Journal of Computational Physics 297 (2015) 237–253

Contents lists available at ScienceDirect

Journal of Computational Physics

www.elsevier.com/locate/jcp

Comparing Coarray Fortran (CAF) with MPI for several structured mesh PDE applications

Sudip Garain [a,*], Dinshaw S. Balsara [a], John Reid [b]

[a] Physics Department, University of Notre Dame, USA
[b] Rutherford Appleton Laboratory, Oxfordshire, UK

ARTICLE INFO

Article history:
Received 13 November 2014
Received in revised form 6 April 2015
Accepted 14 May 2015
Available online 21 May 2015

Keywords:
PDEs
MultiGrid
CFD
FFT
Parallel computing
PetaScale

ABSTRACT

Language-based approaches to parallelism have been incorporated into the Fortran standard. These Fortran extensions go under the name of Coarray Fortran (CAF) and full-featured compilers that support CAF have become available from Cray and Intel; the GNU implementation is expected in 2015. CAF combines elegance of expression with simplicity of implementation to yield an efficient parallel programming language. Elegance of expression results in very compact parallel code. The existence of a standard helps with portability and maintainability. CAF was designed to excel at one-sided communication and similar functions that support one-sided communication are also available in the recent MPI-3 standard. One-sided communication is expected to be very valuable for structured mesh applications involving partial differential equations, amongst other possible applications. This paper focuses on a comparison of CAF and MPI for a few very useful applications areas that are routinely used for solving partial differential equations on structured meshes. The three specific areas are Fast Fourier Techniques, Computational Fluid Dynamics, and Multigrid Methods.

For each of those applications areas, we have developed optimized CAF code and optimized MPI code that is based on the one-sided messaging capabilities of MPI-3. Weak scalability studies that compare CAF and MPI-3 are presented on up to 65,536 processors. Both paradigms scale well, showing that they are well-suited for Petascale-class applications. Some of the applications shown (like Fast Fourier Techniques and Computational Fluid Dynamics) require large, coarse-grained messaging. Such applications emphasize high bandwidth. Our other application (Multigrid Methods) uses pointwise smoothers which require a large amount of fine-grained messaging. In such applications, a premium is placed on low latency. Our studies show that both CAF and MPI-3 offer the twin advantages of high bandwidth and low latency for messages of all sizes. Even for large numbers of processors, CAF either draws level with MPI-3 or shows a slight advantage over MPI-3. Both CAF and MPI-3 are shown to provide substantial advantages over MPI-2.

In addition to the weak scalability studies, we also catalogue some of the best-usage strategies that we have found for our successful implementations of one-sided messaging in CAF and MPI-3. We show that CAF code is of course much easier to write and maintain, and the simpler syntax makes the parallelism easier to understand.

* Corresponding author.
E-mail addresses: sgarain@nd.edu (S. Garain), dbalsara@nd.edu (D.S. Balsara), John.Reid@stfc.ac.uk (J. Reid).

http://dx.doi.org/10.1016/j.jcp.2015.05.020
0021-9991/

1. Introduction

A significant fraction of the resources of any parallel supercomputer are devoted to the solution of partial differential equations (PDEs). As a result, it is very interesting to study the performance of novel parallel programming paradigms and compare them to existing ones. In this paper, we compare two of the most popular recent parallel programming paradigms and they are discussed in a little detail below. We focus on the performance of several PDE solvers on structured meshes and the reason for choosing these applications is also explained. Given the availability of supercomputers with hundreds of thousands of processors, it is also interesting to catalogue the performance of our PDE solvers on many thousands of processors.

The Message Passing Interface (MPI) has long been a mainstay of parallel programming (Gropp, Lusk and Skjellum [12]). While one-sided messaging was first presented in MPI-2, it has been greatly modified in the recently-introduced MPI-3 standard with an eye to improving its usability (MPI Forum [15]). One-sided messaging has the potential to offer great benefits in the solution of PDE systems because the computation proceeds in predictable fashion, with large tracts of code devoted to CPU usage followed by a messaging step where a small or large number of messages of varying sizes may be exchanged. For structured mesh PDE applications, those messages are usually of predictable size and are intended to fill various sections of multidimensional arrays. Consequently, the messages can be properly buffered on both sides of the communication and non-blocking get or put operations can be used. As a result, it is expected that one-sided messaging will fulfill an important need for the solution of PDEs on structured meshes. (For the sake of completeness, it is worth pointing out that most vendors' implementation of MPI-3 is based on Argonne National Lab's MPICH. Consequently, most current MPI-3 implementations are based on MPI-2, which prevents them from realizing the full potential of MPI-3. The Cray implementation of MPI-3 is unique because it can be made to emulate truly one-sided messaging by drawing on Cray's unique architecture. Details for doing this are provided in Appendix A. For this reason, all our results with MPI-3 have been obtained on Cray architectures.)

Another recent line of development comes from Coarray Fortran (CAF), which has been incorporated into the current Fortran standard (known informally as Fortran 2008). At the time of writing, production grade compilers from Cray and Intel that incorporate the standard have become available and GNU will follow in 2015. A video introduction to CAF is available from http://www.nd.edu/dbalsara/Numerical_PDE_Course. CAF provides support for non-blocking one-sided get or put operations. As a result, it is expected to excel for the solution of PDEs on structured meshes. While CAF and MPI both fulfill on a common need, they are based on different philosophies which have different consequences for the end user. (Just as MPI-3 is an emergent paradigm for parallel computing, so is CAF. Its performance can also be tweaked by drawing on Cray's unique architecture. Details for doing this are provided in Appendix B. All our CAF results have also been obtained on Cray architectures.)

MPI is a library-based approach. It is, therefore, easily extensible, portable and not tied to any one compiler. However, the user then becomes bound to the static features of the library and dependent on the vendor's (or system administrator's) fine-tuning of the MPI library. CAF is a compiler-based approach. For example, it might make several alterations to remote data in cache or memory without sending the data back to its owner until the next synchronization and it might reorder statements to allow remote data required in more than one statement to be accessed together. The user necessarily becomes dependent on an available compiler for a particular architecture. This is not much of a problem in a modern setting, since most supercomputer architectures have tended to converge to the same set of chips that are connected by similar interconnect technologies. A good CAF implementation can draw on several compiler-based optimizations that are unavailable to a library-based approach. If the hardware supports specialized interconnect technology, as is the case for several offerings from Cray, then the user can get a greater benefit from those technologies too. If GPUs are also available, and if the vendor's compiler supports OpenACC, the CAF user can get the dual benefits of an optimized messaging along with optimized GPU usage. We see, therefore, that each parallel programming paradigm offers some advantages. A comparison between CAF and modern one-sided communication features of MPI-3 would, therefore, be most useful.

CAF and the newer features in MPI-3 also share a common philosophy – they recognize the value of one-sided messaging. The implementer has, therefore, to rethink the parallelization strategy from the ground-up if s/he is to benefit from these novel programming paradigms. We find that this forces the CAF and MPI-3 codes to have a very similar structure. The codes used in this comparison, therefore, have identical structure except for the messaging. This ensures that a fair comparison has been made. However, we hope that the rest of the paper also demonstrates to the reader that the CAF implementations are cleaner, very compact and very expressive.

Our first application is based on FFT techniques. FFT methods are representative of spectral and pseudo-spectral methods (Canuto [8], Canuto et al., [9]). In such methods, a spectral transform (usually an FFT) is carried out in each direction of a three dimensional problem and the PDE is solved in spectral space. Once the solution is obtained in spectral space, an inverse transform can be used to obtain the solution in real space. The FFTs (or other spectral transforms) work most efficiently if all the data for each one-dimensional FFT is available on a single processor. That is, we need to arrange the data so that it is contiguous in the direction in which the transform is being taken. The transforms have to be taken in all three dimensions, which means that the data have to be rearranged at least twice per timestep. The amount of data communicated can be quite large, and every processor gets data from every other processor. This application, therefore, favors messaging paradigms that optimize bandwidth. Such methods are very desirable for turbulence simulations where

the solution is desired on a logically rectangular grid. Modern compact schemes (Lele [13]) also have a similar structure where each direction has to be treated with an implicit solver.

Our second application involves a magnetohydrodynamics (MHD) code (Balsara [1–3]). The application is prototypical of a large class of higher order Godunov schemes that are very popular for solving computational fluid dynamics (CFD) problems. In this class of applications, the simulated data is present on logically Cartesian patches and the data for each patch are localized on a single processor. The goal of the higher-order Godunov solver is to step this data forward in time using a sequence of timesteps. The update of a zone usually requires a halo of zones around it. A second-order scheme requires a halo of one or two zones, depending on how the time-update is structured. A third-order scheme requires a halo of two or three zones, depending on how the time-update is structured. Thus, one has to make only a few small halo exchanges for a given patch of data. These halo exchanges are easy to buffer. Godunov schemes do several very detailed computations per zone and per timestep. It is not unusual to have several thousands of floating-point operations for the update of a single zone over a single timestep in an MHD calculation. (Or one might need several hundreds of floating-point operations for the update of a single zone over a single timestep in a CFD calculation.) Because of the high on-processor cost of a single timestep, the cost of messaging can be very successfully amortized in applications of this type. As a result, they have excellent scalability. We include them here because we wish to show that both paradigms for parallel programming perform admirably on this application, as expected.

