Migrating a Scientific Application from MPI to Coarrays

John V Ashby *and* **John K Reid,** *STFC Rutherford Appleton Laboratory*

ABSTRACT: *MPI is a de facto standard for portable parallel programming using a message passing paradigm. Interest is growing in other paradigms, in particular Partitioned Global Address Space (PGAS) languages such as Coarray Fortran, UPC and Titanium. Most Computational Science and Engineering codes are written in Fortran, and the 2008 Fortran standard will include coarrays, a Cray initiated PGAS extension of the language. We report on the experience of taking a moderately large CFD program and migrating it to a Cray X1E using coarrays rather than MPI. The MPI and coarray versions are compared, both for ease of programming and legibility, and for performance. We find that coarrays are a useful and expressive addition to Fortran, and that their use does not impact on performance. We discuss various ways in which the use of coarrays can be optimized in a program.*

KEYWORDS: Coarrays, Fortran, PGAS languages, MPI

1. Introduction

The development of scientific computing has been accompanied over many decades by the development of programming languages. With the advent of parallel and vector computing, an early goal was the production of automatic parallelizing compilers. This proved difficult in the framework set by existing languages, though automatic vectorization was successful. Several attempts were made to add seamlessly to a language by means of directives that a compiler was free to act on or not. Typical of these were High Performance Fortran (HPF) [1] and OpenMP [2], which included directives that could be treated as comments by a compiler, or could give suggestions for ways to exploit parallelism within the code. However, directive based parallelization was found to be insufficiently rich to describe the algorithms that were emerging, and the message passing paradigm became established. Here a library of routines is supplied (MPI [3] and PVM [4] are examples) by means of which processes can communicate and cooperate. Another paradigm which may be appropriate depending upon the hardware platform is SHMEM [5]. In either case, the functionality of the library is defined outside any specific programming language, and a language binding is supplied as an API.

Recently a return to incorporating parallelism within programming languages has occurred, with Coarray Fortran [6], UPC [7] and Titanium [8] seeking a small set of additions to existing languages (Fortran 95, C and Java respectively) based on the Partitioned Global Address Space model. In these languages, the communications are largely abstracted out, leaving the programmer writing code that could work equally well on shared or distributed memory architectures. A review of these is given in Ashby [9].

Ideas from these developments are being incorporated in the next generation of languages such as Chapel [10], X10 [11] and Fortress [12], discussed by Ashby [13].

1.1 Concepts of parallel programming

A serial computer program consists of a set of instructions executed sequentially which transform input data to output data, generating temporary intermediate data in the process. A parallel computer program consists of a set of serial programs which execute simultaneously. In some cases these may be largely independent; in process farming, for example, some region of a parameter space is explored by multiple instances of a program running on slightly different initial data to find some optimum output. In other cases, each program needs knowledge of the calculations from the other programs and they need to exchange data regularly. Furthermore, these programs may be the same (SPMD – Single Program Multiple Data) as when a finite difference solution to a partial differential equation is divided among several processors, each working on a part of the domain of the equation, or different (MPMD – Multiple Program Multiple Data), for example a multiphysics aeroelasticity calculation where the flow over a wing is calculated, the pressure field passed to an elastic solver which calculates the deformation of the wing and the new shape is passed back to determine the flow field. Synchronization will be needed to ensure that the data has been calculated.

Each program in a parallel set may need data produced by another member. To access that data it must know a) on which other instance the data resides, b) how that program refers to it and c) that its value has been calculated. In addition, the other program may need to know that it will be asked for the data.

The design of Coarray Fortran was driven by the question "What is the smallest change needed to convert Fortran 95 into a robust, efficient parallel language?" [6]. It achieves this by a simple syntactic extension to the language, an extension which maintains the style of Fortran. Since first being proposed in the late 1980s, Coarray Fortran has been developed into a part of the new Fortran standard currently under discussion for 2008 [14, 15].

With the forthcoming standardization of Coarray Fortran, we consider what advantages it offers the programmer and the feasibility of migrating an existing code from MPI, the de facto standard for distributed memory architectures. We are particularly concerned with the impact on programmer efficiency in terms of the ease of development and maintenance of coarray code, but we do not overlook run-time performance.

