran co-arrays" , which will be the proper reference once they are formally specified in the Fortran standard.

Finally we note that herein we use the terms "image" and "processor" interchangeably, a restriction not forced upon us by Co-Array Fortran. (That is, Co-Array Fortran lets up map more than one image to a processor, but we don't use that capability in our work reported here.)

# 3 Finite difference methods

We demonstrate our finite difference method implementation in two dimensions using 5- and 9-point difference stencils. A five point difference stencil is represented notationally as

$$u_{i,j}^{t+1} = \frac{u_{i,j-1}^{t} + u_{i-1,j}^{t} + u_{i,j}^{t} + u_{i+1,j}^{t} + u_{i,j+1}^{t}}{5}, \text{for} \quad i,j = 1,\ldots n, \quad \text{for} \quad \text{timestep} \quad t.$$

A 9-point stencil includes the (up to) four points diagonally adjacent to $u_{i,j}^{t}$ : $u_{i-1,j-1}^{t}, u_{i-1,j+1}^{t}, u_{i+1,j-1}^{t}$, and $u_{i+1,j+1}^{t}$.

This notion of mapping a continuous problem to discrete space and the issues encountered by decomposing across parallel processes is illustrated in Figure 3, along with the Fortran implementation of the 5-point stencil. This problem definition presumes regular, equally spaced grid points across the global domain. This greatly simplifies the implementation of the algorithm, allowing us to focus in on the performance aspects of interest in our experiments.

# 4 Implementations

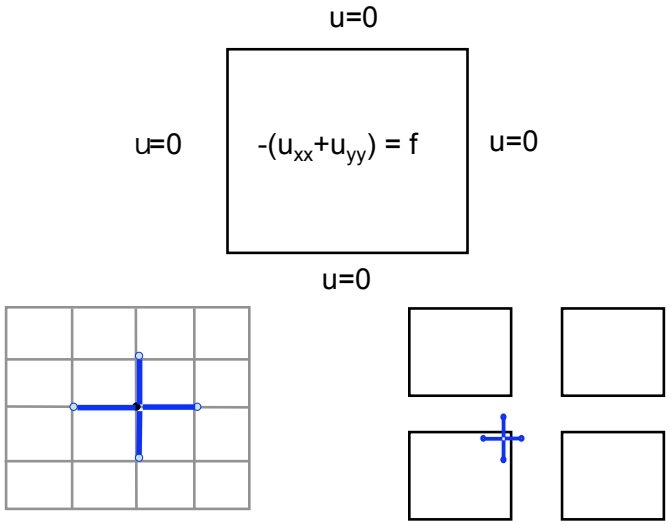
As previously shown, the finite difference computation is straightforward. Mapping the algorithm to a parallel processing environment is also straightforward. The typical decomposition strategy divides the domain into blocks. This creates artificial interior boundaries, along which each parallel process must access off-process data (called ghosts, shadows, or halos, etc) in order to compute the stencil.

The use of co-arrays requires the user to explicitly identify and manage the parallelism of the algorithm. In particular, although co-arrays allow an image to directly load or store data from any other image, the array indexing is from a local viewpoint. This requires the user to make decisions regarding the parallelism strategy. For the code segment shown in Figure 3, we've taken three different philosophical approaches for using co-arrays, described and discussed below. Note that all strategies are "load-based"; no off-process stores are required.

We begin describing the message passing model, implemented using MPI, work our way toward the natural use of co-arrays (as we defined it in section 2), then offer two modifications to that implementation that offers hints to the compiler regarding our intentions, resulting in (we hope) stronger performance

1. **MPI.** The MPI specification[16] provides several methods for moving data from one parallel process to another. The finite difference method boundary exchange requirements map well to the non-blocking point-to-point model. Our implementation is a straightforward exchange of boundary data (stored in ghost zones) followed by local computation. We note that more sophisticated implementations might enable latency hiding; however, in a complex application the coding complexity increases significantly.