Hyperbolic PDE problems, like the one in the previous paragraph, are seldom solved in isolation. Most physical applications have an elliptic or parabolic PDE solver in addition to the hyperbolic PDE solver. The quintessential elliptic problem is the Poisson problem and the fastest way to solve such a problem in a serial setting is the multigrid method (Brandt [5,6], Briggs, Henson and McCormick [7], Trottenberg, Oosterlee and Schuller [18]). As a result, the parallelization of this class of application is also interesting, with the result that multigrid methods constitute our third class of application. The methods consist of improving the solution by considering it on a sequence of meshes, each coarser than the next. These meshes are known as levels, so that the problem solution proceeds from finest level to coarsest level and then back to the finest level. On each level, the solution is improved by performing just a few relaxation steps; usually four to eight relaxation steps are used in three dimensions. These relaxation steps are very inexpensive and cost only a few floating-point operations per mesh point (Yavneh [20,21]). Between each relaxation step, a small number of halo exchanges are needed for each patch of distributed data. The cost of these halo exchanges is, however, very difficult to amortize given the small number of floating-point operations. To transfer the solution from a fine level to a coarser level requires a restriction step. Again, there are very few floating-point operations in a restriction step. Likewise, to transfer the solution from a coarse level to a finer level requires a prolongation step, which again has a very small number of floating-point operations. We see that just like the halo exchanges at a given level, the messaging in the restriction and prolongation steps across levels is very difficult to amortize. The data communicated can be quite few, but many such messages have to be exchanged by each processor. Our multigrid application is the diametric opposite of our FFT application. Our third application, therefore, favors messaging paradigms that minimize latency. The ideal messaging paradigm is one that performs admirably across these competing performance requirements. By including a spectrum of applications in the same paper, we are in a position to ascertain the strengths and weaknesses of CAF and MPI-3.

This paper has three goals. First, we wish to compare the weak scalability of CAF and MPI-3 for all of our target applications on rather large numbers of processors. Second, we wish to compare the newer MPI-3 with the older MPI-2 usage. Third, we wish to catalogue some of the best-usage strategies that we have found for CAF and MPI-3 for our target applications.

For the sake of completeness, it is worth mentioning that an early scalability study of multigrid methods operating under CAF and MPI has been presented in Numrich, Reid and Kim [16]. That study was restricted to multigrid methods on up to 64 processors using a compiler with coarray features as an extension of Fortran 95. By contrast, our study includes several applications that stress parallel programming paradigms in different ways. We also carry out our study on up to several thousands of processors using standard-conforming compilers. Most importantly, Numrich, Reid and Kim [16] only had access to an early version of MPI; whereas here we have an opportunity to compare the one-sided, non-blocking messaging features in CAF and MPI-3 with the older blocking messaging from MPI-2.

It is also very useful to mention other recent innovations in this field. Our work has used traditional multigrid methods where synchronization is done as needed. However, Bethune et al. [4] have studied the value of asynchronous Jacobi iterations where remote halo data may be obtained from any previous iteration. This reduces the need for frequent synchronization, and has potential value for Exascale applications. Messaging transactions for interprocessor communication can also be dynamically aborted, as shown by Gramoli and Harmanci [11]. This again enables another level of performance that might be needed for Exascale computation. Yang et al. [19] and Fanfarillo et al. [10] have also used MPI-3 as a communication substrate for CAF. A GASnet-based communication substrate has also been explored by the same authors. The work of Fanfarillo et al. [10] is especially useful for applications-oriented computational scientists and engineers because they build an MPI-3-based communication substrate directly into the public-domain CAF compiler from GNU. This gives the end-user the ease of expression that CAF provides in conjunction with the portability and generality of MPI-3. Compiler-based optimizations have also been discussed by Fanfarillo et al. [10] and Yang et al. [19]. Fanfarillo et al. [10] also provide an in-depth study of communication bandwidth and latency of CAF versus MPI-3 using a micro-benchmark test suite. Our present study does not use a micro-benchmark test suite; instead, we use complete scientific applications. While micro-benchmark test

suites give greater insight to computer scientists, comprehensive studies like this one are, in our humble opinion, more useful and reassuring to computational scientists and engineers.

The plan of the paper is as follows. Section 2 provides a brief overview of CAF messaging. (We do not provide an analogous section for MPI-3 because the MPI Forum [15] has provided a detailed introduction to MPI-3.) Section 3 considers the fundamental communication problems when solving PDEs. Section 4 provides weak scalability studies using CAF and MPI for FFT, MHD and multigrid applications. Section 5 presents discussions and conclusions.

2. A very brief introduction to coarrays

As with MPI, a coarray Fortran program executes as if it were replicated a fixed number of times. The programmer may retrieve this number at run time through the intrinsic procedure *num_images()*. Each copy is called an image and executes asynchronously usually, but not necessarily, on a separate physical processor. Each image has its own index, which the programmer may retrieve at run time through the intrinsic procedure *this_image()*. Consequently, an image can find its image number and store it in an integer variable "*me*" with the statement "*me = this_image()*". Each image has its own set of data objects all of which may be accessed in the normal Fortran way. Some objects are declared with additional dimensions, called codimensions, in square brackets. Codimensions have a different name because they are different from additional normal dimensions. The syntax for codimensions, however, is designed such that a programmer uses it very much like normal Fortran syntax to access data across memory images. For more details, see Chapter 19 of Metcalf, Reid, and Cohen [14].

We will use the term "image" throughout the rest of this paper. It corresponds to MPI "rank" except that rank runs from 0 and image indices run from 1.

3. The fundamental communication problems when solving PDEs

3.1. Halo exchange

Many methods for solving partial differential equations divide the computational domain into patches and hold the representation of each patch on a single processor. This is reasonable for applications that involve halo operations, and our MHD (or any flow solver) is like that. The patches have boundaries with neighboring patches. Active zones that abut these boundaries form a small halo of zones at the boundaries of each patch. Some of the values in this halo of zones have to be obtained from neighboring patches. As much as possible, computation is performed independently on the patches, but regular exchange of halo data between them is essential for solving the whole problem. It is usually convenient to expand the local data structure to include a copy of the halo data.

For simplicity, we begin by considering the case where a regular 2-dimensional grid is in use, because this shows the essentials of the problem on complicated meshes in 2 and 3 dimensions. On a regular 2-dimensional grid, we might make the array declaration

$$real :: d\,(1 - ihalo{:}mx + ihalo,\ 1 - ihalo{:}my + ihalo)$$

within which the array section *d(1:mx,1:my)* holds the patch's data and the rest is halo data copied from other patches. The halo width is given by *ihalo* and depends on the type of finite difference or finite volume scheme being used. Declaring the above array ensures that one copy of this array is available on each image and we call for sufficient images to cover the whole computational domain. An internal patch has eight neighboring patches: four across edges with halo data in the array sections

$$d\,(1{:}mx,\ 1 - ihalo{:}0),\ d\,(1{:}mx,\ my + 1{:}my + ihalo),$$
$$d\,(1 - ihalo{:}0,\ 1{:}my),\ d\,(mx + 1{:}mx + ihalo,\ 1{:}my)$$

and four across corners with halo data in the array sections

$$d\,(1 - ihalo{:}0,\ 1 - ihalo{:}0),\ d\,(1 - ihalo{:}0,\ my + 1{:}my + ihalo),$$
$$d\,(mx + 1{:}mx + ihalo,\ 1 - ihalo{:}0),\ d\,(mx + 1{:}mx + ihalo,\ my + 1{:}my + ihalo)$$

Unless the halo width is one, none of these array sections is contiguous in memory. Our experience is that CAF is slow when handling array sections that are not contiguous and MPI does not accept them at all. We can avoid the problem by using buffers on both sides. The buffer on the sending side must be declared as a coarray so that its data can be accessed by other images. We focus on the exchange of data from the bottom of one patch to the top of another. The buffer variables are declared as follows:

$$real :: d\_buff\_bottom\,(1 - ihalo{:}0,\ 1{:}my)[*]$$
$$real :: local\_d\_buff\_top\,(1 - ihalo{:}0,\ 1{:}my)$$

Now each of the images can access any data that is stored in *d_buff_bottom* by using a coarray index. The exchange of data from the bottom of one patch to the top of another takes place as follows:

```
! The following code is done on all images. It loads data into
! the coarray.
    d_buff_bottom (1 − ihalo:0, 1:my) = d (1 − ihalo:0, 1:my)

! Invoke a barrier
    Sync All

! The following code is done on the all images. It is the
! messaging step. Please observe how the coarray index is used.
    neighbor_image = ...
    local_d_buff_top (:,:) = d_buff_bottom (:,:) [neighbor_image]
    d (mx + 1:mx + ihalo, 1:my) = local_d_buff_top (1 − ihalo:0, 1:my)
```

Here, it is assumed that each patch of a regular mesh knows how to compute which neighboring patch is to be indexed by the cosubscript *neighbor_image*. Please notice how simple the messaging step is in CAF. That contributes to code maintainability.