2. The Basics of Coarray Fortran

2.1 SPMD

Like MPI, Coarray Fortran is based on a Single Program, Multiple Data model. The program is replicated a fixed number of times (usually chosen on the command line) and each replication (*image* in coarray language) has its own set of variables and mostly executes asynchronously.

The number of images is available as the intrinsic function **`num_images()`**. Each image has an image index in the range **`1:num_images()`**, which is available as an intrinsic function **`image_index()`**.

2.2 Data

Coarray Fortran departs from MPI in that objects can be declared as coarrays, which means that as well as being accessible locally in the normal way, they can be accessed from other images, for example:

```
real a(10),b(10)[*]
a(5) = b(5)[2]
b(1)[4] = a(3)
```

where arrays **`a`** and **`b`** exist on every image and can be accessed normally there, but only **`b`** can be accessed directly from another image.

The cosubscripts in square brackets here refer to image indices, but another range may be chosen by including a lower bound in the declaration, e.g.

```
real b(10)[0:*]
```

and the image indices may be mapped to several Cartesian coordinates by declaring several cosubscripts:

```
real b(10)[nx,ny,*]
```

There is an intrinsic **`image_index(b,cosubs)`** for finding the index of the image that is referenced by the set of cosubscripts in the array **`cosubs`** when used for the coarray **b.**

Coarrays are restricted to having the same shape and bounds on each image. This permits implementations to store them at the same location on each image and allows an image to calculate addresses on other images. Where this is too restrictive, the coarray may be of a derived type that has an allocatable array component:

```
type co_double_2
double precision,allocatable :: array(:,:)
end type co_double_2
integer nx,ny
type (co_double_2) vel[*]
    :
allocate(vel%array(nx,ny))
```

where **nx** and **ny** are local variables whose value varies from image to image.

2.3 Synchronization

Almost all synchronizations are explicit. Here, we will use only the simplest

```
sync_all
```

whose meaning is obvious. Between synchronizations, the system is free to assume that no data accessed by an image is altered by another image and can use all its usual optimizations. If data is altered by one image and accessed by another, it is necessary for there to be synchronization after the alteration and before the access.

2.4 Coexistence with MPI

Since MPI uses the same SPMD model, it is easy to have MPI and coarrays exist together in a program. We have made use of this to convert gradually from MPI to coarrays. We had to take account of the fact that MPI indexes its processors from zero (we gave many of our coarrays a lower cobound of 0 to ease conversion) and that Cartesian coordinates in MPI are taken as indexed from zero and mapped to processors counting most rapidly in the last dimension rather than the first.

3. The Application: SBLI

To investigate the use of coarrays in a real application code, rather than a simple kernel, we have taken the SBLI code (also known as PDNS3D, [16]) - a finite difference formulation of Direct Numerical Simulation (DNS) of

Turbulence from the University of Southampton, UK, and migrated it to use coarrays.

Fluid flows encountered in real applications are often turbulent. There is, therefore, an ever-increasing need to understand turbulence and, more importantly, to be able to model turbulent flows with improved predictive capabilities. As computing technology continues to improve, it is becoming more feasible to solve the governing equations of motion — the Navier-Stokes equations — from first principles. The direct solution of the equations of motion for a fluid, however, remains a formidable task and simulations are only possible for flows with small to modest Reynolds numbers. Within the UK, the Turbulence Consortium (UKTC) has been at the forefront of simulating turbulent flows by direct numerical simulation. UKTC has developed a parallel version of a code to solve problems associated with shock/boundary layer interaction.

The code SBLI was originally developed for the Cray T3E and is a sophisticated DNS code that incorporates a number of advanced features: namely high-order central differencing; a shock-preserving advection scheme from the total variation diminishing (TVD) family; entropy splitting of the Euler terms and the stable boundary scheme. The code has been written using standard Fortran 90 code together with MPI in order to be efficient, scalable and portable across a wide range of high-performance platforms. The PDNS3D benchmark is a simple turbulent channel flow benchmark using the SBLI code. This benchmark comes in three sizes with meshes of 120 cubed, 240 cubed and 360 cubed. The code is very heavy in its use of memory, and so for the small numbers of processors available to us we were only able to exercise the first two of these.