   We first post the non-blocking receives (`MPI_Irecv`), followed by the non-blocking sends. This gives us a chance of executing in "expected message" mode, which may reduce intermediate buffering. Two of the boundaries are stored in memory stride 1, and the other two are stride `N`. The latter requires that we either define an MPI derived type or directly manage and load a buffer which is then passed to MPI. Previous experience has shown that although the derived type can result in "cleaner" code, the performance of the data transfer is poor; thus we directly manage the buffering requirements. Thus two boundaries must be loaded into a buffer prior to issuing the send, and analogously, two boundaries must be unloaded upon receipt of the message. We loop over `MPI_Waitany`, providing the possibility of latency hiding by processing incoming messages as they arrive.




```
REAL, DIMENSION(M+2,N+2) ::  GRID_NEW, GRID_OLD

DO J = 2,N-1

   DO I = 2,M-1

         GRID_NEW(1,J) =                                                  &
                                ( GRID_OLD(I-1,J)+                        &
              GRID_OLD(I,J-1) + GRID_OLD(I,J) + GRID_OLD(I,J+1) +         &
                                  GRID_OLD(I+1,J) )                       &
                    * FIFTH
   END DO

END DO

GRID_OLD = GRID_NEW
```

Figure 3: The top figure show a partial differential equation (the Poisson equation) described on a continuous domain, with Dirichlet boundary conditions. The discretization of this problem is shown in the figure on the middle left. The middle right figure illustrates the inter-process communication requirements when the discretized domain is decomposed across four parallel processes. The Fortran code segment implements a five point differencing scheme on an `M` $\times$ `N` grid. Note the extra (ghost) space allocated for the boundary condition.

```
CALL SYNC_TEAM ( NEIGHBORS )

!  ------------
!  Computation.
!  ------------

DO J = 2, LCOLS+1
    DO I = 2, LROWS+1


       LEFT          = GRID1(II(I,J-1),JJ(I ,J-1))[IMG_LOC(I ,J-1)]
       TOP           = GRID1(II(I-1,J),JJ(I-1,J ))[IMG_LOC(I-1,J )]
       CENTER        = GRID1(I,J)
       BOTTOM        = GRID1(II(I+1,J),JJ(I+1,J ))[IMG_LOC(I+1,J )]
       RIGHT         = GRID1(II(I,J+1),JJ(I ,J+1))[IMG_LOC(I ,J+1)]

       GRID2(I,J)    = ( LEFT + TOP + CENTER + BOTTOM + RIGHT ) * FIFTH

    END DO
END DO
```

Figure 4: This code segment shows the computational loop for the CAF version. All loads are defined within the loop, although their location (local or remote) is not known until runtime.

2. **CAF.** The computational loop interleaves local and remote loads, making no distinction between local and remote array accesses. The code segment is shown in Figure 4. This should provide the best chance for hiding remote image load latencies. However, it places all responsibility for load scheduling on the compiler, in spite of the fact that the no specific information regarding the data distribution is available until runtime. Further, some compilers will inject overhead due to the presence of the co-array bracket notation in spite of the fact that $O(N^2)$ loads will be local to the image compared with $O(N)$ off-image. The need for indirect addressing within arrays will probably also degrade performance.

   However, this model may provide the best performance, as well as a simpler coding model, for algorithms operating on semi-structured or unstructured meshes, particularly when the mesh is dynamic. In this setting all other versions will also require the indexing arrays, degrading their performance.

3. **CAF MPI-style.** We replace the MPI send/receive mechanism with loads from co-arrays into the ghost zones. The code segment is shown in Figure 5. This implementation is mostly for straightforward comparison with the MPI version. However, we don't believe it adheres to the spirit of the co-array programming model, and instead is more representative of the one-sided communication model.

4. **CAF Segmented.** Off-image data is only needed for grid points along inter-image boundaries. In an attempt to provide hints to the compiler that may be used for local and remote load scheduling, we define several separate computational loops: one operating solely on the inner grid (no remote data accesses required), and one loop along each boundary. The latter are intended to inform the compiler of the regular data access patterns. It also eliminates the need for the co-array notation for each array point, which are required for the CAF version.

   A code segment illustrating the idea is shown in Figure 6.

```
CALL SYNC_TEAM ( NEIGHBORS )

IF ( NEIGHBORS(SOUTH) /= MY_IMAGE ) & !  Get boundary data from the south.
   GRID1( LROWS+2, 2:LCOLS+1 ) = GRID1( 2, 2:LCOLS+1 )[NEIGHBORS(SOUTH)]

IF ( NEIGHBORS(NORTH) /= MY_IMAGE ) & !  Get boundary data from the north.
   GRID1( 1, 2:LCOLS+1 ) = GRID1( LROWS+1, 2:LCOLS+1 )[NEIGHBORS(NORTH)]

IF ( NEIGHBORS(WEST) /= MY_IMAGE ) & !  Get boundary data from the west.
   GRID1( 2:LROWS+1, 1 ) = GRID1( 2:LROWS+1, LCOLS+1)[NEIGHBORS(WEST)]

IF ( NEIGHBORS(EAST) /= MY_IMAGE ) & !  Get boundary data from the east.
   GRID1( 2:LROWS+1, LCOLS+2 ) = GRID1( 2:LROWS+1, 2 )[NEIGHBORS(EAST)]

CALL SYNC_TEAM ( NEIGHBORS )

!  "Serial" computation.
```