For a regular 3-dimensional grid, the problem is more severe. For a full stencil, there are 26 neighbors. (A skinnier stencil can bring this number down to 6 neighbors.) There are 6 neighbors across faces, 12 neighbors across edges, and 8 neighbors across corners. For an efficient implementation in CAF or MPI, buffering is needed for all these inter-neighbor data transfers.

Even on complicated 3-dimensional meshes with different data collocations, the idea is essentially the same. Unless the data on the remote image is contiguous in memory, it should be buffered there for access from the local image in a single message. And unless the data is contiguous in memory on the local image, it should be copied first into a buffer. This buffering is essential when MPI is in use and desirable for efficiency when CAF is in use.

While CAF has syntax to perform the required operation directly, see the example above, MPI-3 requires the preliminary step of the establishment of a window for each array to be accessed and that window needs to be locked before each access and unlocked afterwards (i.e. we use passive target communication in MPI-3). We show an example of one-sided MPI-3 usage in Appendix A.

3.2. Data rearrangement

If a spectral transform (usually an FFT) is to be carried out in each direction of a 3-dimensional problem, it is very desirable for the data for each individual FFT to be on a single image. We have chosen to use two distributions, which we call "X" and "YZ". In the X distribution, all the data for any particular X value is held on a single image so that FFTs in the Y and Z directions can be performed without communication. In the YZ distribution, all the data for any particular YZ pair of values is held on a single image so that FFTs in the X direction can be performed without communication. We use two separate coarrays for the two distributions so that data can be copied from one to the other asynchronously without fear of overlap. We assume that the problem is of size $nx \times ny \times nz$. Please see Fig. 1.

In the X distribution, each local array holds data for a range of x-values, which we call a "layer". All the y- and z-values, which are in the range $[1, ny] \times [1, nz]$, are accessible in that layer. For simplicity, we assume that the total number of x-indices is an exact multiple of the number of images so the layers can be all of the same size and fit exactly. It is natural to use three subscripts and one cosubscript. The subscripts are for the index within the layer, for the y-index, and for the z-index; the cosubscript is for the layer. We suppose that there are ni layers each of size $leni$. Note that ni is the number of images and $ni \times leni$ is the total number of x-values, so that $ni \times leni$ is equal to nx. The array might be declared thus

```
complex :: X_dist (leni, ny, nz)
```

In the YZ distribution, each local array holds data for a rectangular block of y- and z-values, and for all x-values. For simplicity, we assume that the overall number of y- and z-indices is such that the blocks can be all of the same size and fit exactly. It is again natural to use three subscripts, now for the x-index, the y-index within the block, and the z-index within the block. We suppose that there are $nj \times nk$ blocks each of size $lenj \times lenk$, so that $nj \times lenj$ is the total number of y-values. Consequently, $nj \times lenj$ is equal to ny. Similarly, $nk \times lenk$ is the total number of z-values so that $nk \times lenk$ is equal to nz. The cosubscript now labels the block, counting pagewise, that is, block j, k with $1 \le j \le nj$ and $1 \le k \le nk$, has cosubscript $j + (k-1) \times nj$. The array might be declared thus

```
complex :: YZ_dist (nx, lenj, lenk)
```

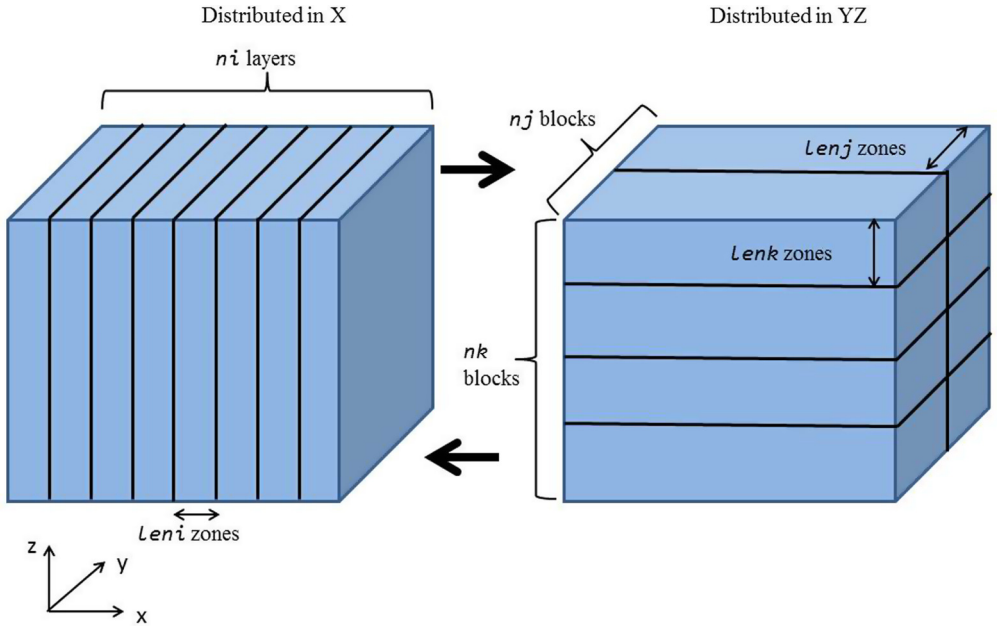

Fig. 1. The left panel shows the data layout across eight images for taking Fourier transforms in the y- and z-directions. The right panel shows the data layout for taking Fourier transforms in the x-direction. The direction(s) in which the FFTs are taken have to be serialized. All images communicate with one another to change the data distribution.

The two distributions are illustrated in Fig. 1, with the X distribution on the left and the YZ distribution on the right. Here, we have 8 images ($ni = 8$). Let us give a concrete example for a mesh with global dimensions $64 \times 64 \times 64$. Each layer has size 8 ($leni = 8$) and each block has size 32 by 16 ($lenj = 32$ and $lenk = 16$).

For the transformation to the X distribution from the YZ distribution, image i needs the array section

```
YZ_dist (i * leni - i + 1:i * leni, :, :)
```

from every image (including itself). This picks out the layers that it needs. This array section is not contiguous in memory, so copying from it would be slow with CAF and impossible with MPI. Therefore, we make every image copy its data into a coarray buffer thus:

```
complex :: buff (leni, lenj, lenk, ni)[*]
do i = 1, ni
    buff (:, :, :, i) = YZ_dist (i * leni - i + 1:i * leni,:, :)
end do
```

After synchronization, all images copy the data they need. Thus we have

```
complex :: temp (leni, lenj, lenk)
me = this_image()
you = me
do iy = 1, ni
   you = you + 1; if (you > num_image()) you = 1
   k = 1 + (you - 1)/nk
   j = you - (k - 1) * nk
   temp (:, :, :) = buff(:, :, :, me) [you]
   X_dist (:, j * lenj - j + 1:j * lenj, k * lenk - k + 1:k * lenk) = temp (:, :, :)
end do
```

Notice that this loop makes the image access its neighbor first, then the image after that, etc. The simple do loop running from 1 onwards would be slow because all the images would first access image 1, then they would all access image 2, etc. Such an access pattern would increase the traffic of messages on one processor at a time, which is not good. The scheme shown above evenly distributes the messages across processors. Notice, too, that a buffer array is needed here because the

section into which the data is placed is not contiguous in memory. All the blocks have the same size and all the images access one block from every other image.

Similar considerations apply to the transformation to the YZ distribution from the X distribution. We make every image copy its data into a buffer. Thus we have

```
complex :: buff (leni, lenj, lenk, ni)[*]
do i = 1, ni
    k = 1 + (i − 1)/nk
    j = i − (k − 1) ∗ nk
    buff (:, :, :, i) = X_dist (:, j ∗ lenj − j + 1:j ∗ lenj, k ∗ lenk − k + 1:k ∗ lenk)
end do
```

After synchronization, all images copy the data they need. Thus we have

```
complex :: temp (leni, lenj, lenk)
me = this_image()
you = me
do ii = 1, ni
    you = you + 1; if (you > ni) you = 1
    i = you
    temp (:, :, :) = buff (:, :, :, me) [you]
    YZ_dist (i ∗ leni − i + 1:i ∗ leni, :, :) = temp (:, :, :)
end do
```

Notice how well balanced these computations are. All the blocks have the same size and all the images access one block from every other image. The amount of data moved is large – almost all the data is moved from one image to another when going from either distribution to the other.

For the actual FFTs, we use a standard one-dimensional FFT routine. These routines are usually interchangeable with one another, enhancing portability and on-core optimization. We also assume that FFTs are required for several sets of data (for example, the three velocity fields in incompressible fluid dynamics). Our coarrays therefore have an extra cosubscript for the real and imaginary parts and for the multiple sets of data. The properties are not materially affected however.

3.3. Multigrid processing

Multigrid methods excel at solving elliptic and parabolic problems. They are, therefore, used as part of an overall strategy in a flow solver that has to model non-ideal terms. Our presentation of multigrid solvers is, therefore, specialized so that the multigrid solver can be used as part of an adaptive mesh flow solver.

