Technical Report

RAL-TR-2005-015

8 December 2005

Novel Parallel Languages for Scientific Computing – a comparison of Co-Array Fortran, Unified Parallel C and Titanium

J.V.Ashby
Computational Science and Engineering Department
CCLRC Rutherford Appleton Laboratory
J.V.Ashby@rl.ac.uk

Abstract

With the increased use of parallel computers programming languages need to evolve to support parallelism. Three languages built on existing popular languages, are considered: Co-array Fortran, Universal Parallel C and Titanium. The features of each language which support paralleism are explained, a simple example is programmed and the current status and availability of the languages is reviewed. Finally we also review work on the performance of algorithms programmed in these languages. In most cases they perform favourably with the same algorithms programmed using message passing due to the much lower latency in their communication model.

1. Introduction

As the use of parallel computers becomes increasingly prevalent there is an increased need for languages and programming methodologies which support parallelism. Ideally one would like an automatic parallelising compiler which could analyse a program written sequentially, identify opportunities for parallelisation and implement them without user knowledge or intervention. Attempts in the past to produce such compilers have largely failed (though automatic vectorising compilers, which may be regarded as one specific form of parallelism, have been successful, and the techniques are now a standard part of most optimising compilers for pipelined processors). As a result, interest in tools to support parallel programming has largely concentrated on two main areas: directive driven additions to existing languages and data-passing libraries. The directive driven approach, typified by HPF (High Performance Fortran) [1] and OpenMP [2] allows a programmer to identify parts of the code which can be executed in parallel or where data can be distributed, and even to give hints or instructions to a compiler on how best to achieve this. With data-passing libraries (which include message-passing libraries such as MPI [3] and PVM [4], but which also includes support for the distributed shared memory model such as SHMEM [5]) the entire onus of producing a parallel algorithm is passed to the programmer, the library merely facilitating such communication between processes (data-transfer, synchronisation, etc.) as might be necessary. The result in both cases is that parallel programming has come to be seen as arcane, difficult and less efficient than it has often promised. Countering this is the understanding that parallel computation provides the only way to tackle many of the more interesting problems in science and data processing, and thus the difficulties are worth overcoming.

In recent years there has been a move to return to the problem from a third angle, to design a language which allows the natural expression of parallel algorithms. This report looks at three such languages, Co-Array Fortran, Universal Parallel C and Titanium, and assesses their usefulness for scientific computation. We first review the essential characteristics of a parallel scientific program and attempt to set out the requirements a good parallel language should satisfy. Then we consider each of the three languages in turn, looking at their suitability, ease of use and where possible their performance.

2. Characteristics of a Parallel Program

A serial computer program consists of a set of instructions executed sequentially which transform input data to output data, in most cases 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 independent to a great extent; an example is process farming where some aspect of a parameter space is explored by multiple instances of the program running on slightly different initial data to find some optimum output. In other cases each program needs knowledge of the calculations being carried out by some or all of the other programs and the programs need to exchange data with each other. 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.

It should be noted that in the preceding paragraph we have used the term program in a broader sense

than is customary. Usually we think of a program as the entire set of statements executed between one user interaction with the operating system (through a shell or similar mechanism) and another. Since the operating system and the shell are themselves sets of instructions with input and output data, they are also programs by our definition, and the user programs (in the conventional sense) can be seen as nested within them. Similarly a user program can contain programs in our sense, the most obvious example in a serial code being a subroutine or procedure. A user program can consist, for example, of a serial part which reads in data and prepares it for processing, this then spawns multiple instances of a parallel part to perform some calculation, the results of which are gathered for a final serial reduction to the required output. The parallel parts may be instantiated as threads in a multi-threading shared memory system, or as independent processes on separate processors in a message-passing environment, and it is to avoid seeming as though this discussion applies only to one or other of these that we avoid the terms threads or processes and use the more generic program.

Each program in a parallel set may need data produced by another member. To access that data it must know a) which other program to ask for it (though in a shared memory system this is less of an issue) and b) how that program refers to it. In addition, the other program may need to know that it should expect to be asked for the data. Each of the languages we discuss in this report take the approach of requiring a uniform data representation across programs. Thus if Program A needs to know some entity which it represents as an array X from Program B, B will also have an array X which refers to the same entity.

3. Programmer Requirements

Scientific computation is largely carried out by people who are scientists first and programmers or computer scientists second. This gives a slant on their requirements for parallel programming support which might not pertain among pure computer scientists. This should not be taken to imply that either is any better than the other, merely that different groups have different approaches and needs, which are best served by a multiplicity of solutions. Below we try to give programmer requirements for a parallel programming language which are generic and could apply equally to both sets. Subsequently we shall discuss the specifics of these requirements for scientific computing.

Any computer language stands or falls primarily on its modelling ability, the capacity of the language to represent the objects or entities the programmer wishes to manipulate in as natural a way as possible. Again we reiterate that what is natural, what “feels right”, may vary from group to group. For example, mathematicians or physicists may find it hard to think in terms of pointers to memory rather than arrays, while computer scientists may find subscripted arrays restricting.

The language should be sufficiently expressive for its intended domain of application. On the other hand, it should have a syntax which is compact but comprehensible and not contain too many features that will rarely or never be needed. A common concern about languages such as Fortran95 and C++ is that they are too large, that they have become bloated with extra features (though some of this can be explained by simple resistance to change). On the other hand, while it would be possible to perform scientific calculations in a language that only supported integer arithmetic (by mimicking floating point) it would clearly be cumbersome to do so. Equally, in the context of parallel programming, a language should support multiple methods of data distribution, that is to say, a programmer should have as much control over how data is distributed as possible.