Figure 5: This code segment shows the boundary exchange using the CAF remote image (bulk) load capabilities.

```
IF ( NEIGHBORS(NORTH) /= MY_IMAGE ) THEN

   DO J = 2, LCOLS-1

      GRID2(1,J) =                                                  &
                        ( GRID1(LROWS,J)[NEIGHBORS(NORTH)] +        &
           GRID1(1,J-1) + GRID1(1,J) + GRID1(1,J+1) +               &
                          GRID1(1,J) )                              &
                * FIFTH
   END DO

ELSE

   DO J = 2, LCOLS-1

      GRID2(1,J) =                                                  &
           GRID1(1,J-1) + GRID1(1,J) + GRID1(1,J+1) +               &
                          GRID1(2,J) )                              &
                * FIFTH
   END DO

END IF
```

Figure 6: This code segment shows the computation across the north boundary of the local grid points, interleaving local and remote data accesses. The regular stride constant off-image load requirements could be recognized by the compiler for coordinated data movement.

| | |
|---|---|
| *Operating system* | `Unicos/mp 3.1.0.7` |
| *Programming environment* | `5.5.0.1` |
| *Compiler command* | `ftn -f free -O3 -Omsp -rm -Z` |
| *Runtime command* | `aprun <-p:16m> -A -n <numMSPs> -m 2000M <a.out>` |

Table 1: Each implementation described in this report was compiled and executed on the Cray X1E in the same programming environment using the same compiler options, as listed above. The runtime command enforces a virtual processor mode, whereby images are forced onto logically contiguous nodes, and each image is assigned a maximum of two gigabytes of memory.

Correctness is verified for all implementations using a simple, well-defined problem: a single "heat source" is applied to the center of an initialized grid, and the heat dissipation is tracked for $N/2$ time steps, where $N$ is the global dimension of the grid. Thus the sum of the values across the grid must be equal to the initial heat source. For the performance report here, however, such tracking is not done (it requires global communication), and the grid is filled with random numbers to ensure that a compiler doesn't optimize out work. (The latter is further ensured by passing "the answer" into a procedure.

The MPI and CAF MPI-style versions require allocation of ghost regions for storing off-image grid points; the CAF and CAF-Segmented versions can be configured to avoid the need for ghost regions. However, this increases the complexity of the code due to difference between physical boundaries and boundaries created by the domain decomposition (interior boundaries)[2]. For simplicity, we allocate ghost zones for the CAF version, yet because CAF-Segmented is inherently more complex anyway, we do not allocate ghost zones. For our regular, structured domain, the memory cost is only $4 * N$ relative to the $N^2$ space required to store the (local) grid points. However, for irregular or adaptive grids, this cost may be significantly higher.

# 5 Performance

The simplicity of our implementations lets us clearly see the work required for parallel processing. The computational workload is captured in a straightforward doubly nested loop; its easy to see that this work should map well to the X1E MSP processor; that is, it should operate entirely within vector and multi-stream mode. This assumption is borne out by the compiler generated loopmark listing files for each implementation as well as the Cray Performance Analysis Tools hardware performance counter reports. (Neither shown here in the interest of space limitations)

All experiments described in this report were executed on Phoenix, the Cray X1E located at the National Center for Computational Sciences at Oak Ridge National Laboratory (described in section 2.1). Phoenix is currently running UNICOS/mp 3.1.0.7, and we operated within the latest default programming environment, version 5.5.0.1. Our free-form Fortran implementations, using dynamic memory allocation and organized into modules though little else from the modern syntax, were compiled using the same options. Details are shown in table 1.

All experiments were run in weak scaling mode; that is, we assign a fixed number of grid points to each image, and thus the global size of the grid increases linearly with the number of images. We use a one-to-one mapping between images and (X1E MSP) processors. The maximum local grid dimension shown in these results is 8000, which means the two grid arrays consume about half of the memory available for each MSP. However, grid dimensions that consumed practically all available memory (14,000) showed the same performance characteristics.

[2]This observation is a function of the boundary conditions of the defined partial differential equation. Here we defined Dirichlet boundary conditions, our approach is easy.