For our multigrid work, we use a code that has been developed for solving PDEs on adaptive grids. It is based on the use of smallish grid patches of a fixed number of zones, usually $16 \times 16 \times 16$ or $32 \times 32 \times 32$ zones. We start with a small number of patches that form a regular rectangular grid, which we label “level 1”. There may be only one patch in level 1. Each level is formed from the previous one by replacing all of the patches by eight patches with halved zone size (but same number of zones) in all three dimensions. The level number is incremented as the mesh is refined. For a fixed mesh multigrid calculation, which we report on here, we seek to solve our PDE on the finest level, using the other levels to help.

Multigrid and adaptive mesh codes, therefore, have a similar data structure in this programming paradigm. For an adaptive mesh calculation, which we will report on in a subsequent paper, only a few of the patches on one level may be subdivided into finer patches in order to obtain the next refined level; and the process can be repeated. The relationships between the patches at different levels may be represented as a tree. When a level is fully refined, each patch spawns eight child patches. I.e. the eight patches that replace a single patch at the previous level are children of the single parent patch spawned them. In fact, when a patch is subdivided, it knows who its children are and the children in turn know their parent so that each patch can keep a small local list of its parent and children as these are built. In such an approach, each patch can naturally keep track of its parent, its immediate neighbors in the same level as itself, and its children. The number of such neighbors and children stays bounded, resulting in a very precise load balancing strategy. It is known as an “oct-tree” because a node with children always has eight children on uniform meshes. Because each patch has the same number of zones, the load balancing becomes easy and this method yields highly scalable adaptive mesh applications. Please note that a patch at a given level finds its neighbors from amongst the other patches that reside on the same level.

We have already shown how the parent and child lists can be maintained on each patch by keeping track of this process at the time of building a finer level. We also point out that each patch can find its neighbors very easily once a level is built. The neighbors of a patch at any level are always found by first visiting its parent and asking for the parent’s neighbors. The patch then finds its neighbors by building a list of its parent’s children or the children of its parent’s neighbors. This list will not be large. The patch then inspects this list of potential neighbors. If a halo overlap is found, the patch in the list is

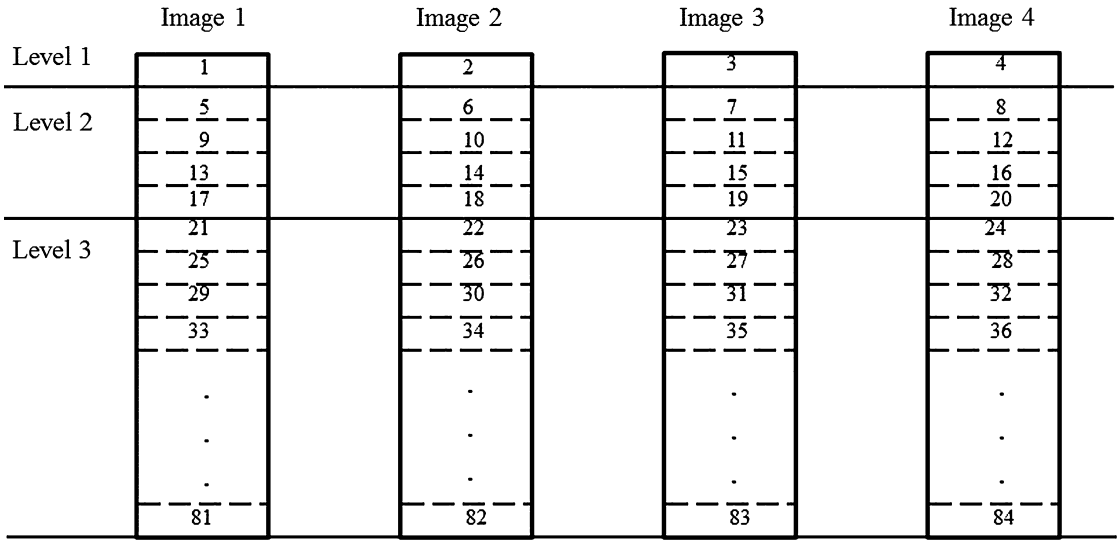

Fig. 2. Schematically represents the arrangement of 84 patches in a three level multigrid hierarchy that is distributed across four images. Each of the small rectangles represents a patch with the same number of zones in two dimensions. Because we show a two-dimensional problem, the number of patches quadruples at each level. The dots indicate patches that are not illustrated.

now taken to be a true neighbor of the original patch. On the coarsest level we always have an easy strategy for identifying neighbors. Thereafter, this process will work recursively. The tree structure of this hierarchy is now evident.

We store values for the meshes at all levels, distributed over all the images. For load balancing, it is clearly desirable to distribute the patches as evenly as possible. We therefore begin by putting the level 1 patches on images 1, 2, . . . , wrapping at *num_images()* if necessary. Each level starts from where the previous level finished, always wrapping at *num_images()*. For multigrid iteration, we need access from each level to data at adjacent levels, which makes it desirable to store parent and child patches on the same image. We therefore order the patches added at each level to achieve this as much as possible. Fig. 2, which will be discussed in detail later, shows such an example.

Once a multilevel mesh structure is established, one has to perform certain operations on the meshes. Multigrid processing is based on the realization that simple iterative schemes (like Jacobi or Red–Black relaxation) can make the solution to an elliptic problem converge quite rapidly on a coarse mesh. However, the convergence on a fine mesh, used by itself, can be rather slow. By using the help of all the levels in a multigrid mesh hierarchy, this convergence can be sped up on all levels of the hierarchy, including the finest level where the solution is desired. For the calculation to be efficient, the solution on any given level has to be improved using a few relaxation steps. Then the solution is either taken to a coarser mesh or it is interpolated to a finer mesh. The technical name for taking the solution to a coarser mesh is “restriction”. The process of interpolating the solution to a finer mesh is called “prolongation”. The relaxation step is intended to be very, very light weight; each relaxation step usually takes less than a few tens of floating-point operations per zone. After each relaxation step, halo exchanges are needed, which involves messaging. The restriction and prolongation steps also involve messaging, followed by only a few floating-point operations.

For multigrid prolongation, we mostly use data from the parent. The problems are essentially those of halo updating that we discussed in Section 3.1. It is very desirable to group all the data needed from one image by another. For multigrid restriction, we mostly use data from the children. Note, too, that some of the children may be on different images. It is again very desirable to group all the data needed from one image by another.

For the work we report in Section 4.3, all grids are regular. For a fixed mesh multigrid calculation, all patches on one level are subdivided to yield the finer level. Every patch, except at the finest level, has exactly 8 children. Furthermore, at one of the levels there are exactly the same number of patches as processors. This means that the children of any node at this or a finer level will be on the same image as their parent. This choice gives CAF and MPI-3 the best odds for doing an optimized calculation.

It is interesting to discuss the data structures needed in CAF (and MPI-3) for supporting this one-sided messaging. While the applications we present in the next section are indeed three dimensional, we present the two-dimensional case here for the sake of simplicity. Please focus on Fig. 2 which corresponds to a case where four images are used to process a multigrid hierarchy with three levels. It shows a situation where there are four patches covering the physical domain on level 1, which is the coarsest level. Those patches have been numbered 1 to 4. We also have 16 patches, each with the same number of zones (and half the zone size), on level 2. Those patches are numbered 5 to 20. To have the best on-processor restriction and prolongation, we ensure that patches 5, 9, 13 and 17 are the immediate children of patch number 1, and so on. This also helps with halo exchanges between neighbors. Because all patches at any level have the same number of zones, there are 64 level 3 patches and they are numbered 21 to 84. As with level 2, we can ensure that patches 21, 25, 29 and 33 are

the children of patch 5, and so on. We see that on each image we will need to hold data for 21 patches. Consequently, the four images shown in Fig. 2 will hold all of the 84 patches needed in the multigrid hierarchy. Each image keeps a list of its own patches so that local patches can be processed rapidly just by going down the list. This is important, for example, when doing a relaxation step at a given level.

We can now extend the data declaration in Section 3.1 to declare

```
Integer :: n_teams = 21
real :: d (1 − ihalo:mx + ihalo, 1 − ihalo:my + ihalo, n_teams)
```

This ensures that data for 21 two-dimensional patches can be stored on each image. Each patch has $mx \times my$ active zones along with a halo width of *ihalo* zones. As in Section 3.1, each patch will store neighboring patch data in the eight array sections that constitute its halo of ghost zones. As before, we do buffered messaging and illustrate it for the exchange of data from the bottom of one patch to the top of another. The case illustrated here is for halo exchange from neighbors which are all at the same level, but the same concept is easy to extend to restriction and prolongation. The declaration of buffer variables is now extended as follows:

```
real:: d_buff_bottom (1 − ihalo:0, 1:my, n_teams)[∗]
real:: local_d_buff_top (1 − ihalo:0, 1:my)
```

Now each of the images can access any data that is stored in *d_buff_bottom* by using a coarray index *and also the third index of the array itself!* The exchange of data from the bottom of one patch to the top of another takes place as follows:

```
! The following code is done on all images for a computation at level
! “ilevel”. It loads data at that level into the coarray.
    DO j = level_start (ilevel), level_end (ilevel)
        d_buff_bottom (1 − ihalo:0, 1:my, j) = d (1 − ihalo:0, 1:my, j)
    END DO

! Invoke a barrier.
    Sync All

! The following code is done on the all images at level
! “ilevel”. It is the messaging step at the current level.
! Please observe how the coarray index is used.

    DO j = level_start (ilevel), level_end (ilevel)
        neighbor_location = ...
        my_neighbor = ...
        local_d_buff_top (:, :) = &
        d_buff_bottom (:, :, neighbor_location) [neighbor_image]
    d (mx + 1:mx + ihalo, 1:my, j) = local_d_buff_top (1 − ihalo:0, 1:my)

END DO
```

Here *level_start* is an integer array that holds the starting index of the set of patches on the image at any given level and *level_end* is an integer array that holds the ending index of the same set. For example, in Fig. 2, and for level 3 we have *level_start(3)* = 6 and *level_end(3)* = 21. Each patch is also made to store its neighbor's image index so that it always knows that the neighbor is located on the image with index *neighbor_image*. For Fig. 2, *neighbor_image* lies between 1 and 4 because we have 4 images. The desired neighbor can be one of several patches stored on *neighbor_image*. Which of those patches has to be picked out is stored in *neighbor_location*.

