RAL-TR-2007-001

CCLRC

Co-arrays in the next Fortran Standard

John Reid and Robert W. Numrich

January 2007

Council for the Central Laboratory of the Research Councils

Library and Information Services
CCLRC Rutherford Appleton Laboratory
Chilton Didcot
Oxfordshire OX11 0QX
UK
Tel: +44 (0)1235 445384
Fax: +44(0)1235 446403
Email: library@rl.ac.uk

CCLRC reports are available online at:
http://www.clrc.ac.uk/Activity/ACTIVITY=Publications;SECTION=225;

ISSN 1358-6254

RAL-TR-2007-001

Co-arrays in the next Fortran Standard[1,2]

John Reid[3] and Robert W. Numrich[4]

Abstract

The WG5 committee, at its meeting in Delft, May 2005, decided to include co-arrays in the next Fortran Standard. A Fortran program containing co-arrays is interpreted as if it were replicated a fixed number of times and all copies were executed asynchronously. Each copy has its own set of data objects and is called an image. The array syntax of Fortran is extended with additional trailing subscripts in square brackets to give a clear and straightforward representation of access to data on other images.

References without square brackets are to local data, so code that can run independently is uncluttered. Any occurrence of square brackets is a warning about communication between images.

The additional syntax requires support in the compiler, but it has been designed to be easy to implement and to give the compiler scope both to apply its optimizations within each image and to optimize the communication between images.

The extension includes execution control statements for synchronizing images and intrinsic procedures to return the number of images, to return the index of the current image, and to perform collective operations.

The paper does not attempt to describe the full details of the feature as it now appears in the draft of the new standard. Instead, we describe a subset and demonstrate the use of this subset with examples.

Keywords: Fortran 95, parallel programming, SPMD.

---

[1] To appear in special issue of *Scientific Programming* commemorating 50th anniversary of Fortran.

[2] Current reports available from "http://www.numerical.rl.ac.uk/reports/reports.html".

[3] Convener, ISO/IEC Fortran Committee WG5.

[4] Minnesota Supercomputing Institute, University of Minnesota, Minneapolis, MN USA.

Computational Science and Engineering Department,
Atlas Centre, Rutherford Appleton Laboratory,
Oxon OX11 0QX, England.

January 19, 2007

1 Introduction

The co-array extension to Fortran consists of two new features. One is a simple syntactic extension called the co-dimension that describes data distribution. The other is the Single-Program-Multiple-Data (SPMD) programming model that describes work distribution. The programmer moves data between replicated objects called co-arrays using explicit co-array syntax that looks and feels like normal Fortran. The programmer also determines the execution path through replicated copies of the program using explicit control constructs and synchronizations.

The programmer uses co-array syntax only where it is needed. A reference to a co-array with no square brackets attached to it is a normal reference to the local object. Since most references to data objects in a parallel code should be local, co-array syntax should appear only in isolated parts of the code. If not, the syntax acts as a visual flag to the programmer that too much communication among images may be taking place. It also acts as a flag to the compiler to generate code that avoids latency whenever possible.

Because co-arrays are integrated into the language, remote references automatically gain the services of Fortran's basic data capabilities, including the typing system and automatic type conversions in assignments, information about structure layout, and even object-oriented features.

The co-array feature adopted by WG5 was formerly known as Co-Array Fortran, an informal extension to Fortran 95 by Numrich and Reid (Numrich and Reid 1998). Co-Array Fortran itself was formerly known as F^{--}, which evolved from a simple programming model for the CRAY-T3D described only in internal Technical Reports at Cray Research in the early 1990s. The first informal definition (Numrich 1997) was restricted to the Fortran 77 language and used a different syntax to represent co-arrays. It was extended informally to Fortran 90 by Numrich et al (Numrich and Steidel 1997, Numrich, Steidel, Johnson, de Dinechin, Elsesser, Fischer and MacDonald 1998*b*), and defined more precisely for Fortran 95 by Numrich and Reid (Numrich and Reid 1998).