The most important communications structure within SBLI is a halo exchange between adjacent computational sub-domains. Providing the problem size is large enough to give a small surface area to volume ratio for each sub-domain, the communications costs are small relative to computation and should not constitute a bottleneck. This is the case for our benchmarks.

4. Parallel operations in SBLI

4.1 The Process

When migrating software to use the new features offered by developments in the language, the process by which elements of the code are identified for migration and the decisions made as to how to deal with them can be very important. In this case, we first performed a paper walk-through of the code, identifying the broad functionality of each procedure and noting particularly any which made use of MPI routines. The parallel portions of SBLI fell into five parts.

1. First, the various pieces of global data are read in by one master process and broadcast to all others. These include physical data such as the Reynolds Number, ambient pressure and specific heat ratio, γ, algorithmic data such as time step size, choice of TVD scheme and turbulence model, and control data such as the frequency with which to save restart files.
2. Next, the global mesh is read by the master process and portions of it sent to other processes to form local meshes. The mesh partitioning, which is a recursive geometric bisection, may produce different sized local meshes on different images.
3. Some quantities are set across all processes by a collective operation; the wall temperature, for example, is set using **`mpi_allreduce`** to find the minimum across the processes.
4. The bulk of the parallel work in SBLI involves the halo exchanges and the enforcement of periodic boundary conditions. The code is well structured and performs these exchanges in a handful of routines.
5. There is the gathering of data to write a solution and routines to re-read and distribute this data to enable a restart. This I/O element of the software has two optional solutions: gathering the data onto the master process and writing a file from there or using MPI-IO to write (and read) the data in parallel.

Throughout the migration, the code was run and tested at suitable breakpoints. The final results were the same to the seven decimal digits printed.

4.2 Broadcasting global data

In the MPI code, the various pieces of data such as the number of grid points, Reynolds number, choice of turbulence model, etc. are read in by the master process, then packed into real, integer and logical buffer arrays (here and elsewhere we use real to denote floating point numbers; in the application itself they are double precision). These buffers are sent using **`mpi_bcast`** to all other processes, where they are unpacked. The resulting code is:

```
if (master) then
    r(1) = reynolds
    ...
    r(18) = viscosity
    call mpi_bcast(r, 18, real_mp_type, &
            masterid, MPI_comm_world, ierr)
else
    call mpi_bcast(r, 18, real_mp_type, &
            masterid, MPI_comm_world, ierr)
    reynolds = r(1)
    ...
    viscosity = r(18)
endif
```

There is a more elegant way to code this, but this is the way the original code used.

The simplest coarray version uses the master image to store the data into all other images local copies:

```
sync all
reynolds = reynolds[masterid]
...
viscosity = viscosity[masterid]
```

Note that the references on the left hand side of the assignments have no coindex. They refer to the data in the local image. The synchronization ensures that an image does not try to acquire data from the master before it is available. The MPI version has an implicit synchronization within `mpi_bcast`.

The coarray code has the advantage that what is being transferred is transparent and there is no need to check that the packing and unpacking are in correspondence. This makes adding a new variable less prone to programmer error.

Another possible approach would be to use the master image alone to put the data onto the remote images, thus:

```
sync all
if (master) then
    do i=1, num_images()-1
            reynolds[i] = reynolds
            ...
            viscosity[i] = viscosity
    end do
end if
sync all
```

The disadvantages of this approach are that it is inherently serial and that it requires two synchronizations. Two synchronizations are needed; the first stops the master trying to write to data on a remote image before it has been instantiated and initialised, the second stops the remote images from continuing execution before the data has been set by the master.

We ran some timing tests on the Cray X1E to which we have access and found that there was no improvement in speed if we packed all the data into an array, copied that and then unpacked it. This approach might allow more efficient communication since there would be only one transfer. It is possible that the compiler is doing this automatically. This simple case illustrates the scope for optimization of communication that Coarray Fortran provides.

4.3 Distributing Partitioned Data

The global mesh data, coordinates and the derivatives to map from computational space to physical space are read onto the master process and partitioned according to the number of processes available. This can result in some local arrays being of differing lengths on different processes. We chose to handle this in Coarray Fortran by making the arrays be components of a coarray of derived type. We have introduced a set of user-defined types, `co_double_1` to `co_double_4`, where the final digit labels the rank of the component array. In the main code an array, `q`, would be defined as:

```
type(co_double_4)::q[*]
```

and accessed by

```
q%array(i,j,k,l)
```

on the local image or by:

```
q[procn]%array(i,j,k,l)
```

on image `procn`.