Many scientists have spent a lot of time and effort establishing their existing practices, and so any new development will stand a greater chance of acceptance if it is a small perturbation from what

they currently do. Thus a small addition to an existing language (which all three of the languages we consider here are) is much more likely to succeed than a completely fresh language designed from scratch, however attractive it might be theoretically.

Two issues which also play a large part in decisions about which language to use are portability and software re-use. Portability is best served through the use of internationally agreed standards (or, *faute de mieux*, de-facto standards). Software re-use implies that languages should be capable of building generic libraries; it would be foolish if, for example, every global summation had to be hand-coded.

4. Co-Array Fortran

Co-Array Fortran, (CAF for short) was originally known as F-- (eff minus minus), a name meant to imply a small addition to the Fortran language. This was also a jokey reference to C++ which was seen to be overly complex. Its design was driven by the question "What is the smallest change needed to convert Fortran95 into a robust, efficient parallel language?" 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, CAF has been developed into a proposed section of the new Fortran standard currently under discussion for 2008 [6]. This differs in several respects from the original language, but since the new standard is not yet in place and there are existing compilers for the old language, we shall discuss both languages here.

Numrich and Reid [6] distinguish between work distribution and data distribution. For the former CAF uses the SPMD model discussed above. In CAF terms each replication of the program is called an image and (apart from synchronisations) each image can be considered as an independent program. Data is distributed by the use of Co-Arrays. Each image has its own memory with data declared in it, but it also has access, through the co-array syntax, to the data in other images. The co-array indices which address the images in similar fashion to the way ordinary array indices address elements in memory, are placed after the normal indices in square brackets. Thus `A(5,10)[6]` refers to the (5,10) element of the array A on image 6. `A(5,10)` refers to the same element on the local image. Since references across images imply communication, the presence of numerous square brackets acts as a visual signal to the programmer that too much communication may be taking place and that this part of the code may therefore create a bottleneck. Arrays may be allocated at different (local) sizes on different images by using a derived type with an allocatable (or pointer) component. In this case (as with all derived types) the co-array index is tied to the base array. If `B` is a derived type with a pointer component, `comp`, then to access the values associated with that component on image `n` CAF uses `B[n]%comp(:)` rather than `B%comp(:)[n]`.

The co-array index acts in the same way as a normal array index, and can be of up to rank 7, as can the local rank. In the proposed 2008 standard this is relaxed to the requirement that the sum of rank and co-rank be limited to 15. Co-array indices are mapped to images in the usual Fortran "first runs fastest" manner and the final index will usually be left to run free. The effect is to give each co-array a hyper-cuboidal arrangement of images. These need not be the same for all co-arrays, either in bounds or in dimensionality. Some care must be exercised to ensure that a co-array is not addressed beyond the number of images available. The exact process by which the number of images is set is not determined by CAF, but is left as an implementation detail. A compiler is free to provide an additional subroutine to set the number of tasks, or to set it from the command line arguments or through a launcher program akin to `mpirun`. A program can interrogate how many images are running through the `num_images()` function, and an image can find out its own designation using the function `this_image()`. This designation will vary for different variables

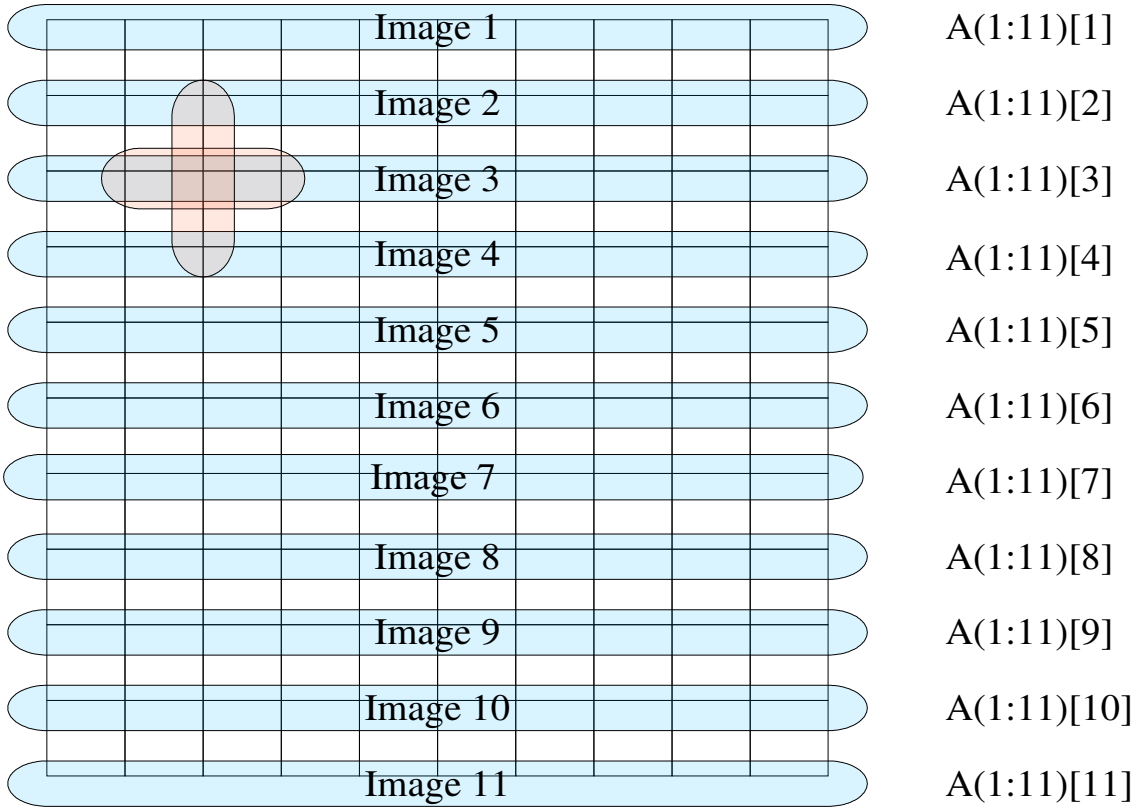

Figure 1 A finite difference grid distributed slice-wise between images

if they are distributed differently. For instance if we have two co-arrays declared as `A(:)[4,*]` and `B[*]`, then if image 13 invokes `this_image(A)` the returned value will be the integer array /1,4/, while `this_image(B)` will return the integer scalar 13.