## 5.1 5-point stencil

The performance of the various implementations of the 5-point difference stencil as run on the Cray X1E is shown at Figure 7. For small local grid dimensions, the co-array versions significantly outperform the MPI version. This supports the notion that the co-array protocol for loading (and presumably, storing) data from and to remote images effectively takes advantage of the X1E interprocessor communication infrastructure. However, real-world applications would be expected to assign significantly larger grid sizes to each image. In that case we see that as the local grid dimension increases, the fixed cost (latency) of the MPI version is amortized across the larger messages sizes, and therefore becomes competitive with the co-array versions.

We also see that the CAF version does not improve significantly with the increasing grid dimension. We attribute this to the complexity of the code in this implementation, and the implications on compiler optimization strategies. In particular, the indirect addressing of all loads of grid and neighbor array elements means that access patterns will only be revealed at runtime. This assumption appears to be supported by the strong performance of the CAF-Segmented and CAF-MPI versions. Apparently the compiler is able to recognize patterns that can be translated into efficient data access.

The performance of the CAF-Segmented, CAF-MPI, and MPI versions increase with grid dimension as the amount of computation grows at $O(n^2)$ while the amount of communication grows with $O(n)$. The CAF-MPI version seems the winner in this run-off, probably due to the flexibility that the implementation offers to the compiler. That is, the compiler might recognize the local-only loads within the computational loop, enabling latency hiding of the remote loads. Further, the co-array remote loads can be accomplished directly within the context of the computation, avoiding the buffering requirements of the message passing version. Again we'll note that a more sophisticated use of MPI functionality might enable similar optimizations.

This speculation will be analyzed in future work in order to better determine cause and effect.

## 5.2 9-point stencil

The 9-point stencil introduces up to four new (diagonal) partner processes, each contributing only a single grid point. This should not present any problems to the CAF versions: its simply another load. It also plays to a particular strength of the message passing model, since with a little attention to message coordination, the programmer can avoid increasing the number of messages required for the 5-point stencil, albiet at the expense of an extra inter-neighbor synchronization point: complete the boundary transfer with north and south neighbors, then exchange boundaries with east and west neighbors. The first exchange places the single point from each diagonal neighbor onto the horizontal neighbor, which is then properly shared in the east-west exchange. (Of course the order could be east-west, then north-south.) This method is also applied in the CAF-MPI implementation.

The performance of the four implementations are shown in Figure 8. For the smallest grid size, the relative performance stays about the same as with 5-point stencil, although the CAF segmented and CAF MPI-style versions are about even for the middle sizes. All implementations outperform their 5-point counterparts, due to the increased computation relative to the amount of communication.

For larger grid sizes, unlike the 5-point stencil, the MPI version becomes the clear best performer. Outperforming the CAF-Segmented version was somewhat expected since it must load the diagonal elements as four single coefficients, each from a different image; however, the CAF MPI-style version uses the same message coordination method as MPI. As with MPI, this requires two synchronization points, between pairs; we speculate that the asynchronous nature of MPI outperforms the two calls to `sync_team`, a notion we will closely examine in future work.