An alternative would be to have the coarrays declared allocatable and allocate them to the maximum size in each dimension. There is some advantage to this in code recognition and in minimizing the number of different lines in the code between the two versions, which might be important if MPI and coarray versions are to be maintained side-by-side. However, it does represent an overhead in memory use, particularly in codes which may be load imbalanced. The extra work of indirection through the component references is found to be small.

In the main program it is, of course, impossible to declare the co-dimension as `[nx,ny,*]` as would be desirable, since `nx` and `ny` depend on the partitioning of the total number of images which is not known at compile time. This limitation can be overcome by dynamically re-coshaping the coarrays when they are passed to subroutines as arguments.

```
type(co_double_4),dimension[*]::q
call distrib(q)
...
subroutine distrib(q)
use parallel_mod, only : nx, ny
type(co_double_4)::q[nx,ny,*]
```

This replaced the functionality provided in the MPI version by using Cartesian communicators. Our original migration retained the use of these to manage the image topology, in the spirit of incremental change, but in the final version we moved to re-coshaping to use the full power of the coarray syntax. As we see in section 4.5, this has advantages for clarity of expression.

The data are distributed using routines which encapsulate the process, so that the specific means of achieving the parallel communication is held in a very few places in the code rather than being widespread. The MPI algorithm is similar to that for broadcasting global data, except that the required region of the global data must be identified:

```
if (master)
    for each process j
            find start and end indices of array for j
            pack global array(start:end) to buffer
            send buffer to process j
    next
else
    receive buffer
    unpack to local array
end if
```

For the coarray version, we still need to find the region of the global array, but the sending is simpler:

```
if (master)
    for each image j
            find start and end indices of array for j
            local(:)[j]=global(start:end)
    next
end if
```

In this case, the images are sitting idle while the master image farms out the data. We call this the “push” version where data are pushed from the master out to the worker images. Alternatively each worker image could “pull” the data thus:

```
find start and end indices on this image
```

local(:)= global(start:end) [masterid]

There is scope here for parallelism if the memory access mechanism will support it.

As in section 4.2, synchronization is important in this routine and must be explicitly added.

The routines for distribution in the original code distributed both rank 3 and rank 4 arrays, being called with the size of the fourth dimension set to 1 in the case of rank 3 arrays. This would not work when the arrays were translated into `co_double_3` and `co_double_4` objects, so a generic interface was built in order to preserve both the structure and the exact statements of the original code as far as possible. Experience with code transformation has shown that it is important to disturb the appearance of the code as little as possible so that the authors can still recognise it. Failure to do so can lead to a loss of maintainability. Improvement of this is one of our goals.

4.4 Collective operations

Collectives for coarrays have been proposed but are currently not available in Cray Fortran and are not intended to be part of the Fortran 2008 standard, though they are expected to be included in a Technical Report. We wrote our own simple `co_min` function (which returns the minimum over the images of a co-scalar). This is not a true collective, but runs on each image. The synchronizations ensure the same result is returned on all images.

```
double precision function co_min(a)
double precision a[*]
integer i
sync_all
co_min = a
do i=1, num_images()
    if (a[i] .lt. co_min) then
            co_min = a[i]
    end if
end do
sync_all
end function co_min
```

This is clearly not an efficient implementation but was suffifcient for our purposes where it is only called once during the program's run.

In the same way, the current Cray Fortran implementation does not include the `image_index()` function, which is a recently proposed addition to the coarray language, so we wrote a simple emulator for this.

4.5 Halo Exchange

At the heart of the time stepping algorithm is a set of routines for halo exchange in each of the three cartesian directions. Again, these are used for rank 3 and 4 arrays and generic interfaces were required. The MPI approach is the familiar one of packing data into a buffer, sending to the appropriate neighbour and the neighbour unpacking it into the correct place in the local array.