Synchronisation calls ensure that when one image asks for data from another, that data is ready to be given. Some language statements carry implicit synchronisation, as explained further by Numrich and Reid [6], as a safeguard that the language will not produce errors. For example allocation of a co-array is synchronised so that it will not be possible for an image to reference the co-array on another image before it has been allocated there.

In other cases the programmer must be responsible for synchronisation as part of the underlying algorithm. Three intrinsic subroutines are defined by the language:

- `sync_file(unit)` synchronises I/O on a file identified by the unit number, `unit`. If the unit does not exist or is not connected to the invoking image it has no effect. If it is connected, then I/O operations (including replication of the file for other images to access) is completed.
- `sync_team(team[,wait])` is the routine that will most flexibly synchronise images. The `team` argument is an integer scalar or rank-one array containing a list of images with no repeated values. It specifies a team of images which must synchronise. The `wait` argument specifies a list of images which must synchronise before execution can continue.
- `sync_all([wait])` operates as a special case of synch_team where the team argument is the entire set of images.

The final language elements control the execution of critical sections. In essence a critical section is a portion of the code which should only be executed by one image at a time. Such sections are enclosed in calls to `start_critical()` and `end_critical()` subroutines. In the proposed standard this is changed to introduce new statements, `critical` and `end critical` (c.f. `do` and `end do`, `if` and `end if`) which delimit a critical section.

4.1 A simple example

One of the simplest examples, due to Numrich and Reid [6], is the evaluation of a two-dimensional Lapacian operator using a five-point finite difference approximation. A co-array, `A`, is defined such that the values for a given y are all held in one image and neighbouring images have neighbouring y-values. This is demonstrated in Figure 1. Also shown is the five point star where it can be seen that there are contributions from within an image and from the two neighbouring images.

To program this in a subroutine whose arguments are the extent in the x-direction, `nrow`, the extent in the y-direction, `ncol` (less than or equal to the number of images), and the co-array, `A`, we need also to define a local array into which the data can be gathered. This avoids overwriting values which may be needed by other images. Thus the routine starts:

```
Subroutine laplace(nrow, ncol, A)
     integer, intent(in) :: nrow, ncol
     integer, intent (inout) :: A(ncol)[*]
     real :: new_A(ncol)
     integer :: i, me, up, down
```

`new_A` is an automatic array which will temporarily hold the updated solution until all communications have been performed. It is a local array, that is, one copy of it exists in each image, and is not available to other processors. The integers `me`, `up` and `down` will refer to the current and adjacent images. We apply periodic boundary conditions. First we make the calculation in the x-direction:

```
     new_A(1) = A(ncol) + A(2)
     new_A(ncol) = A(ncol-1) + a(1)
     new_A(2:ncol-1) = A(1:ncol-2) + A(3:ncol)
```

Now determine which image we are in and calculate the neighbouring images, again applying periodic boundary conditions:

```
     me = this_image(A)
     up = me + 1; if (me == nrow) up = 1
     down = me – 1; if (me = 1) down = nrow
```

The next few statements are where the communication takes place. First we synchronise with the neighbouring images to make sure the correct data is available to us, add it to the local storage, then synchronise again before overwriting it in case other images have not yet finished with the data:

```
     call sync_all(/up, down/)
     new_A(1:ncol) = new_A(1:ncol) +A(1:ncol)[up] + A(1:ncol)[down]
     call sync_all(/up, down/)
```

Finally we update the distributed data and return.

```
     A(1:ncol) = new_A(1:ncol) – 4.0 * A(1:nrow)
end subroutine laplace
```

Although this is a very simple example, it embodies the qualities CAF seeks to provide: The code looks like Fortran95 in all but a few respects, and thus the algorithm is expressed in familiar (to Fortran programmers) terms. There is no complex procedure for initialising the parallel communications. Communication is clearly signalled by the presence of square brackets and is localised to a small region of the code. The areas of code which have no communications are amenable to the usual set of optimisations available to serial compilers, but in the absence of synchronisation there are additional optimisations available to the compiler such as the re-use of a non-local reference from cache to reduce the communication overhead.

What drawbacks may there be to CAF? Its very simplicity may tell against it – there are algorithmic elements it might find difficult to express, although several of these are addressed by Numrich and Reid and proposed extensions to the language exist, such as the passing of co-array sub objects as arguments to subroutines.

The model of data distribution, as hyper-cuboidal, may be restrictive for some applications. An example of this could be in unstructured mesh partitioning where the graph representing the connectivity of partitions does not have the regularity that a hyper-cuboidal distribution would imply. To overcome this it is possible to have a co-array of a derived type, one of whose components is allocatable. This component can then be allocated at different sizes on different images. The co-indices, however, remain hyper-cuboidal.