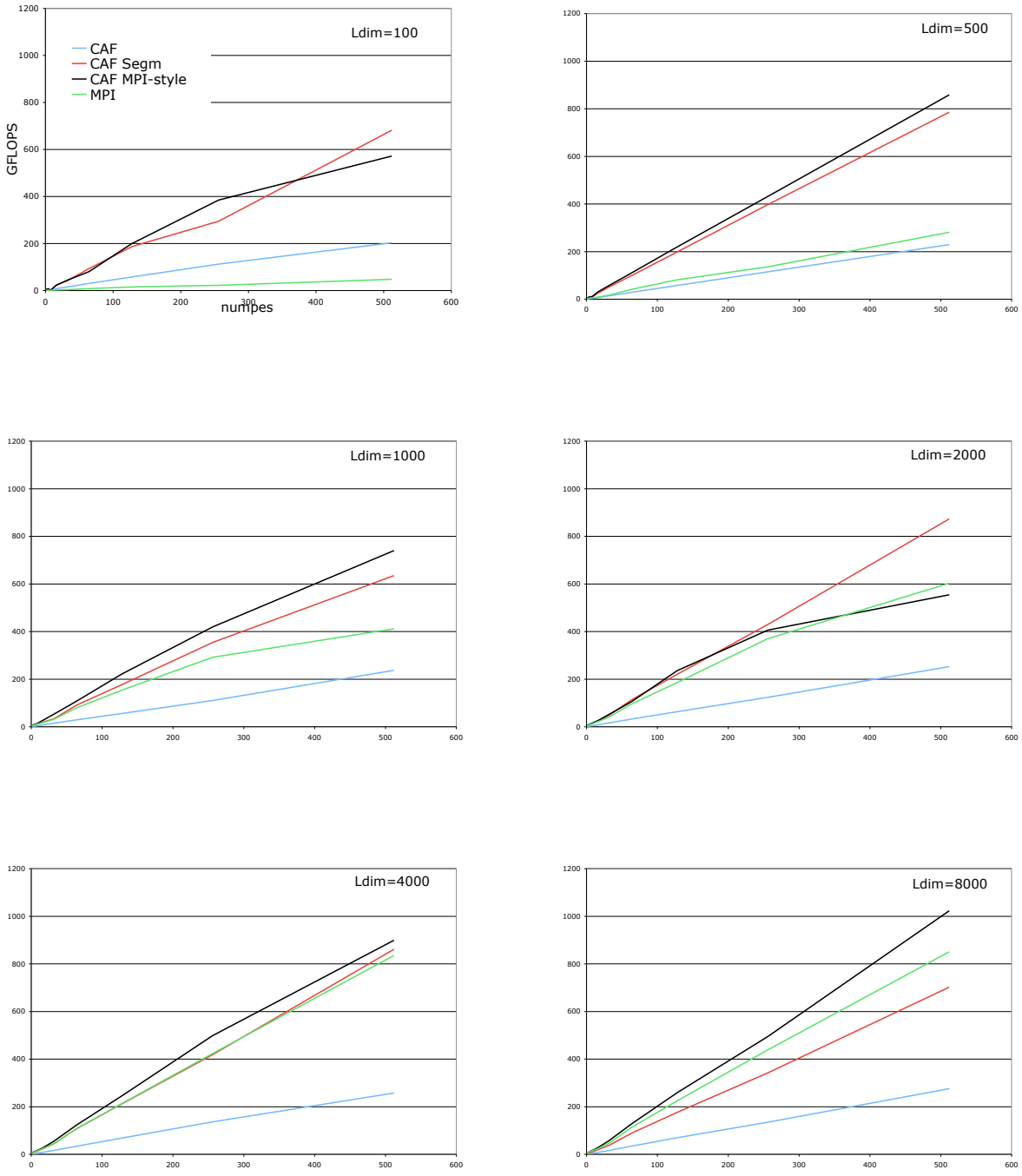



Figure 7: These graphs show the weak scaling performance of the 5-point difference stencil for the four implementations described in this report. The x-axis represents the number of MSP processors, the y-axis represents the computation rate in terms of billions of floating point operations per second (GFLOPS); the lines represent the implementation: CAF (blue), CAF segmented (red), CAF MPI-style (black), and MPI (green). The number of grid points assigned to each processor is increasing from the top left to the bottom right: per processor, the square grid dimensions (`Ldim`, show on each graph) are 100, 500, 1,000, 2,000, 4,000, and 8,000.