Notice that all the data structures in this sub-section can be put inside a single module. The restriction, prolongation, relaxation and halo exchanges can also be packaged as subroutines that reside in this module. Since recursive processing is permitted in Fortran, the entire multigrid hierarchy can be recursively traversed. V- and W-cycles, which are some of the most popular cycling strategies in multigrid processing can then be implemented naturally in this formulation.

Table 1a
Showing the times in seconds for three-dimensional complex-to-complex Fourier transform of three variables followed by a three-dimensional complex-to-complex inverse Fourier transform of the same three variables. The results from averaging ten iterations on Darter are shown.

Number of cores	Problem size	CAF (s) total time; time spent in messaging	MPI-3 (s) total time; time spent in messaging
16	1024 × 256 × 256	13.28; 1.59	14.20; 2.47
32	2048 × 256 × 256	13.61; 2.26	14.70; 3.23
64	2048 × 512 × 256	14.80; 2.97	16.11; 4.26
128	2048 × 512 × 512	13.69; 4.09	14.52; 4.93
256	4096 × 512 × 512	14.77; 4.62	15.64; 5.43
512	4096 × 1024 × 512	15.53; 5.02	16.90; 6.39
1024	4096 × 1024 × 1024	15.98; 6.60	16.55; 7.19
2048	8192 × 1024 × 1024	17.34; 7.67	17.81; 8.10
4096	8192 × 2048 × 1024	20.59; 11.07	19.70; 10.42

Table 1b
Showing the times in seconds for three-dimensional complex-to-complex Fourier transform of three variables followed by a three-dimensional complex-to-complex inverse Fourier transform of the same three variables. The results from averaging ten iterations on Blue Waters are shown.

Number of cores	Problem size	CAF (s) total time; time spent in messaging	MPI-3 (s) total time; time spent in messaging
16	1024 × 256 × 256	30.41; 2.37	30.20; 3.57
32	2048 × 256 × 256	31.50; 3.21	31.39; 4.52
64	2048 × 512 × 256	32.82; 4.14	32.09; 4.92
128	2048 × 512 × 512	30.35; 6.07	31.39; 7.27
256	4096 × 512 × 512	33.44; 8.54	32.45; 7.50
512	4096 × 1024 × 512	36.24; 10.83	34.24; 9.08
1024	4096 × 1024 × 1024	35.06; 16.78	37.50; 19.79
2048	8192 × 1024 × 1024	37.91; 18.84	39.96; 21.54
4096	8192 × 2048 × 1024	50.69; 32.11	47.81; 29.01
8192	8192 × 2048 × 2048	63.86; 45.44	62.30; 44.64

4. Weak scalability comparisons between CAF and MPI-3

In this section we present weak scalability studies for CAF and MPI-3. The codes were nearly identical with the only difference that one-sided messaging was used for CAF (see Section 3.1) whereas one-sided MPI_GET routines were used for MPI-3 (see Appendix A for an example). Scalability studies are shown for FFT techniques, Multigrid methods and for an MHD code.

CAF is now available on all Cray machines. We had access to two different types of Cray machines that span two generations of architecture from that vendor. We had access to a Cray XC30, called Darter, at NICS/XSEDE. Darter has 11,968 compute cores based on the Intel *XEON Sandy Bridge* processor and an interconnect based on the Cray *Aries* router. We also had access to a Cray XK, called Blue Waters, at NCSA/NSF. Blue Waters has almost 300,000 compute cores of older *Opteron* vintage and a Cray *Gemini* interconnect.

4.1. Fast Fourier techniques

We ran ten iterations of an FFT-based spectral code. The code carried out three complex-to-complex FFT transforms from real to Fourier space. This was followed by a small amount of computation in Fourier space and three complex-to-complex inverse FFT transforms from Fourier space to real space. The use of three FFTs is very representative of a spectral scheme-based incompressible fluid flow code. The basal one-dimensional FFT was from Temperton [17]. The results from averaging ten iterations on Darter are shown in Table 1a. We see that CAF is usually (but not always) slightly faster than MPI-3 on Darter for all processor sizes. Table 1b is similar to Table 1a but shows a slightly larger weak scalability study on Blue Waters. We see larger fluctuations on Blue Waters because the older Gemini network in that machine is more prone to saturation. The Gemini network is based on a three-dimensional torus topology. The Aries network in Darter is based on a butterfly topology and is less prone to saturation. It is, however, satisfying that CAF and MPI perform comparably. This shows that there CAF is entirely competitive with the best that MPI-3 has to offer for even the largest numbers of processors that we tried here. Table 1c shows the results of using MPI-2 on Blue Waters. Comparing Table 1c to Table 1b we see that both CAF and MPI-3 show a considerable improvement over MPI-2 at large numbers of processors.

A one-dimensional FFT on "N" data points is an "$N \log N$" process. In the table, each doubling of one of the dimensions of the physical problem has been matched by a doubling of the number of processors. The number of floating point operations involved in an $n_x \times n_y \times n_z$ zone FFT transform is proportional to $n_x n_y n_z(\ln_2(n_x) + \ln_2(n_y) + \ln_2(n_z))$. A doubling of one dimension, say the x-dimension, from "n_x" to "$2n_x$" results in a fractional increase in the floating point operations given by $2(m+1)/m$, where $m \equiv \ln_2(n_x) + \ln_2(n_y) + \ln_2(n_z)$. The factor of "2" in the previous formula can be offset by a doubling

Table 1c
Showing times in seconds for the same runs as in Table 1b. The only difference is that MPI-2 was used on Blue Waters.

Number of cores	Problem size	MPI-3 (s) total time; time spent in messaging
16	1024 × 256 × 256	26.27; 6.71
32	2048 × 256 × 256	27.59; 7.83
64	2048 × 512 × 256	29.09; 7.88
128	2048 × 512 × 512	44.06; 16.77
256	4096 × 512 × 512	39.54; 10.82
512	4096 × 1024 × 512	39.0; 10.28
1024	4096 × 1024 × 1024	38.44; 13.92
2048	8192 × 1024 × 1024	48.55; 22.58
4096	8192 × 2048 × 1024	67.72; 41.97
8192	8192 × 2048 × 2048	102.17; 78.59

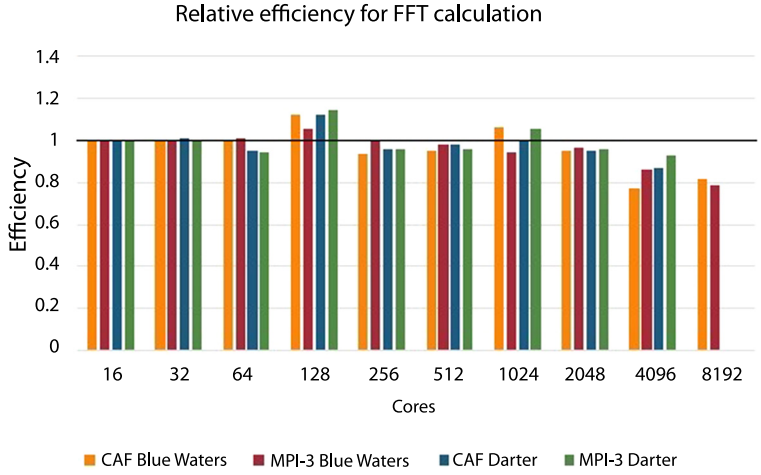

Fig. 3. Shows the parallel efficiency relative to every doubling of processors for CAF and one-sided MPI-3. The results are based on FFT simulations from Table 1.

Table 2a
Shows CAF v/s MPI-3 timings (in seconds) for a single multigrid W-cycle on Darter. These are for variable number of levels and the synchronization of a patch uses the full 27 point stencil.