The proposed 2008 standard will also introduce collective intrinsic functions which take a co-array argument and return a result of the same shape on each image of a team. The functions are:

`co_all(mask[,team]`	True if all values are true
`co_any(mask[,team])`	True if any value is true
`co_count(mask[,team,kind])`	Number of true values
`co_maxloc(array,[team])`	Image indices of maximum values
`co_maxval(array[,team])`	Maximum values
`co_minloc(array,[team])`	Image indices of minimum values
`co_minval(array[,team])`	Minimum values
`co_product(array[,team])`	Products of elements
`co_sum(array[,team])`	Sums of elements

All collective functions carry implicit synchronisation of the team.

4.2 Status of Co-array Fortran.

Co-array Fortran was developed originally by Cray as a fall-back position for HPF. CRAFT and MPI, it then came to be seen as a model of the one-sided get/put as embodied in the SHMEM library on the Cray T3D/E and the Cray (later SGI) Origin 2000. CAF has been implemented in the Cray Fortran95 compiler [7]. To access it a special compiler flag, -Z, is provided. The number of images can be set at compile time if desired using another flag, -X, or at runtime by executing the program with the mpprun command. Tools such as the debugger and performance analyser treat co-arrays as multiple local instances, one per image. Thus the data is accessible by switching between images, but the co-array subscripts are not. Particularly in the case of complex multi-co-dimensional co-arrays, this could create problems.

There is also a translator from CAF to OpenMP [8]. This only implements a (significant) subset of the language, but most programs require very little adjustment to bring them into line with this subset. The subset has been chosen to make it possible to implement co-arrays as ordinary arrays of

higher rank, ideal in a shared memory environment. Once the code has been translated, an OpenMP capable compiler is needed, but these are widely available for a range of platforms from small workstations through commodity clusters to supercomputers.

The group at Rice University have produced a portable CAF compiler [9]. It is actually a source translator from CAF to Fortran90 with a library of standard message passing calls. This library must be tailored to the target platform, so the system is currently only available for RedHat Linux, OSF1 Tru64 on Alphaservers and SGI Origins running IRIX64.

The current Fortran standard is Fortran2003. Work is currently in progress on the next revison of the standard, due in 2008. As stated above, it is currently planned to incorporate Co-array Fortran in that standard [6]. In that event, Fortran2008 could be the first general purpose language standard to explicitly support parallel computing.

5. Universal Parallel C

Just as Co-array Fortran builds on the existing Fortran language, Universal Parallel C (UPC) [10] is an explicitly parallel extension to the ANSI C standard, and attempts to maintain the concepts and methodology of C programming. The language was designed following earlier experience with similar languages such as SplitC, AC and PCP.

UPC uses a distributed shared memory programming model which establishes a global address space for all the memory available to a program, whether distributed or shared, with additional constructs to use data locality. The global shared memory is divided into partitions, each with an affinity to a thread. Threads in UPC are akin to images in CAF. The memory associated with a thread is further divided into private and shared address spaces. Private memory holds the local variables and arrays of a thread, shared memory, as its name implies, holds data that may be accessed by other threads. Again, in CAF terms the shared memory holds the co-arrays.

Where a variable sits in this memory is determined by how it is declared. Distribution of data in memory is by (block) round-robin. If arrays are thought of as intrinsically one-dimensional, the default distribution of an array is to place the 0-th element on thread 0, the 1-st element on thread 1, up to the (N-1)-th element on thread N-1, then the N-th element is placed on thread 0 and the process begins again. If more flexibility is required, elements 0 to m-1 can be placed on thread 0, m to 2m-1 on thread 1, and so on. This loses the flexible referencing model of CAF with its 7 local and 7 co-dimensions, but is in keeping with traditional C practice. On the other hand, with CAF a co-array reference which resolves to an image number greater than the number of images running at the time will generate an error, in UPC this will not happen, though there may be implications for load balancing and efficiency of performance.

Data coherency and work distribution are managed by several approaches. The degree of memory consistency, strict or relaxed, can be specified for a program, a section of code or an individual variable. Locks can be applied to sections of code to prevent simultaneous execution by more than one thread, and barriers provide explicit synchronisation points.

Data is declared in the same way as in C, with the addition of the keyword `shared`. Thus an integer array which is to be distributed across threads is declared:

```
shared int arr[ARRLEN];
```

If the block size is to be different from 1 then the declaration becomes:

```
shared [ARRBLOCK] int arr[ARRLEN];
```

To distribute a two-dimensional array by rows, the declaration is:

```
shared [NCOL] int arr [NROW] [NCOL];
```

There is no equivalent `private` keyword, the omission of `shared` implies the data is private, just as the omission of co-dimensions in CAF implies data is local. Shared data must be global.

The round-robin distribution of data must be used with care, at its most naïve it would mean that data that is close together on a serial architecture may be spread across threads in parallel. However, if the programmer is careful about the choice of data distribution, this can be a powerful and expressive tool.

Since UPC is a dialect of C it supports pointers. There are four distinct possibilities for a pointer in UPC since they can be declared as either shared or private, and they can point to either shared or private memory. The private to private pointer (`int *p1`) is simply an ordinary C pointer in each thread. A private pointer pointing into the shared memory (`shared int *p2`) will exist on each thread and can provide speed and flexibility, a shared pointer into shared memory (`shared int *shared p3`) will only exist on one thread (usual thread 0) but be accessible from all other threads. Finally a shared pointer into private space (`int *shared p4`) will generate an access violation for any thread other than the thread to which that private space belongs and should be avoided. Because UPC pointers have to keep track of more than simply the memory location they point to, their representation can have an impact on the use of the code. For example, a typical representation needs to store the virtual address of the block containing the pointee, the thread to which it has affinity and the phase (position in the block). If the bit count for the thread section is low, then this will restrict the maximum number of threads a program can use. While one would hope that compiler writers will take such things into account when designing such data structures, it is as well for programmers to be aware of them. The example given in [10] has 10 bits for the thread identifier, which would give a maximum processor count of 1023.