The data sent are the values on the interior portion of the mesh which form the halo region on a neighbouring process. In the coarray version, simple coaddressing is used. If `nx` holds the extent of the local mesh in the *x*-direction, then to send the data from an image to the image one x-step lower (`procmx`):

```
nxp=nx[procmx]
do k=1, nz
do j=1, ny
    a[procmx]%array(nxp+1:nxp+xhalo,j,k)&
    = a%array(1:xhalo,j,k)
end do
end do
```

Using the re-coshaping facility gives code which is more transparent. In the following routine which performs a halo exchange in the negative *x*-direction, using the array `d` to contain the location of the current image in the three dimensional grid of images allows the image one step below in x to be directly addressed:

```
subroutine exch_x_minus(a,nx)
use parallel_mod, only : nxim, nyim
type(co_double_3)::a[nxim,nyim,*]
integer nx[nxim, nyim,*]
integer d(3), nxp
d = this_image(a)
if (d(1) .gt. 1) then
    nxp = nx[d(1)-1,d(2),d(3)]
    a[d(1)-1,d(2),d(3)]%array(nxp+1:&
    nxp+xhalo,:,:) = a%array(1:xhalo,:,:)
end if
end subroutine exch_x_minus
```

In the MPI code `nx` was not passed through the argument list but was instead accessed from the `parallel_mod` module (which also contains the definition of the `co_double_3` type). To keep the interface to the exchange routines consistent between the two versions, we were forced to do likewise and so `nx` could not be re-coshaped. We were able to use the `image_index()` function to overcome this:

```
subroutine exch_x_minus(a)
use parallel_mod, only : nxim, nyim,nx
type(co_double_3)::a[nxim,nyim,*]
integer d(3), nxp, imgxm
d = this_image(a)
if (d(1) .gt. 1) then
    imgxm=image_index(a,(/d(1)-1,d(2),d(3)/))
    nxp = nx[imgxm]
    a[d(1)-1,d(2),d(3)]%array(nxp+1:&
    nxp+xhalo,:,:) = a%array(1:xhalo,:,:)
end if
end subroutine exch_x_minus
```

4.6 Input and output

Since the routines which use MPI-IO are complex we opted to continue with MPI for these routines at present, in keeping with our incremental approach. This choice does not affect our results on performance since the full solution is not written to disk in our experiments.

5. Results

5.1 Clarity

The measures of interest when assessing the success of the transformation described in section 4 are first of all the clarity and readability of the transformed code and secondly the performance in terms of speed and scalability.

Clarity and readability are subjective, but by retaining as much of the original code as possible through the use of generic functions, we maintain a high level of recognisability which plays a large part in making the sort of code transformation we have done acceptable to the principal developers. Since the original code used pre-processor directives to provide various optional compilation alternatives we have included the coarray version as such an option. This has the added advantage that we can be as sure as possible that the comparison of MPI vs coarrays is being carried out on the same code.

It is clear from the examples given above that coarrays are capable of expressing a parallel algorithm in a significantly simpler style than MPI. The advantages are that the same code can more often be used for all images and there is no need to write separate code for sending and receiving messages. The requirement that we retain the structure of the code has occasionally entailed a loss of clarity, as, for example, when we were constrained from changing the argument list for certain subroutines and so needed to use the `image_index()` function. In a code being written from scratch or where backwards compatibility was less of an issue, still more clarity could be obtained.

One objective measure of clarity is the number of lines of code required to encapsulate a given functionality. In this respect coarrays were successful. The MPI version of the routine which performs halo exchange in the *x*-direction is 176 non-comment lines long. The same functionality is achieved in coarrays in 105 lines, in spite of being split over two subroutines to accommodate rank 3 and rank 4 objects. Similar statistics apply to the other core routines. For example the MPI routine which broadcasts global parameters is 230 lines, whereas the coarray version is only 117.

The response from SBLI's current developer was very positive. His major qualm was about the introduction of the `co_double` types, but otherwise he found the code readable and comprehensible. A concern over such types, beyond the question of readability and recognition, centres on their potential to affect performance. To access an array on another image the code must first access the coarray to find the (allocated) address of the array component, then use that for the remote access, resulting in a doubling of accesses. However, there is scope for compiler optimization here since the compiler is allowed to assume that it may access cached memory between synchronizations, and only at synchronization points will the complete update of data be performed.