Number of cores	Zones on finest mesh	Maximum # of levels	CAF total time; interlevel time; halo at level time	MPI-3 total time; interlevel time; halo at level time
8	128 × 128 × 128	3	0.79; 0.07; 0.18	0.85; 0.08; 0.27
16	256 × 128 × 128	3	0.79; 0.07; 0.18	0.87; 0.08; 0.28
32	256 × 256 × 128	3	0.84; 0.08; 0.20	0.98; 0.09; 0.38
64	256 × 256 × 256	4	1.27; 0.13; 0.32	1.43; 0.15; 0.49
128	512 × 256 × 256	4	1.34; 0.18; 0.31	1.56; 0.23; 0.51
256	512 × 512 × 256	4	1.40; 0.22; 0.33	1.80; 0.27; 0.68
512	512 × 512 × 512	5	2.31; 0.34; 0.59	2.72; 0.40; 0.94
1024	1024 × 512 × 512	5	2.47; 0.47; 0.57	3.00; 0.55; 0.98
2048	1024 × 1024 × 512	5	2.61; 0.51; 0.63	3.45; 0.62; 1.32
4096	1024 × 1024 × 1024	6	4.40; 0.81; 1.07	5.48; 0.90; 1.92

of processors. From Table 1a we can clearly see that the ratio is not preserved, leading us to accept that communication dominates computation at large numbers of processors. This is a well-known trend in FFT-based computations.

Fig. 3 is based on Table 1 and shows the parallel efficiency relative to every doubling of processors. Since our problem size is also being doubled, the relative parallel efficiency for a given number of processors is $(m+1)/m$ times the time taken at half the number of processors divided by the time taken at the current number of processors, where $m = k + 21$ on 2^k cores (see the grid sizes shown in Table 2). The parallel efficiency shows some variance, consistent with the data in Table 1. The solid black line shows a relative parallel efficiency of unity. Actual parallel efficiencies that are close to unity indicate good performance. We see from Fig. 3 that CAF and MPI-3 both show good performance.

Table 2b
Shows CAF v/s MPI-3 timings (in seconds) for a single multigrid W-cycle on Blue Waters. These are for variable number of levels and the synchronization of a patch uses the full 27 point stencil.

Number of cores	Zones on finest mesh	Maximum # of levels	CAF total time; interlevel comm. time; halo at level comm. time	MPI-3 total time; interlevel comm. time; halo at level comm. time
8	128 × 128 × 128	3	1.09; 0.07; 0.20	1.10; 0.09; 0.34
16	256 × 128 × 128	3	1.11; 0.07; 0.21	1.17; 0.10; 0.36
32	256 × 256 × 128	3	1.32; 0.08; 0.24	1.38; 0.12; 0.49
64	256 × 256 × 256	4	1.99; 0.16; 0.40	1.95; 0.20; 0.62
128	512 × 256 × 256	4	2.14; 0.24; 0.42	2.24; 0.32; 0.69
256	512 × 512 × 256	4	2.54; 0.29; 0.48	2.58; 0.37; 0.89
512	512 × 512 × 512	5	3.81; 0.47; 0.78	3.83; 0.59; 1.18
1024	1024 × 512 × 512	5	4.32; 0.74; 0.85	4.41; 0.82; 1.34
2048	1024 × 1024 × 512	5	5.26; 0.95; 0.99	5.40; 1.11; 1.89
4096	1024 × 1024 × 1024	6	7.81; 1.33; 1.63	7.94; 1.62; 2.33
8192	2048 × 1024 × 1024	6	9.06; 1.92; 1.75	10.70; 3.37; 2.67
16,384	2048 × 2048 × 1024	6	12.30; 3.61; 2.14	11.64; 3.13; 3.54
32,768	2048 × 2048 × 2048	7	17.68; 4.39; 3.43	19.10; 5.32; 5.17
65,536	4096 × 2048 × 2048	7	21.98; 7.66; 3.53	21.30; 6.09; 5.88

Table 2c
Shows CAF v/s MPI-3 timings (in seconds) for a single multigrid W-cycle on Blue Waters. These are for fixed number (3) of levels and the synchronization of a patch uses the full 27 point stencil.

Number of cores	Zones on finest mesh	Maximum # of levels	CAF total time; interlevel comm. time; halo at level comm. time	MPI-3 total time; interlevel comm. time; halo at level comm. time
8	128 × 128 × 128	3	1.05; 0.09; 0.31	1.15; 0.10; 0.51
16	256 × 128 × 128	3	1.06; 0.09; 0.31	1.23; 0.11; 0.57
32	256 × 256 × 128	3	1.13; 0.09; 0.36	1.32; 0.11; 0.65
64	256 × 256 × 256	3	1.17; 0.09; 0.40	1.51; 0.12; 0.81
128	512 × 256 × 256	3	1.47; 0.11; 0.63	1.47; 0.12; 0.76
256	512 × 512 × 256	3	1.33; 0.10; 0.52	1.50; 0.12; 0.80
512	512 × 512 × 512	3	1.37; 0.10; 0.56	1.85; 0.14; 1.07
1024	1024 × 512 × 512	3	1.96; 0.13; 1.02	1.66; 0.13; 0.92
2048	1024 × 1024 × 512	3	2.07; 0.14; 1.12	1.76; 0.14; 0.99
4096	1024 × 1024 × 1024	3	1.60; 0.11; 0.74	2.13; 0.15; 1.28
8192	2048 × 1024 × 1024	3	1.76; 0.12; 0.84	1.99; 0.15; 1.20
16,384	2048 × 2048 × 1024	3	2.14; 0.14; 1.18	2.61; 0.18; 1.63
32,768	2048 × 2048 × 2048	3	2.69; 0.17; 1.64	2.54; 0.17; 1.67
65,536	4096 × 2048 × 2048	3	2.82; 0.18; 1.85	3.63; 0.16; 1.46

Table 2d
Shows MPI-2 timings (in seconds) for a single multigrid W-cycle on Blue Waters. These are for variable number of levels and the synchronization of a patch uses the full 27 point stencil.

Number of cores	Zones on finest mesh	Maximum # of levels	MPI-2 total time; interlevel comm. time; halo at level comm. time
8	128 × 128 × 128	3	1.67; 0.15; 0.49
16	256 × 128 × 128	3	1.90; 0.17; 0.55
32	256 × 256 × 128	3	2.14; 0.19; 0.72
64	256 × 256 × 256	4	2.77; 0.32; 0.88
128	512 × 256 × 256	4	3.42; 0.42; 1.04
256	512 × 512 × 256	4	3.91; 0.51; 1.36
512	512 × 512 × 512	5	5.18; 0.73; 1.71
1024	1024 × 512 × 512	5	7.21; 1.38; 2.16
2048	1024 × 1024 × 512	5	7.57; 1.23; 2.63
4096	1024 × 1024 × 1024	6	10.44; 1.87; 3.35
8192	2048 × 1024 × 1024	6	17.39; 3.30; 6.03
16,384	2048 × 2048 × 1024	6	17.59; 3.58; 6.22
32,768	2048 × 2048 × 2048	7	28.42; 6.65; 9.37
65,536	4096 × 2048 × 2048	7	46.25; 9.98; 16.67

4.2. Multigrid methods

There are two types of multigrid processing that are popularly used, the V-cycle and the W-cycle. The V-cycle visits all levels twice and only twice per cycle. I.e. each level is processed once when traversing the multigrid hierarchy from finest to coarsest mesh and then once again when traversing the hierarchy from coarsest to finest mesh. For serial processing, the V-cycle is not very advantageous because it only reduces the residual by a small factor. Multiple V-cycles have to be carried out to reduce the residual to a value that is below discretization error. Since much of the reduction in the residual comes from having a converged solution on coarser meshes, the W-cycle emphasizes larger number of visits to the coarser levels, with the coarsest level receiving the largest number of visits. In serial setting, this can result in an order of magnitude reduction in the residual in one W-cycle. A very small number of W-cycles are needed to reduce the residual to a value that is below discretization error. Consequently, this form of multigrid processing is very advantageous in serial processing.

In a parallel context, the advantage may shift away from the W-cycle. This is because the coarser levels involve far fewer floating-point computations but almost as much communication time (dominated by latency). Consequently, in some implementations, it is advantageous to replicate the processing of coarser levels on multiple sets of processors. In the interest of simplicity, we have not done that here. Similarly, it is advantageous to increase the number of iterations at a given level so as to maximize convergence at any given level. We have used eight Jacobi relaxation steps at each level traversal, whether it is an upward or downward traversal. The settings in the Jacobi relaxation step were optimized for convergence in keeping with the prescription of Yavneh [20,21].

Table 2a shows CAF v/s MPI-3 timings (in seconds) for a single multigrid W-cycle on Darter. Because we use an oct-tree approach, 32^3 zone patches were used to build the mesh at any level. A 27 point stencil was assumed because the target application is likely to use a full stencil. Times reported here represent the average of ten calls to the multigrid routine from the finest mesh. The total time in seconds for one call to the full multigrid W-cycle is shown. We also show the time that was spent (during each full W-cycle) for inter-level transfers of data during the restriction and prolongation steps. It is also useful to show the time spent (during each full W-cycle) for halo exchanges at all levels. These last two times are useful because they give us a clear breakdown of the time spent in messaging. We see that a modest fraction of the time was indeed spent for messaging. That fraction increases with increasing number of levels because increasing the number of levels in a W-cycle also inevitably increases the number of visits to the coarser levels. We have also repeated the results of Table 2a when a 7 point stencil was used and found no significant difference. Table 2b is similar to Table 2a but shows a slightly larger weak scalability study on Blue Waters. On Darter we do find that CAF has a small advantage over MPI-3. The processors on Blue Waters are, on average, half as fast as those on Blue Waters. This is not surprising considering that Darter uses modern Xeons whereas Blue Waters uses Operons of an older vintage. The FFT results in Table 1 also show a comparable trend. Table 2b shows conclusively that CAF is entirely competitive with MPI-3 even on many tens of thousands of processors.