UPC provides four special functions to access information about pointers. The function `upc_threadof(shared void *ptr)` returns the number of the thread with affinity to the shared object pointed to by ptr. This extends the functionality of CAF's this_image. `upc_phaseof` and `upc_addrfield` return the position in the block and the local address associated with a pointer respectively. There are also three enquiry functions which return various sizes associated with a pointer.

It is worth commenting that the presence of pointers in the language, as for conventional C, can hinder optimisation of the code and hence restrict performance.

Two special values are available as keywords: `THREADS` specifies the total number of threads running in the current execution. `MYTHREAD` returns the thread number being executed by a particular thread, it runs between `0` and `THREADS-1`.

Synchronisation is achieved in two parts in UPC. Firstly there is the notion of memory consistency, which can be strict or relaxed. In strict mode the program is run in a sequential consistency model; it appears to all threads that memory references within the same thread appear in the program order. In relaxed mode it is sufficient that all references to shared data within a thread appear in the program order to the issuing thread. The `upc_fence` function will locally synchronise the shared memory accesses. For global synchronisation, `upc_barrier` is provided for a blocking barrier and the pair of functions `upc_notify` and `upc_wait` will give a split-phase (non-blocking) barrier.

To create locks, portions of code which can only be executed by one thread at a time, is only a little more complicated. Such locks are used, for example,to ensure that shared data does not change during a calculation. Consider the line of code `a = a + b` where `a` is shared and `b` has been locally determined. The idea is to finish the calculation with all `b`'s summed in `a`. If two threads arrive close together in time it is possible for the second to move `a` into its registers before the first thread has updated it. Thread 1 will re-write its updated `a` and then thread 2 will write its own version, and the contribution from thread 1 will be lost. To overcome this problem, the updating code is locked and unlocked. A pointer is declared as:

```
upc_lock_t *lock;
```

When needed, it is allocated using:

```
lock = upc_all_lock_alloc()
```

This has the customary behaviour of malloc. When the segment of code to be locked is reached the program issues

```
upc_lock(lock);
```

and releases the segment again with:

```
upc_unlock(lock);
```

Finally when the lock is no longer required the memory can be released using:

```
upc_lock_free(lock).
```

The three UPC versions of `memcpy`: `upc_memcpy`, `upc_memput` and `upc_memget`, operate on shared-to-shared, private-to-shared and shared-to-private transactions respectively.

Dynamic memory allocation is handled in a special way with four allocation functions. `upc_all_alloc` and `upc_global_alloc` are global in that they allocate shared space across threads. The former is a collective function returning the same pointer value on all threads (it thus follows that it must be called with the same arguments). The latter is not collective – if called by more than one thread a region of shared memory is allocated for each thread. `upc_local_alloc` and `upc_alloc` are local, they allocate shared memory with affinity to the calling thread only. `upc_local_alloc` was in the original specification of UPC, but is now deprecated in favour of `upc_alloc`.

Finally, UPC introduces one control construct into the language. `upc_forall`. This operates in a similar fashion to the standard C `for` loop, except that each loop execution must be independent. It takes an extra parameter, the affinity of the loop, thus determining which thread should run the current loop. This parameter is either an integer (which is reduced to a thread identifier by modular division by `THREADS`) or an address, the thread to which that address has affinity defining the affinity of the loop. By using a variable to define the affinity, the threads so created will execute in parallel.

5.1 An example

We shall use the same example as for CAF to allow comparison. This example does not fully explore the richness of the UPC language, but it will serve to show how the language can be used. Clearly the first task is to define and distribute the shared data in A.

```
#include <upc_relaxed.h>
#include <stdio.h>
#define NX 11
#define NY 11
shared [NX] float A[NY][NX];
```

A is declared as a two-dimensional array, distributed so that A[0][0] to A[0][NX-1] are on thread 0, A[1][0] to A[1][NX-1] are on thread 1, and so on. Now comes the computation of the Laplacian:

```
void laplace (int nx, int ny, shared [nx] float A[ny][nx]){
     int i, up, down;
     float *new_A;
     new_A= upc_alloc(nx);
```

A different new_A is allocated on each thread as a private array. We can now do the local part of the calculation easily

```
     new_A[0] = A[MYTHREAD][1] + A[MYTHREAD][nx-1];
     new_A[nx-1] = A[MYTHREAD][0] + A[MYTHREAD][nx-2];
     for (i=1; i<nx-1; i++){
          new_A[i] = A[MYTHREAD][i-1] + A[MYTHREAD][i+1]}
          }
```

Now we perform the part of the calculation which requires communication. First as in CAF we place a synchronisation barrier at the start:

```
     upc_barrier;
     upc_forall (j=0; j<ny; j++; j){
          up = j + 1; if (up == ny) up = 0;
          down = j-1; if (down == -1) down = ny-1;
          for (i=0; i<nx; i++){
               new_A[i] = new_A[i] + A[up][i] + A[down][i]
               }
          }
     upc_barrier
     for (i=0; i<nx; i++){
          A[MYTHREAD][i] = new_A[i] – 4.0*A[MYTHREAD][i]
          }
     }
```

Another synchronisation barrier has been placed at the end of the upc_forall loop and before the updating loop.

The expression of the algorithm is very similar to the CAF version, the main added complexity is caused by C's lack of array syntax which has greatly compacted the array assignments in CAF.