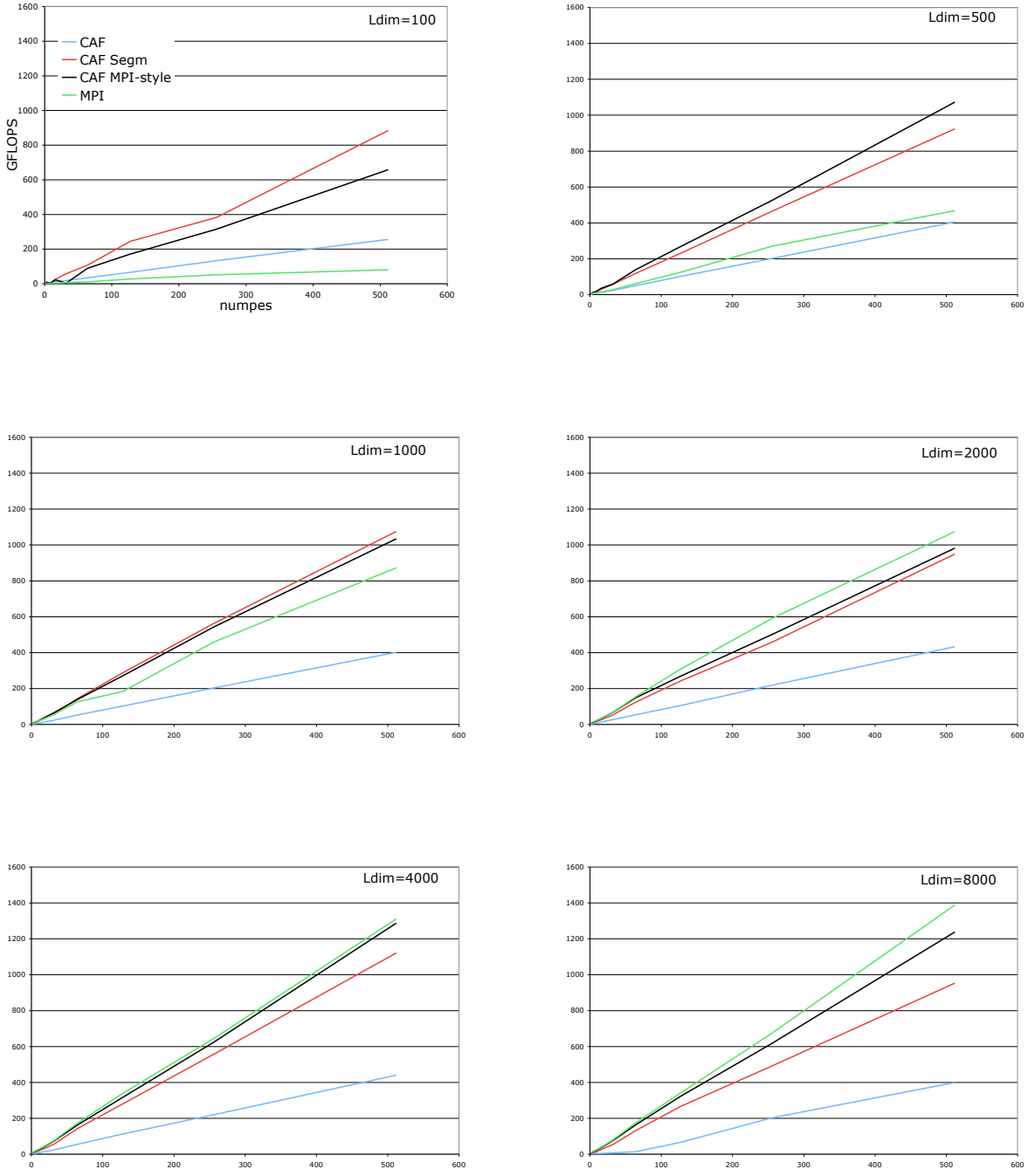



Figure 8: These graphs show the weak scaling performance of the 9-point difference stencil for the four implementations described in this report. The x-axis represents the number of MSP processors, the y-axis represents the computation rate in terms of billions of floating point operations per second (GFLOPS); the lines represent the implementation: CAF (blue), CAF segmented (red), CAF MPI-style (black), and MPI (green). The number of grid points assigned to each processor is increasing from the top left to the bottom right: per processor, the square grid dimensions (`Ldim`, show on each graph) are 100, 500, 1,000, 2,000, 4,000, and 8,000.

# 6 Conclusions

Although Co-Array Fortran adds of only a small set of extensions to the Fortran specification, that is enough to provide a wealth of tools necessary to implement our algorithm several different ways. (Those presented here do not represent the complete set. We implemented others, and others were suggested.) We've also seen that the Cray X1E provides a sufficient environment for achieving strong CAF performance. However, we caution the reader that this work is not offered as a definitive study of Co-Array Fortran, nor is it a complete comparison between the Co-Array Fortran model and the message passing (MPI) model. For one thing, many other implementations are possible using MPI functionality. Other implementations might take advantage of the particular strengths of the MPI implementation on the X1E.

As expected, all co-array based implementations showed a distinct advantage over the message passing model for small local grid sizes due to the associated small data transfers. Also as expected, this advantage shrank or was eliminated as the boundary length increased, which allowed the MPI latency to be sufficiently amortized over the bandwidth.

For the 5-point stencil, the CAF-MPI style implementation exhibited the best performance, although its advantage decreased as the boundary lengths increased. For the 9-point stencil, the MPI implementation surpassed the performance of all other implementations. Regardless, we don't believe that this offers a strong justification for the Co-array Fortran model since this use is really a one-sided communication approach, rather than a more general use of the load and store capability for which Co-Array Fortran was developed.

The CAF-Segmented version is a strong performer, but lost its advantage when the message length reached around 4000 double precision elements, at which point the MPI version has the advantage. Although it is conceivable that a compiler could recognize and take advantage of the inter-process communication pattern, this does not appear to be the case on the X1E. Regardless, the complexity of this implementation probably makes it a poor choice for most situations.