Co-Array Fortran has been incorporated into the Cray Fortran compiler and various applications have been converted to the syntax (Numrich 2005*a*, Numrich 2005*b*, Numrich, Reid and Kim 1998*a*). It was also an unsupported feature of the SGI Fortran compiler (Numrich et al. 1998*a*) for a brief period in the late 1990s, but it never became a supported product. A portable compiling system for a subset of the extension has been implemented by Dotsenko, Coarfa, and Mellor-Crummey (Coarfa, Dotsenko and Mellor-Crummey 2004). It performs source-to-source transformation of co-array code to standard Fortran 90 code augmented with calls to platform-specific communication protocols. One instantiation uses the Aggregate Remote Memory Copy Interface (ARMCI) library for one-sided communication (Nieplocha and Carpenter 1999) and another uses loads and stores for communication on shared-memory machines. Experience with this compiling system is related in several papers by this group (Coarfa, Dotsenko, Eckhardt and Mellor-Crummey 2003, Dotsenko, Coarfa and Mellor-Crummey 2004*a*, Dotsenko, Coarfa, Mellor-Crummey and Chavarría-Miranda 2004*b*).

Reid (Reid 2005*a*, Reid 2005*b*) proposed that co-arrays be included in the revision of Fortran that is planned for 2008. The ISO/IEC Fortran Committee agreed to include co-arrays in their May 2005 meeting (ISO/IEC Fortran Standards Committee, ISO/IEC JTC 1/SC22/WG5 n.d.) but made some changes. The US Fortran Standards Technical Committee J3 (US Fortran Standards Committee J3, a technical subcommittee of the InterNational Committee for Information Technology Standards (INCITS) n.d.), the primary development body, have

made further changes since then. This paper contains a description of the main features of the proposal as it now stands with applications to some simple examples. For a more complete summary, see Reid (Reid 2007), and for more extensive discussion and examples, see Numrich and Reid (Numrich and Reid 1998, Numrich and Reid 2005).

2 Execution model

The co-array extension adopts the SPMD programming model. A single program is replicated a fixed number of times with each replication called an `image`. The name image is used to avoid any connotation of how a particular run-time system might implement the co-array extension, for example, using a thread model versus a process model, or how an image is mapped onto a physical processor. The number of images should be thought of as the number of virtual or logical processors; it may be the same as the number of physical processors, may be more than the number of physical processors, or may be less than the number of physical processors.

A particular implementation may permit the number of images to be chosen at compile-time, at link-time, or at run-time. Once fixed, however, it does not change. The programmer may retrieve the number of images at run-time through the intrinsic function `num_images()` and may retrieve the unique image index from the intrinsic function `this_image()`, which returns the index of the image that executes the function. Images are numbered from one in the normal Fortran fashion.

Each image executes asynchronously, and the normal rules of Fortran apply. The programmer determines the execution path on each image using normal Fortran control constructs and explicit synchronizations with the help of the image index. The actual execution path followed on each image may, and probably will, be different from image to image.

Provided no image encounters an error condition that may be expected to terminate its execution, all images continue execution until they have all executed a `stop` or an `end program` statement. We call this 'normal termination'. If an error condition occurs on one image, the computation is probably flawed, and it is desirable to stop the other images as soon as is practicable. We call this 'error termination'.

An image that initiates normal termination does not complete termination until all other images have initiated either normal or error termination allowing its data to remain accessible to the other images. The new statement `all stop` allows the programmer to terminate execution on all images. When this statement executes on an image or an error condition occurs on an image, the image initiates error termination for itself and causes all other images that have not already initiated termination to initiate error termination. Within the performance limits of the processor's ability to send signals to other images, this propagation of error termination should be immediate and cause the whole calculation to stop.

3 Co-dimensions

Each image has its own set of replicated data objects with the same names in each image. All of the objects may be accessed in the normal Fortran way on each image. Some objects are also co-array objects that may be accessed across images.

3.1 Specifying co-array objects

Co-arrays are declared with **co-dimensions** in square brackets immediately following dimensions in parentheses or in place of them. For example, the declarations,

```
real           :: a(20)[20,*]      ! An array co-array
real           :: c[*], d[*]       ! Scalar co-arrays
character      :: b(20)[-5:20,0:*]
integer        :: ib(10)[*]
type(interval) :: s[20,*]
```

define co-arrays of various types. The form for the dimensions in square brackets is the same as that for the dimensions in parentheses for an assumed-size array. The total number of subscripts plus co-subscripts is limited to 15.

Each co-array exists with the same name and has the same shape on each image. In the example, each image has a real co-array `a` of size 20 in its own local memory, two scalars `c` and `d`, and so forth. There is no concept of a shared array without co-subscripts that spans the set of images. The programmer is responsible for decomposing such a global array into local pieces that reside on each local image.