5.2 Status of UPC

UPC evolved from earlier projects such as Split-C and AC. It has achieved a higher degree of uptake than those and has been adopted by several influential manufacturers. Cray have incorporated it into their compiler for X1 and future systems [11], and support for earlier Cray systems such as the T3E is available from gcc [12]. gcc will also support UPC on x86 architectures and SGI Irix machines. Hewlett Packard also have a compiler which was the first commercial UPC

compiler running on Tru64 and HPUX systems [13]. There is a suite of tools including compilers and MPI-based runtime libraries for linux and Tru64 from Michigan Technological university [14].

Finally the Berkeley UPC compiler [15] supports a wide range of systems and hardware including SPARC/Solaris, PowerPC/MacOSX, and many others. It does this by translating to C and can be used with many C compilers, native and commercial.

6. Titanium

The third language we consider is a dialect of Java called Titanium [16]. It shares many of the features and philosophies of Co-Array Fortran and Universal Parallel C, and indeed, several of its developers were also involved in UPC.

Once again the execution model is SPMD and the data model contains both local and globally addressable memory. Titanium refers to the different instances of a program as processes. In Titanium the default declaration, unlike in CAF and UPC, is for data to be global (shared). The `local` qualifier overrides this. In addition data can be declared as `nonshared` (similar to local but can be applied more widely) or `polyshared` which may be shared or non-shared. Since there is a large overhead with using global pointers to reference local quantities on a distributed memory machine these should be avoided where possible.

Titanium supports three communication paradigms: broadcast (one-to-all), exchange (all-to-all) and one-to-one. The first two have elemental statements in the language; thus

```
broadcast E from p
```

evaluates the expression `p` which must be an integer within the range of the number of processes, then evaluates `E` on process `p` and communicates the value to all other processes. For this to work correctly, of course, `p` must be the same on all processes. This is achieved through the concept of the single-valued variable, a variable which is required to be the same for all processes.

Exchange (all-to-all communication) is performed by a method defined on a variable. A simple piece of code using exchange is:

```
int [1d] single allData;
allData = new int [0:Ti.numProcs()-1];
allData.exchange(Ti.thisProc()*2);
```

Here an array of integers is declared and created. Its length on creation is the number of processes, accessed through the `Ti.numProcs()` function. By being declared single it is guaranteed to be the same on all processes. The exchange then causes each process to insert twice its process number (accessed by `Ti.thisProc()`) in its place in the array. Normally it is references to an object that are passed by exchange, rather than primitive values.

One-to-one communication is done implicitly through the global address space. Since all data is accessed by reference, we set up the references as for the exchange. Then the 0-th element of `allData` refers to process 0, and so on. To access a variable on a particular process, we merely need to reference it through the appropriate element of `allData`. Thus we could increment the data on process 3 by 25 by the statement:

```
allData[3].val += 25;
```

The data initialisation above creates a single variable on each process. To create two-dimensional arrays on each process we would declare

```
int [1d]single [2d] allData = new int [0:Ti.numProcs()-1][2d];
```

which creates a vector of pointers, one for each process, which point to two dimensional arrays, again one on each process.

Synchronisation is achieved by calls to `Ti.barrier()`. There is the facility within the language (not yet implemented) to partition the processes into teams. In that case a call to `Ti.barrier` synchronises the team to which the calling process belongs. One difference between Titanium and the other languages is that each process must execute *exactly the same* barrier call before they can proceed. This simplifies the compiler analysis, but it does mean that some innocent looking programs are illegal. This restriction also applies to implicit synchronisation such as broadcasts and reductions.

To ensure the consistency of shared data, Titanium defines the order in which memory operations issued by one processor are seen by other processors to take effect. This follows closely the model of memory consistency defined in ordinary Java for thread-based memory events. They are expressed through formal rules but in essence say that a) dependencies in the program will be followed, b) reads from memory and writes to memory performed in a critical demesne (a region of memory attached to a process) take place in that demesne, c) synchronisation events must not be re-ordered, d) the usual semantics of memory and synchronisation constructs apply and operations on volatile memory must execute in order.

One of the things which makes understanding the Titanium documentation more difficult than than for the other languages under consideration in this report is that Titanium has taken it upon itself not only to provide support for parallel computation, but also to make good perceived deficiencies in Java as a language for scientific computation. Java has no real support for multi-dimensional arrays. Titanium overcomes this, but compared with the simplicity of both C and Fortran, declaring a two-dimensional array is cumbersome:

```
Point <2> l = [1,1];
Point <2> u = [15,35];
RectDomain <2> r = [l : u];
double [2d] A = new double[r]
```

as opposed to:

```
double precision, dimension=(1:35,1:15) :: A
```

or

```
double A[13][34]
```

Two Points are defined at the bottom-left and top right of the array, these are used to define a rectangular domain which is then used to create the array A. As well as RectDomains, Titanium has general Domains formed by the union of two or more RectDomains.

Titanium only introduces one new control construct into Java, and this deals more with the support for multi-dimensional arrays than with parallel processing as such. The foreach statement iterates over a Domain, but does not guarantee an order of execution, thus enabling the compiler to make aggressive optimisations while guaranteeing that array bounds are not breached.