The CAF version is in our opinion the most natural use of the co-array model ("load it when you need it"), but is the poorest performer of all of the implementations presented in this report. This is due to the code complexity presented to the compiler as well as the absence of hints regarding communication requirements prior to runtime. However, preliminary work leads us to believe it has the potential to offer the best performance for more complex problems, for example those where the grid is not perfectly regular or is adaptive. It may be argued that the indexing scheme creates too much indirection, but this will be the case for any implementation when the grid is not regular. Further, the co-array brackets indicate the possibility of a remote load or store, which must be checked during runtime. However, this is exactly the sort of approach that must be strongly supported by a language designed for transparency with regard to data locality.

Perhaps Co-array Fortran could benefit from a runtime capability that could recognize repetitive communication patterns, something along the lines of the persistent communication capability found in MPI. Since the access patterns are repeated over time, the underlying communication protocol should be able to schedule data movement in coordinated ways.

Looking toward unstructured mesh-based applications, we are quite concerned about the current restriction requiring the co-array to be of equal length on each image. This will limit its utility for many applications, such as a multi-material problem, where materials enter and exit domain regions, and dynamic mesh based codes. This limitation can be overcome, but at the expense of indirection, which degrades performance.

We note that the `sync_team` functionality is an important capability. Beyond the explicit cost, again looking toward less regular applications, a global barrier would force synchronization of execution where it is otherwise not present. A user could create custom methods for controlling data flow, but the inclusion of a standard method would significantly increase programmer productivity.

Further, although deep analysis of the "ease-of-programming" issue is beyond the scope of this report, we did not find the co-array programming model offers advantages over the message passing

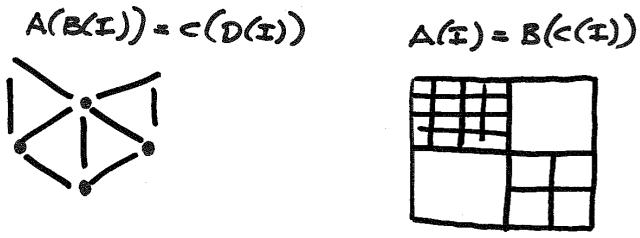



Figure 9: The sketch on the left illustrates an unstructured mesh composed of nodes, edges, and vertices, and the array indexing (double) indirection typically required to share data between parallel processes. The sketch on the right illustrates a semi-structure mesh, perhaps as used in an adaptive mesh refinement strategy, and the array indexing (single) indirection typically required to share data between parallel processes.

model in this regard. In fact co-arrays can add complexity to implementations: like message passing, CAF requires explicit knowledge of locality; unlike message passing based applications, where application-specific inter-process communication interfaces are typically defined (or definable) to which the user codes [6, 11, 2, 7, 3, 9, 5, 8, 4], the co-array model seeks to provide flexibility to compiler scheduling by co-mingling local and remote stores, limiting code re-use opportunities. Thus approaches like the CAF-Segmented would most likely not be feasible. Taking all of this into account, our experience leads us to believe that Fortran co-arrays can succeed as a new programming model in the high performance scientific computing community only if it offers significant performance improvements over the MPI model (and other message passing or one-sided communication models).

These strong statements aside, we believe our experiments show that the Co-Array Fortran model can be effectively implemented on the Cray X1E as well as current and future platforms that provide strong support for remote memory loads and stores. This is especially true for peta-scale (and beyond) architectures since communication costs must be controlled in order to achieve acceptable performance. In particular, the multi-core processor could provide special opportunities for addressing performance issues confronting the "MPI-everywhere" model. In that same vein, special opportunities may exist for extending message passing codes to incorporate co-arrays, creating a hybrid programming model.

## 6.1 Looking ahead: Irregular mesh based applications

For regular, structured grids of the dimension typically used in large scale scientific applications, our experiences with co-arrays on the Cray X1E compare favorably with the message passing (MPI) version. Yet we are even more optimistic regarding co-arrays for unstructured and semi-structured mesh-based applications. (For convenience, we will use the term "irregular" to include unstructured and semi-structured.) Our optimism is attributed to different requirements and conventions for irregular mesh implementations relative to regular meshes.