5.2 Numerical Experiments

For the experiments on performance we used a small problem of a turbulent channel flow on a mesh of 120 cubed points. Although our experiments show that the time spent in communication is small compared with the calculation time, the relative time is still smaller for larger problem sizes. We ran the code for 100 time steps so that the overall time is dominated by the time spent in computation and in halo exchange, approximately 99%. In production, this code would be run for even longer and the computation and halo exchange would be an even more dominant part of the code performance.

The experiments were run on a small Cray X1E with the code compiled in SSP-mode. We were able to use a maximum of 64 processors/images but these clearly indicated the trends.

In Figure 1 we show the speedup relative to the speed attained by the MPI code on one processor. The coarray code running on a single image runs 0.3% faster, a figure which is well within the variability of different runs, but which suggests that the overhead of indirection through the use of allocatable components is negligible. The same data is presented numerically in Table 1.

We see that both versions scale almost linearly up to four processors where communication is via the shared memory of an MSP module, but scaling is reduced as communication goes off-module. However, it is reduced by the same amount in both cases, the speeds are almost identical. Coarray Fortran is marginally slower, though again the difference is within the bounds of variability. We see a major change in scaling in the move from 4 to eight processors. Finer grained timing experiments suggest that this is neither a communication effect nor caused by load imbalance. The vast majority of the time is spent in local computation, and it is this which does not scale. We attribute this to the memory contention which can occur in memory intensive applications on the X1E when both processors in a socket are active .

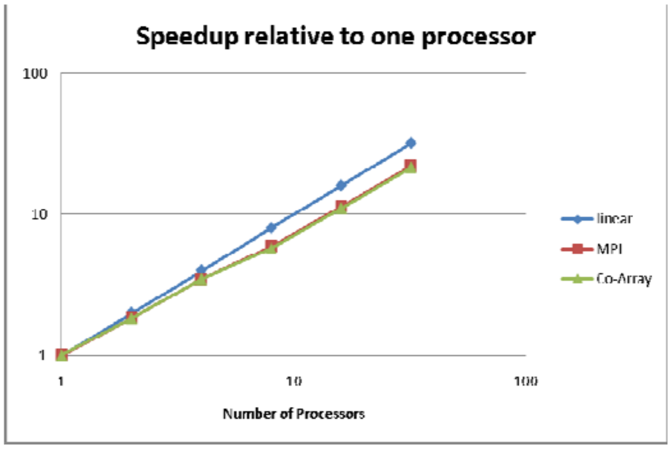

Figure 1: Speedup(log-log scale) for the 120 cubed problem relative to the MPI version on one processor

No images	Time MPI	Time CAF	Speed MPI	Speed CAF	Speedup MPI	Speedup CAF
1	2289	2282	1.00	1.00	-	-
2	1239	1252	1.85	1.83	1.85	1.83
4	656	657	3.49	3.48	1.89	1.91
8	384	400	3.96	5.72	1.71	1.64
16	201	207	11.39	11.06	1.91	1.93
32	104	106	22.01	21.29	1.93	1.95

Table 1: Times and speeds for the 120 cubed problem. In this table speedup is the speed relative to the speed on half the number of images.

This is the major conclusion to be drawn from these experiments: that Coarray Fortran offers very similar performance to well-wrought MPI code with less programmer effort, both in developing code and maintaining it.

We now consider what scope there is within coarrays for optimization. The MPI standard is large, powerful and contains many ways of effecting communication which a programmer can use to optimize a code. In contrast coarrays are deliberately simple and minimal, and for the most part the programmer is reliant upon the efficiency of the implementation. However, there are a few things a programmer can do to help the compiler. Just as in serial Fortran, the order of memory accesses can have a large impact on the performance of a code. This usually translates to nesting do loops in the correct order to reduce large strides through memory, but can also be affected by the order of allocating memory.

With the distribution of partitioned data we saw that there were two options: "push" and "pull". These differ in which side of an assignment has the remote coarray reference. In "push" mode, local data is stored into remote memory:

```
    a[k] = a
```

while in "pull" mode the remote data is read and stored locally:

```
    a = a[k]
```

We have separately timed the routine which distributes the partitioned mesh in SBLI for a mesh of 240 cubed points. Timings are given for both coarray modes, and for the original MPI routine. The memory requirements of the rest of the code meant that the MPI version did not run on 8 processors or fewer.