6.1 An Example

We start as before by defining the distributed data. Since the default is for data to be shared this is straightforward:

```
double [1d][1d] A = new double [0:Ti.numProcs()-1][0:nx-1];
```

In contrast, the local variables and arrays must be explicitly declared as such:

```
int local i, up, down;
double local [1d] new_A = new double [0:nx-1];
```

The local part of the calculation proceeds as before:

```
new_A[0] = A[Ti.thisProc()][1] + A[Ti.thisProc()][nx-1];
new_A[nx-1] = A[Ti.thisProc()][0] + A[Ti,thisProc()][nx-2];
for (int i = 1; i < nx-1; i++) {
     new_A[i] = A[Ti.thisProc()][i-1] + A[Ti.thisProc()][i+1]
}
```

Then for the part of the calculation which requires communication we start (and end) with a synchronisation call:

```
Ti.barrier();
up = Ti.thisproc+1
down = ti.thisProc()-1;
if (up==Ti,numProcs()) up = 0
if(down==-1) down = Ti.numProcs()-1
for (int i = 0; i < nx; i++) {
     new_A[i] = new_A[i] + A[up][i] + A[down][i]
}
Ti.barrier();
for (int i = 0; i <nx; i++) {
     A[Ti.thisProc()][i] = new_A[i] – 4.0 * A[Ti.thisProc()][i]
}
```

Again this example does not do justice to the richness of the Titanium language and does not seek to use the Object Oriented features that basing Titanium on Java makes available. It does, however, serve to illustrate the ease with which a naive programmer can make use of the language.

6.2 Current status of Titanium

Titanium, while it is based on the Java language, is not a true extension of Java,and there are several Java constructs, most notably threads, that will not work under it [16]. In part this is because rather than acting as a Java compiler, the Berkeley `tc` compiler actually translates to C (currently C++). The advantage of this, of course, is that it makes the language quite portable. Installations of `tc` exist and are active on many systems, including Linux, AIX, IRIX, Solaris and Microsoft Windows [17]. Obviously, much depends on the quality of the runtime layers on these various systems, and a great deal of work has clearly gone into creating a system which will port readily.

This reliance on another language is a drawback, since one of the major advantages of a true Java-based parallel language would be,as with Java itself, its use of the Java Virtual Machine to remove portability problems and make heterogeneous computing entirely transparent. Bull and Telford [18] have surveyed investigations into more “pure” parallel Java. They conclude that work in this area

needs to progress through standardisation and experience with message passing systems before Java is sufficiently mature to consider data-parallel extensions, though this report was published several years ago.

7. Perfomance measurements

All the languages described here have been implemented and the implementations tested for performance. As is often the case, the fundamental question that must be answered is “With what should the implementation be compared?” There are two common approaches, either to compare a code written in the parallel language with the same algorithm coded in the parent language and using another approach to provide parallelism (usually MPI), or to compare the same algorithm expressed in another language, either an intrinsically parallel language or via MPI. Both approaches are fraught with difficulties of interpretation as they run the risk of measuring compiler performance rather than language performance. Experiments comparing the serial languages Fortran, C and Java [19] have shown that on the same platform there can be as great a variability in performance from compilers of the same language as from different languages.

The Rice University group [20] have analysed the performance of their CAF compiler on the kernels from NAS parallel benchmark set [21]. They compare with hand-coded Fortran95 MPI versions. In most cases the MPI gives slightly better figures for speed-up as the number of processors increases. Unfortunately they do not give raw speeds, only speed-up relative to a single processor, so these results are inconclusive as to the absolute effectiveness of CAF.

Wallcraft [22] has compared implementations of ocean models using MPI, OpenMP CAF and SHMEM on an SGI Origin. He uses the caf2omp translator and finds that it performs as well as hand-coded OpenMP. Both perform close to the ideal, maintaining their speed as the number of processors increases, in contrast to MPI which falls away dramatically.

El Ghazawi and Cantonnet [23] have measured performance of the implementation of UPC for a Compaq Alphaserver on a variety of benchmarks. They find that the UPC code does not run as fast as MPI versions, nor is its scaling as good, but ascribe this to the immaturity of the compiler and express confidence that improved compiler optimisations will result in competitive performance. The same authors, together with Yao and Vetter [24], have analysed the performance of UPC on a Cray X1 using the STREAM [25] and other benchmarks. They comment that the ultimate performance is limited by the quality of the underlying C compiler, but that UPC can provide high performance.

Dawson [26] and Johnson [27] have respectively discussed the performance of CAF and UPC on a Cray X1. They both find that the speed-up factor is much closer to the ideal as the number of processors increases than it is with MPI code. They attribute this to the much lower latency in the one-sided communication of the two language.

Datta, Bonachea and Yelick [28] have used the same NAS benchmarks as in [19] to look at the performance of Titanium in comparison to Fortran/MPI implementations. They find that Titanium's performance is generally lower by up to 25% than Fortran's. It is unclear from their data how much of this reduction is due to the C++ compiler used and how much is inherent in the Titanium language itself.

8. Conclusions

The three languages studied in this report all take a broadly similar approach to the problem of designing a language for general parallel programming. All three adopt an SPMD model with global memory addressing, though they use slightly different methods of achieving this. While the memory is viewed as globally accessible, areas of memory are tied to processes (images in CAF, threads in UPC) which are deemed to own them, but to make them available to other processes on demand. The way in which global data is accessed is, in general, through an extension of the base language's method for accessing local data. Thus CAF adds the co-array index to extend Fortran's ordinary arrays, while UPC introduces the shared keyword and relies on the programmer to keep track of how arrays are spread across processors, in keeping with C's underlying data model which is closer to the addressing of physical memory than to the abstractions of Fortran datatypes. Titanium uses the referencing mechanism of Java and a set of pointers into the partitioned memory.