First, computations on irregular meshes in scientific applications typically access data via (at least) one level of indirection. (Figure 9 provides a visual illustration.) This means that boundary data to be exchanged is typically not organized such that it will be contiguous in memory. Thus if the inter-process data sharing mechanism transmits data as a continuous stream (e.g. message passing and shmem), the non-contiguous data must be copied into a contiguous buffer. Our CAF version places no such restriction.

Second, our CAF implementation scales well. The reason for the poor performance relative to the other versions is exactly the reason we expect better relative performance for irregular performance: it will avoid the copies needed to the data into buffers for transmission, and the other methods will also incur the cost of indirect memory addressing.

Third, irregular mesh applications are typically load imbalanced, so the `sync_team` (and perhaps `sync_memory`) functionality will be important. However, tempering our enthusiasm is that this load imbalance is a function of a differing numbers of grid points assigned to each image, yet the proposed specification for co-arrays requires that they be of equal length across all images. (This is currently a requirement on the X1E; it is not a requirement of the Rice compiler[12].) The "workaround" is to declare a Fortran derived type as the co-array, with a regular array as a member. On the one hand this can provide a coding convenience, since the type could be, say, "`GRID`", with components describing the state of the grid. However, address indirection will be required to load or store grid data on remote images, which negatively impacts performance.

This work is in progress, and will be reported in the near future.

## Acknowledgements

Mark Fahey of ORNL suggested the CAF-Segmented approach. Bill Long of Cray provided general guidance regarding the use of CAF. Nikhil Bhatia of ORNL assisted in the generation of the PAT reports. Trey White and other ORNL colleagues provided valuable suggestions and comments.

## References

[1] ISO/IEC 1539-1:2004. Information technology–programming languages–fortran–part 1: Base language.

[2] R.E. Alcouffe, R.S. Baker, and S. A. Turner. PARTISN (parallel, Time-Dependent Sn). Technical Report LA-CC 98-62, Los Alamos National Laboratory, 1998.

[3] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 - Revision 2.0.22, Argonne National Laboratory, 1998.

[4] R. Barrett. L7 communication library: User Guide, Reference Manual, and Developer Guide. Technical Report LA-UR 03-2179, Los Alamos National Laboratory, 2003.

[5] R. Barrett and M. McKay Jr. UPS User's Guide and Reference Manual. Technical Report LA-UR 99-2857, Los Alamos National Laboratory, 1999.

[6] Blackmon, M. B., B. Boville, F. Bryan, R. Dickinson, P. Gent, J. Kiehl, R. Moritz, D. Randall, J. Shukla, S. Solomon, G. Bonan, S. Doney, I. Fung, J. Hack, E. Hunke, and et al. J. Hurrell. The Community Climate System Model. *BAMS*, 82, 2001.

[7] Blackford et al. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA, 1997.

[8] R.C. Ferrell, D.B. Kothe, and J.A. Turner. PGSLib: A library for portable, parallel, unstructured mesh simulations. In *Proceedings of 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[9] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An Overview of the Trilinos Project. *ACM Transactions on Mathematical Software*.

[10] Y. Dotsenko J. Mellor-Crummey and C. Coarfa. A multi-platform co-array fortran compiler for high-performance computing. In *Los Alamos Computer Science Institute 4th Annual Symposium (LACSI 2003)*, October 2003.

[11] Darren J. Kerbyson, Henry J. Alme, Adolfy Hoisie, Fabrizio Petrini, Harvey J. Wasserman, and Mike Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the IEEE/ACM Conference on Supercomputing SC'01*, November 2001.

[12] J. Mellor-Crummey. Personal communication. Rice University, 2005.

[13] R.W. Numrich and J.K. Reid. Co-Array Fortran for parallel programming. *ACM Fortran Forum*, 17(2):1–31, 1998. `www.co-array.org`.

[14] R.W. Numrich and J.K. Reid. Co-Arrays in the next Fortran Standard. *ISO/IEC JTC1/SC22/WG5 N1542*, 1998.

[15] R.W. Numrich, J. L. Steidel, B. H. Johnson, B. Dupont de Dinechin, G. Elsesser, G. Fischer, and T. MacDonald. Definition of the F– extension to Fortran 90. In *International Workshop on Language and Compilers for Parallel Computing*, August 1997 1997.

[16] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference: Volume 1 - 2nd Edition*. The MIT Press, 1998.

## About the author

Richard Barrett is a computational scientist in the Future Technologies Group in the Computer Science and Mathematics Division (CSM) of Oak Ridge National Laboratory (ORNL), where he is also a member of the Scientific Computing Group in the National Center for Computational Science (NCCS). He can be reached at *rbarrett@ornl.gov*.