In Tables 2a and 2b we coarsen the meshes till the coarsest multigrid level has a side with only 32 zones. For a real application one would not coarsen the multigrid meshes too much. Instead, one would use a very important suggestion from Gropp, E. Lusk and A. Skjellum [12]. The suggestion is to only use a few levels of multigrid along with the use of a direct solver on the coarsest level. The results from such a simulation are shown in Table 2c, where only three levels of multigrid processing were used. We now see that CAF again has a small advantage over MPI-3 on Blue Waters and that advantage is sustained as the number of processors is increased. Table 2d is analogous to Table 2b, with the only difference that it was based on MPI-2. We clearly see that the one-sided, non-blocking messaging model in CAF and MPI-3 enables these modern programming paradigms to out-perform MPI-2 by a wide margin on large numbers of processors.

Fig. 4 shows the parallel efficiency relative to a doubling of processors. This is shown as a function of increasing numbers of processors for CAF and MPI-3 applied to three-levels of multigrid processing. The solid black lines in Fig. 4 show a relative parallel efficiency of unity, which is optimal. Actual parallel efficiencies that are close to unity indicate good performance. Fig. 4(a) pertains to a 27 point stencil while Fig. 4(b) pertains to a 7 point stencil. (We have obtained the timing data for a 7 point stencil but we have not shown the timing tables here for the sake of brevity.)

Table 2d shows the performance of MPI-2 on Blue Waters. As with the FFT data, it is quite significant that MPI-2 underperforms both CAF and MPI-3 by a very substantial margin. This underscores the immense utility of the new approaches catalogued here.

4.3. Computational fluid dynamics application – MHD as an example

We used a second order MHD code that was based on a time-explicit higher order Godunov methodology. Such applications rely on halo exchanges between nearest neighboring blocks of data. While the minimal halo exchange would call for a halo made of two zones, our application code uses a halo of four zones. Typically such applications utilize several hundred to several thousand float point operations per zone. As a result, this use of a larger set of halo zones did not constitute a problem. Table 3a shows CAF v/s MPI-3 timings (in seconds) for a single fluid timestep on Blue Waters. (In practice, the average from ten timesteps is shown.) Fig. 5 shows the parallel efficiency relative to a doubling of processors. This is shown as a function of increasing numbers of processors for CAF and MPI-3 applied to the MHD code. The solid black line in Fig. 5 shows a relative parallel efficiency of unity, which is optimal. We see that both CAF and MPI-3 operate at the highest levels of parallelism with comparable parallel efficiency. Furthermore, that efficiency is close to optimal even for the largest

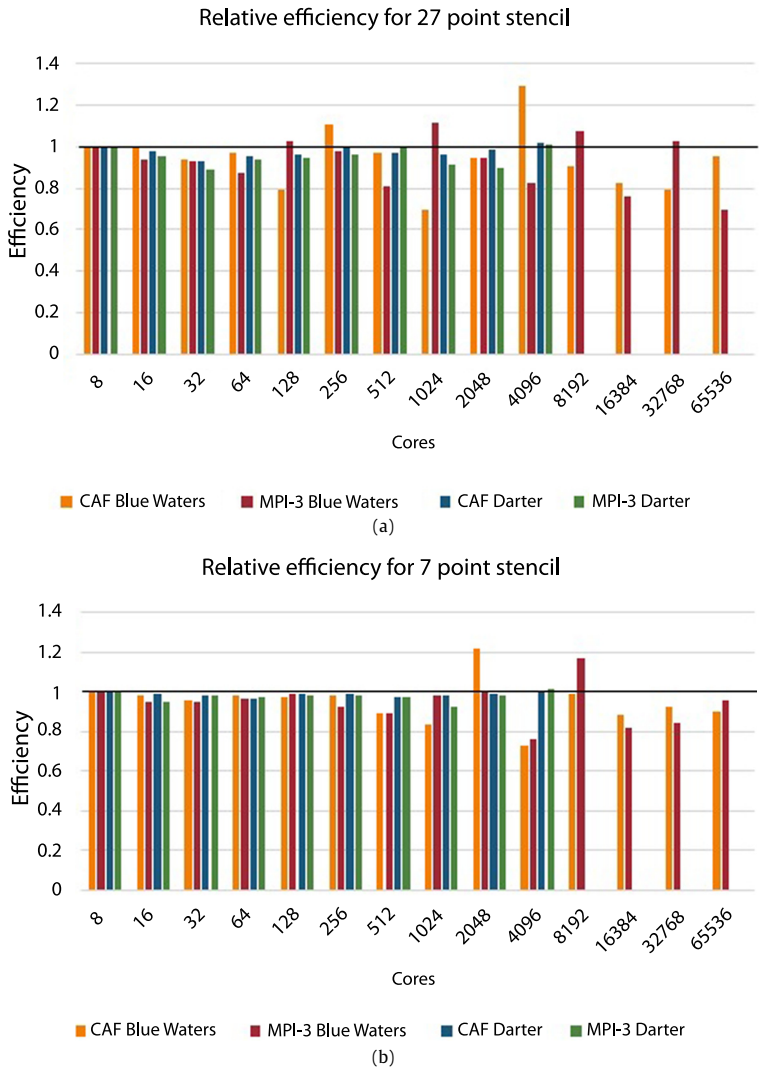

Fig. 4. Shows the parallel efficiency measured relative to every doubling of processors for CAF and one-sided MPI-3. The results are based on 3-level Multigrid simulations from Table 2. Results for a 27 point stencil are shown in (a); results for a 7 point stencil are shown in (b). We see that the stencil width only has a modest impact on parallel efficiency.

numbers of processors. By scanning the messaging times in Table 3a we see that the one-sided messaging in CAF is slightly more efficient that the one-sided messaging from MPI-3 for all numbers of cores that we tested. The same simulations were also run with MPI-2 and the results are shown in Table 3b. We see that CAF and MPI-3 show a significant improvement over MPI-2.

5. Conclusions

CAF and MPI-3 represent new paradigms for one-sided messaging that are especially well-adapted to advanced PetaScale and future ExaScale architectures. This style of messaging has the potential of reducing messaging time and enhancing performance on those architectures. CAF is a language-based approach and MPI-3 is a library-based approach; both approaches to parallelism have their unique advantages. CAF has become available via several compiler vendors and the MPI-3 library, with some of the newer one-sided messaging features, has also become available. It is, therefore, interesting to compare CAF and MPI-3 for a few algorithms that are routinely used to solve partial differential equations on structured meshes. We have compared the performance CAF with MPI-3 for spectral techniques, multigrid techniques and for applications drawn from computational fluid dynamics.

By using code that is identical on all counts except for the messaging, we are able to focus on the messaging capabilities of these two parallel programming paradigms. Weak scalability studies are presented on up to 65,536 processors. Both paradigms show excellent scalability that is sustained on large numbers of processors, showing them to be well-suited for PetaScale applications. On some applications, CAF outperformed MPI-3 by a small margin. On other applications, they drew

Table 3a
Shows CAF v/s MPI-3 timings (in seconds) for MHD on Blue Waters.

Number of cores	Zones on finest mesh	CAF total time; messaging time	MPI-3 total time; messaging time
8	48 × 48 × 48	1.04; 0.02	1.07; 0.03
16	96 × 48 × 48	1.04; 0.02	1.08; 0.03
32	96 × 96 × 48	1.04; 0.02	1.08; 0.03
64	96 × 96 × 96	1.05; 0.03	1.09; 0.04
128	192 × 96 × 96	1.05; 0.03	1.09; 0.04
256	192 × 192 × 96	1.07; 0.03	1.10; 0.04
512	192 × 192 × 192	1.07; 0.03	1.10; 0.05
1024	384 × 192 × 192	1.07; 0.04	1.11; 0.05
2048	384 × 384 × 192	1.09; 0.05	1.12; 0.06
4096	384 × 384 × 384	1.10; 0.06	1.15; 0.08
8192	768 × 384 × 384	1.14; 0.10	1.20; 0.13
16,384	768 × 768 × 384	1.20; 0.16	1.22; 0.15
32,768	768 × 768 × 768	1.22; 0.18	1.29; 0.22
65,536	1536 × 768 × 768	1.23; 0.19	1.29; 0.21

Table 3b
Shows MPI-2 timings (in seconds) for MHD on Blue Waters.