A co-array cannot be a pointer, but it may be allocatable,

```
real, allocatable :: u(:)[:], v[:,:]
```

The allocate statement has been extended so that the co-bounds can be specified at run-time,

```
allocate ( u(10)[*], v[-1:p,0:*] )
```

The programmer might, for example, pick the variable `p` at run-time to set a particular logical relationship between the actual number of images. The co-bounds must always be included in the allocate statement and the upper bound for the final co-dimension must always be an asterisk. The value of each bound, co-bound, or length type parameter must be the same on all images.

The data in parentheses in a co-array's declaration or allocation statement specify the normal **rank**, **bounds**, **extents**, **size**, and **shape** of the co-array. They determine the access rules within an image's own memory. The data in square brackets specify the co-array's **co-rank**, **co-bounds**, and **co-extents**. They determine the access rules from one image to another. The syntax and semantics for co-dimensions mirror those of assumed-size arrays. The final extent is always indicated with an asterisk, and there is no final co-extent, no final upper co-bound, and no co-shape. Since a co-array exists on all images, the co-size of a co-array is always equal to the number of images. This convention allows the programmer to write code independent of the number of images the code will eventually use.

Co-dimensions allow the programmer to define a logical relationship among images for each co-array. This logical relationship may be different for different co-arrays and may even change for the same co-array across procedure calls. The set of subscript indices that correspond to the current image are always calculated from the declaration of the co-array in the current procedure on that image. The programmer may retrieve them from the alternative form of the intrinsic function `this_image(z)` for any co-array `z`. For the co-array,

```
real :: z[10,0:9,0:*]
```

for example, `this_image()` has the value 5 on image 5, and `this_image(z)` has the value `(/5,0,0/)`. For the same example on image 213, `this_image(z)` has the value `(/3,1,2/)`.

Conversely, the programmer may retrieve the image index that corresponds to a set of co-subscript indices from the inverse intrinsic function `image_index`. For example, the value of `image_index(z,(/5,0,0/))` is 5 on every image, and the value of `image_index(z,(/3,1,2/))` is 213.

3.2 Referencing images

Co-array indices in square brackets provide a convenient notation for accessing an object within another image's memory in the same way that array indices in parentheses provide a notation for accessing an object within an image's own memory. A reference to an object without square brackets is always a reference to the object on the invoking image. A reference to an object with co-subscripts enclosed in square brackets is a reference to the object on the image corresponding to the co-subscripts. Co-subscripts map to an **image index** in the same way as array subscripts map to a position in array element order. For example, if two objects, `x` and `y`, are declared as co-arrays,

```
real :: x(n)[-3:*],y(n)[5,*]
```

then an image that executes the statement,

```
x(:) = y(:)[q,r]
```

copies the array `y(:)` from the image corresponding to co-subscripts `[q,r]` into array `x(:)` in its own memory.

Images execute statements they encounter by the normal rules of Fortran. Co-subscripts do not determine how work is assigned to different images. They are a logical description of the relationship between memory assigned to different images. Whether the co-subscripts map to the executing image or not has no bearing on whether that image evaluates the expression or assignment. For example, any image encountering the statement,

```
x(:)[p] = x(:) + y(:)[q,r]
```

executes it no matter what values are given for `p`, `q` and `r`. The invoking image copies array `y(:)` from the image corresponding to `[q,r]`, adds it to its local data `x(:)`, and stores it to the remote array `x(:)` on the image corresponding to `p`.

The co-subscripts `[q,r]` map to an image index based on the co-dimensions in the declaration statement for co-array `y`. The co-subscript `[p]` maps to an image index based on the co-dimension in the declaration statement for co-array `x`. Neither of these images is involved in the execution of this statement. If it is necessary to restrict execution to a specific image, the programmer must use a Fortran `if` or `case` statement in the normal way. If it is necessary to avoid memory race conditions, the programmer must use synchronization statements such as those discussed in Section 4.

The programmer is responsible for generating valid co-subscripts that map to valid image indices. For example, for the co-array `y` declared at the start of this subsection, a reference to `y(:)[1,4]` on 16 images is valid since it has co-subscript order value 16, but a reference to `y(:)[2,4]` is invalid since it has co-subscript order value 17.