All three require the programmer to make explicit provision for synchronisation and locking memory, and this, perhaps, is the point at which the programmer is forced to confront the low-level computer implementation of the program. All programming languages deal to some extent or another in abstractions, some closer to the physical hardware than others. In an ideal language these abstractions would be cleanly separated from their physical manifestation, and the programmer could think in abstract alone. In reality the effects of the physical, the finite time taken to move data, the possibility of two processes updating the same variable, and so on, must all be taken into account when writing a program.

The programming model is actually more than SPMD, it is implicitly homogeneous SPMD in that there does not appear to be provision in these languages for a parallel program working on a heterogeneous cluster, a *modus operandi* which is supported, for example, by MPI. It is possible, however, that future implementations of the languages, particularly their run-time support, may overcome this, but the one-sided communication model used will make this complicated.

As suggested in the section on Programmer Requirements, each of these languages has something to offer a programmer: some will find it easier to program in Co-Array Fortran, others in Unified Parallel C and others still will relate best to the Object Oriented approach of Titanium. This is a healthy state of affairs, one that should be encouraged, and it is certainly not the purpose of this report to suggest that one language should be preferred over another. Software developers should be able to choose the best language in which to express their algorithm. But in order for this to be possible, it is necessary that compilers for these languages exist and are widely available. To date CAF is easily available only on some Cray architectures. The Rice compiler [9] is still in development, and is unlikely to be as well tuned for performance as a vendor supplied compiler. The adoption of co-arrays in the 2008 proposed standard is welcome, but at the end of 2005, nearly a year after the final publication of the Fortran2003 standard, there is still no fully compliant compiler. In contrast, UPC has been adopted into vendor compilers by several leading manufacturers, and Titanium circumvents this problem by translating to C++ and relying on a good implementation of that language. That part of the scientific community which still sees Fortran as its language of choice should be vociferous in urging vendors to track the new standard and incorporate co-arrays as soon as possible. Similarly, where UPC compilers are not available the C community should be lobbying, and it is well worth investigating whether a native Titanium compiler would perform better than a translator.

A potential barrier to the adoption by the scientific computing community of these languages is the investment already made in MPI programming. Many codes are mature and have MPI intimately

woven into their fabric. However, the results of the performance analyses discussed above, in particular references [26] and [27], suggest that re-programming critical portions of the algorithm in the appropriate one of these languages could have a major impact on performance. Fortunately it is quite possible to mix MPI and CAF or UPC (the case of Titanium is less clear, given its translation to C++), so it is possible to focus effort only on some portions of a code and make the transition to CAF or UPC in a staged fashion.

In summary, all three languages have a great deal to offer developers of parallel scientific computer programs, both in terms of performance through the one-sided communication model with its low latency, and in programmer productivity by expressing the parallelism in the program directly through language constructs rather than by (often cumbersome) library calls.

Acknowledgements

I would like to thank John Reid and Mike Ashworth for useful comments on earlier drafts of this report.

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. Numrich, R.W. and Reid,J.K. (2005), Co-Arrays in the next Fortran Standard, To appear in ACM Fortran Forum (2005). Available online as ftp://ftp.nag.co.uk/sc22wg5/N1601-N1650/N1642.pdf.

[7] Cray Fortran manual is available at http://docs.cray.com/books/S-3694-50/html-S-3694-50/S-3694-50-toc.html.

[8] Subset Co-Array Fortran into OpenMP Fortran http://www.co-array.org/caf2omp.htm

[9] www.hipersoft.ric.edu/caf.

[10] 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/.

[11] http://www.cray.com/products/x1e/.

[12 gcc Unified Parallel C, http://www.intrepid.com/upc/.

[13] HP upc overview, http://h30097.www3.hp.com/upc/.

[14] UPC projects at Michigan Technological University, http://www.upc.mtu.edu/.

[15] The Berkeley compiler can be downloaded from http://upc.lbl.gov/download/.

[16] 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.

[17] A list of Titanium compilers and other software is available online at http://titanium.cs.berkeley.edu/software.html.

[18] Bull, M. and Telford, S., Pogramming Models for Parallel Java Applications, UKHEC Technical report (2000), http://www.ukhec.ac.uk/publications/reports/paralleljava.pdf.

[19] Ashby, J V., Comparison of C, Fortran and Java for numerical computing DL Technical Report DL04003, also available at http://www.cse.clrc.ac.uk/arc/jaspa.shtml.

[20] http://www.hipersoft.rice.edu/caf/performance/index.html.

[21] http://www.nas.nasa.gov/Software/NPB/.

[23] Wallcraft, Alan J., SPMD OpenMP vs MPI for Ocean Models. http://www.co-array.org/ajw_omp99.ps.

[23] El-Ghazawi, T. and Cantonnet, F., UPC Potential and Performance: A NPB experimental study. SuperComputing 2002, p1 (2002), also at http://www.gwu.edu/~upc/publications/SC02paper.pdf.

[24] El-Ghazawi, T., Cantonnet, F., Yiyi Yao and Vetter, J., Evaluation of UPC on the Cray X1. http://www.gwu.edu/~upc/publications/cug05.pdf

[25] STREAM: Sustainable Memory Bandwidth in High Performance Computers. http://www.cs.virginia.edu/stream/.

[26] Dawson, J., Co-array Fortran for Productivity and Performance, Army HPC Research Center Bulletin, **14**, 4 (2004)

[27] Johnson, A. A., Unified Parallel C in CFD Codes on the Cray X1 system, Army HPC Research Center Bulletin, **14**, 13 (2004)

[28] Datta, K., Bonachea, D. and Yelick, K. Titanium Performance and Potential: an NPB Experimental Study. http://titanium.cs.berkeley.edu/papers/datta-bonachea-yelick-ti_npb.pdf.