Number of cores	Zones on finest mesh	MPI-2 total time; messaging time
8	48 × 48 × 48	1.66; 0.05
16	96 × 48 × 48	1.65; 0.05
32	96 × 96 × 48	1.67; 0.05
64	96 × 96 × 96	1.70; 0.06
128	192 × 96 × 96	1.74; 0.06
256	192 × 192 × 96	1.78; 0.07
512	192 × 192 × 192	1.75; 0.07
1024	384 × 192 × 192	1.79; 0.07
2048	384 × 384 × 192	1.81; 0.09
4096	384 × 384 × 384	1.81; 0.09
8192	768 × 384 × 384	1.88; 0.15
16,384	768 × 768 × 384	1.89; 0.16
32,768	768 × 768 × 768	1.90; 0.16
65,536	1536 × 768 × 768	1.96; 0.22

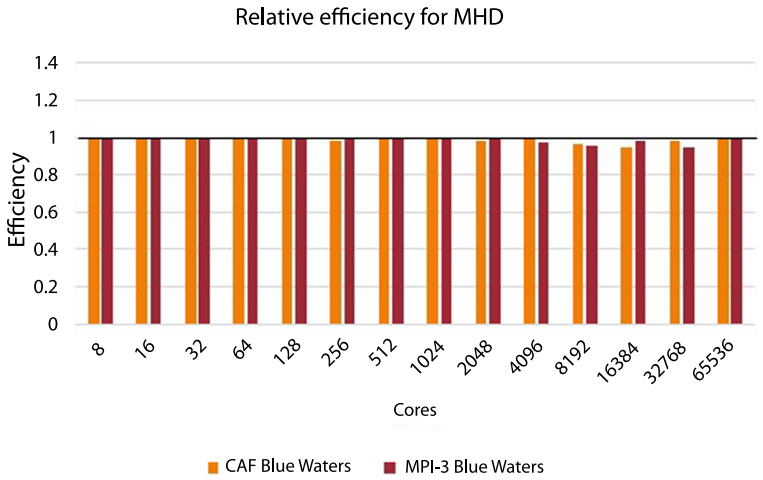

Fig. 5. Shows the parallel efficiency relative to every doubling of processors for CAF and one-sided MPI-3. The results are based on MHD simulations from Table 3a.

level. Both CAF and MPI-3 outperformed the older MPI-2 by a wide margin in all our applications. This was especially true as the numbers of processors were increased. The excellent scalability results reported here for CAF and MPI-3 suggest that they are both robust paradigms for parallel computing that should be developed for future ExaScale computing platforms.

In addition to the scalability studies, we also catalogue some of the best-usage strategies that we have found for our successful implementations of one-sided messaging in CAF and MPI-3. We find that both paradigms require entirely analo-

gous restructuring of code. However, we find that CAF has some advantages when it comes to ease of implementation and use. It also offers greater transparency in the way in which the parallelism is expressed.

Acknowledgements

DSB acknowledges support via NSF grants NSF-AST-1009091, NSF-ACI-1307369 and NSF-DMS-1361197. DSB also acknowledges support via NASA grants from the Fermi program as well as NASA-NNX 12A088G. Computer support on NSF's XSEDE and Blue Waters computing resources is also acknowledged. All three authors gratefully acknowledge the cheerful help and insightful advice provided by Bill Long and Pete Mendygral of Cray.

Appendix A

A textbook cataloguing the features of MPI-3 is not available at the time of this writing. Because MPI-3 is not yet well-known, we will show here how it can be used to get the same effect as the CAF statement:

```
temp(:,:,:) = buff(:,:,:,me)[you]
```

in Section 3.2. Here "*you*" is the image number of the remote processor from which we wish to do a one-sided get in CAF. To perform the same task in MPI code requires the creation of a window for the remote array:

```
call MPI_WIN_CREATE (buff, window_size, &
     size_of_real, MPI_INFO_NULL, MPI_COMM_WORLD, &
     buff_handle, ierr)
```

where the integer *window_size* holds the array size in bytes, the integer *size_of_real* holds the size of an array element in bytes, *buff_handle* is an integer that is needed whenever the array is accessed remotely, and *ierr* is an integer whose value will indicate whether the call is successful. Given that this call has been made successfully, the following code gives the same effect as the line of CAF code at the start of this appendix:

```
call MPI_WIN_LOCK (MPI_LOCK_EXCLUSIVE, you − 1, 0, buff_handle, ierr)
call MPI_GET (buff, size(temp), MPI_REAL, you − 1, size(temp) ∗ (me − 1), &
              size(temp), MPI_REAL, buff_handle, ierr)
call MPI_WIN_UNLOCK (you − 1, buff_handle, ierr)
```

Our practical experience has been that a call to *MPI_WIN_CREATE* is very time-consuming. As a result, it is beneficial to make all the calls to this MPI routine only once at start-up. The code has to be structured so that one can minimize the number of windows created. The data structures that we have described in this paper, once they are shorn of their coarray subscript, can be used in this fashion.

Compilation lines on Cray machines for MPI-3 codes are the following:

```
module load craype-hugepages2M
ftn -x omp -default64 -o xtest file.F -Wl,–whole-archive,-ldmapp,–no-whole-archive
```

At runtime, we add the following lines in our job submission script file:

```
module load craype-hugepages2M
setenv MPICH_RMA_OVER_DMAPP 1
aprun -n Np -j 1 -r 1 -ss./xtest
```

where *Np* is the number of cores. -r 1 reserves cores for the OS and stops that activity from interrupting user threads. -ss confines memory allocation to the NUMA domain of the user threads, which is a big deal on Interlagos processors.

Appendix B

Compilation lines on Cray machines for CAF codes are the following:

```
module load craype-hugepages2M
ftn -x omp -h caf -default64 -o xtest file.F
```

At runtime, we add the following lines in our job submission script file:

module load craype-hugepages2M

aprun -n Np -j 1 -r 1 -ss./xtest

where *Np* is the number of cores.

References

[1] D.S. Balsara, Second-order-accurate schemes for magnetohydrodynamics with divergence-free reconstruction, Astrophys. J. Suppl. Ser. 151 (2004) 149–184.
[2] D.S. Balsara, Divergence-free reconstruction of magnetic fields and WENO schemes for magnetohydrodynamics, J. Comput. Phys. 228 (2009) 5040–5056.
[3] D.S. Balsara, Self-adjusting, positivity preserving high order schemes for hydrodynamics and magnetohydrodynamics, J. Comput. Phys. 231 (2012) 7504–7517.
[4] I. Bethune, J.M. Bull, N.J. Dingle, N.J. Higham, Performance analysis of asynchronous Jacobi's method implemented in MPI, SHMEM and OpenMP, Int. J. High Perform. Comput. Appl. 28 (1) (2014) 97–111.
[5] A. Brandt, Multi-level adaptive solutions to boundary-value problems, Math. Comput. 31 (1977) 333–390.
[6] A. Brandt, Multi-level adaptive techniques (MLAT) for partial differential equations: idea and software, in: Mathematical Software III, Academic Press, New York, 1977, pp. 277–318.
[7] W. Briggs, V.E. Henson, S. McCormick, A Multigrid Tutorial, second edition, SIAM, 2000.
[8] C. Canuto, Spectral Methods in Fluid Dynamics, Springer Ser. Comput. Phys., Springer-Verlag, 1991.
[9] C. Canuto, M. Hussaini, A. Quarteroni, T. Zang, Spectral Methods in Fluid Dynamics, Springer-Verlag, 1993.
[10] A. Fanfarillo, T. Burnus, V. Cardellini, S. Filippone, D. Nagle, D. Rouson, OpenCoarrays: open-source transport layers supporting coarray Fortran compilers, Conference Paper, PGAS 2014, http://nic.uoregon.edu/pgas14/papers/pgas14_submission_7.pdf.
[11] V. Gramoli, D. Harmanci, On the input acceptance of transactional memory, Parallel Process. Lett. 20 (1) (2010) 31–50.
[12] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message Passing Interface, 2nd edition, MIT Press, 1999.
[13] S.K. Lele, Compact finite difference schemes with spectral-like resolution, J. Comput. Phys. 103 (1992) 16–42.
[14] M. Metcalf, J. Reid, M. Cohen, Modern Fortran Explained, Oxford University Press, 2011.
[15] MPI Forum 2012, MPI: A Message Passing Interface Standard, Version 3.0, http://www.mpi-forum.org, 2012.
[16] R.W. Numrich, J. Reid, K. Kim, Writing a multigrid solver using co-array Fortran, in: Applied Parallel Computing Large Scale Scientific and Industrial Problems, in: Lect. Notes Comput. Sci., vol. 1541, 1998, pp. 390–399.
[17] C. Temperton, A generalized prime factor FFT algorithm for ANY $N = (2^P)(3^Q)(5^R)$, SIAM J. Sci. Stat. Comput. 13 (May 1992) 676–686.
[18] U. Trottenberg, C. Oosterlee, A. Schuller, Multigrid, Elsevier Academic Press, 2001.
[19] C. Yang, W. Bland, J. Mellor-Crummey, P. Balaji, Portable, MPI-interoperable coarray Fortran, in: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'14, 2014, pp. 81–92.
[20] I. Yavneh, Multigrid smoothing factors for red–black Gauss–Seidel relaxation applied to a class of elliptic operators, SIAM J. Numer. Anal. 32 (1995) 1126–1138.
[21] I. Yavneh, On red–black SOR smoothing in multigrid, SIAM J. Sci. Comput. 17 (1996) 180–192.