Number of images	Push (s)	Pull (s)	MPI
8	2.289	1.492	-
16	2.154	1.406	2.646
32	1.427	0.593	1.994
64	1.018	0.644	2.079

Table 2: Times to distribute the 240 cubed mesh

Coarrays are more efficient in both modes than MPI. Within coarrays, we find that pulling data is more efficient. For SBLI the difference is small in absolute terms, though significant relatively. For a code which relies heavily on gathering and scattering data, for example some electronic structure codes, there could be a large benefit from optimizing in this way, and in using coarrays generally.

6. Conclusion

We have successfully migrated a significant CFD program from MPI to use coarrays. The main advantage of using coarrays is that the code is more readable and understandable, and thus more maintainable. We have demonstrated this by various examples drawn from the program in question. Coarrays build on an existing language base in a consistent fashion and thus the learning time to become proficient can be small even for a programmer with little prior experience of parallel program development.

The performance of a code in production runs is of high importance, and our tests show that the performance of our code does not suffer from the change to coarrays. We have noted that there is scope for programmers to optimize code by skilful use of addressing, both within and across images. However, the simplicity of the coarray syntax allows many optimizations to be done automatically by the compiler.

With the forthcoming Fortran 2008 standard, coarrays represent the first International Standard parallel programming language, and our experience suggests that their use will enhance productivity and further the exploitation of parallelism.

Acknowledgments

We would like to thank Bill Long of Cray Research for useful discussions and for arranging access to their X1 and X1E systems, and Mike Ashworth and Roderick Johnstone of STFC Daresbury Laboratory for access to the SBLI code.

References

[1] HPF Home Page http://dacnet.rice.edu/Depts/CRPC/HPFF/index.cfm.
[2] OpenMP http://www.openmp.org/.
[3] MPI http://www-unix.mcs.anl.gov/mpi/.
[4] PVM http://www.csm.ornl.gov/pvm/pvm_home.html.
[5] SHMEM http://www.npaci.edu/T3E/shmem.html.
[6] Numrich, R.W. and Reid,J.K. (1998), Co-Array Fortran for Parallel Programming, ACM Fortran Forum,

17, pp 1-3. Available online as
ftp://ftp.numerical.rl.ac.uk/pub/reports/nrRAL98060.pdf.
[7] Chauvin S. et al, UPC Manual (The George Washington University, available at http://www.gwu.edu/~upc/documentation.html. See also Berkeley UPC – Unified Parallel C http://upc.lbl.gov/.
[8] Hilfinger P. et al, Titanium Language Reference Manual, University of California, Berkeley Report No UCB//CSD-04-1163x (2004). Available online at http://titanium.cs.berkeley.edu/papers.html.
[9] Ashby J.V(2005). Novel Parallel Languages for Scientific Computing – a comparison of Co-Array Fortran, Unified Parallel C and Titanium, Rutherford Appleton Laboratory Technical report RAL-TR 2005-015 http://epubs.cclrc.ac.uk/work-details?w=35072
[10] http://chapel.cs.washington.edu/ Chapel – the Cascade High Productivity Language. This page contains links to many reports on Chapel.
[11] http://domino.research.ibm.com/comm/research_projects.nsf/pages/x10.index.html The X10 Programming Language. This page contains a link to a presentation on X10 by V. Sarkar.
[12] http://fortress.sunsource.net/ Fortress Project home. This page contains links to many reports on Fortress.
[13] Ashby J.V(2007). New Languages for High Performance, High Productivity Computing, Rutherford Appleton Laboratory Technical report RAL-TR-2007-012 http://epubs.cclrc.ac.uk/work-details?w=40805
[14] Reid, John (2008). Coarrays in the next Fortran Standard. ISO/IEC JTC1/SC22/WG5 N1724. See ftp://ftp.nag.co.uk/sc22wg5/N1701-N1750/N1724.pdf
[15] ISO/IEC (2008). CD revision of the Fortran Standard. ISO/IEC JTC1/SC22/WG5 N1723. See ftp://ftp.nag.co.uk/sc22wg5/N1701-N1750/N1723.pdf
[16] M. Ashworth, PDNS3D - Benchmarking a Direct Numerical Simulation Code http://www.cse.scitech.ac.uk/arc/pdns3d.shtml