3.3 Co-arrays as actual arguments of procedures

A co-array may be an actual argument of a procedure either with or without co-subscripts in square brackets. If the interface is not explicit, the compiler assumes that no dummy argument is a co-array. If the actual argument contains co-subscripts,

```
real :: x[p,*]
call mysub(x[q,r])
```

the compiler is likely to make a local copy of `x[q,r]` before entering the routine and a remote copy upon exit from the routine. To limit the number of copies made, the programmer must supply an explicit interface. For example, the interface,

```
interface
  subroutine mysub(x)
    real,intent(out) :: x
  end subroutine mysub
end interface
```

alerts the compiler that only copy out is required.

This feature is very important. It means that all the intrinsic procedures of Fortran 2003 and any procedure that has been written for a uni-processor is available for co-arrays. Here is a very simple example:

```
s = sum(a(:)[p])
```

3.4 Co-arrays as dummy arguments of procedures

A dummy argument of a procedure is permitted to be a co-array. It may be of explicit shape, assumed size, assumed shape, or allocatable:

```
subroutine subr(n,w,x,y,z)
   integer :: n
   real :: w(n)[n,*]  ! Explicit shape
   real :: x(n,*)[*]  ! Assumed size
   real :: y(:,:)[*]  ! Assumed shape
   real, allocatable :: z(:)[:,:]
```

When the procedure is called, the corresponding actual argument must be a co-array or a subobject of a co-array with no co-subscripts. The association is with the whole or part of the co-array itself and not with a copy, so the programmer must avoid calls for which making a copy of an argument is necessary. For example, a noncontiguous section such as `a(1:6:2)` is not acceptable if the actual argument is an explicit-size array. Copying is disallowed because the called procedure may access the corresponding co-array on another image and that other image may have not yet have made a copy of its co-array or indeed may never do so since it is executing some other procedure.

The interface is required to be explicit so that the compiler knows that the dummy argument is a co-array. Here is an example

```
interface
  subroutine sub(x,y)
    real :: x(:)[*], y(:)[*]
  end subroutine sub
end interface
   :
real, allocatable :: a(:)[:], b(:,:)[:]
 :
call sub(a(:),b(1,:))
```

If a dummy argument is an allocatable co-array, the corresponding actual argument must be an allocatable co-array of the same rank and co-rank.

The rules for resolving generic procedure references take no account of the co-array properties and are therefore unchanged. The rules cannot be extended to allow overloading of array and co-array versions since the syntactic form of an actual argument would be the same in the two cases.

Each image independently associates its non-allocatable co-array dummy argument with an actual co-array and defines the co-rank and co-bounds afresh. It uses these to interpret each reference to an object with co-subscripts, taking no account of whether the remote image is executing the same procedure with the same co-array.

3.5 Co-arrays in procedures

Automatic-array co-arrays are not permitted (an automatic array is not a dummy array and has one or more bounds that are not constant expressions). Were they permitted, it would be necessary to require image synchronization, both after memory is allocated on entry and before memory is deallocated on return. A co-array function result is like an automatic co-array and is disallowed for the same reasons.

Unless it is allocatable or a dummy argument, a co-array that is declared in a procedure must be given the `save` attribute. This ensures that the system does not reallocate it on every entry. An allocatable co-array is not required to have the `save` attribute because a recursive procedure may need separate allocatable arrays at more than one level of recursion.

4 Execution control statements

Each image executes on its own as a Fortran program without regard to the execution of other images. Each image can define data in any other image's memory and may access data from any other image's memory at any time during execution. A memory race condition may occur whenever an image alters the contents of a co-array, even in its own memory. It is the programmer's responsibility to avoid memory race conditions by using execution control statements. The programmer must ensure that when an image assigns a new value to a co-array, no other image still needs the old value. And when an image accesses the value of a co-array, it receives the expected value, either an old value unchanged by other images or a new value changed in a controlled way by another image. In this and later sections, we discuss the more important

execution control statements available to the programmer: `sync all`, `allocate`, `deallocate`, `notify`, `query`, `sync team`, `call form-team`, `open`, `close`, and a call of a collective subroutine.

For code between execution control statements, the compiler is free to use all its normal optimization techniques as if only one image were present. Execution of a execution control statement usually suppresses compiler optimizations that might reorder memory operations across the statement. All memory operations initiated by an image before the statement must complete before any memory operations following the statement are initiated by the image unless the compiler can establish that failure to do so could not alter processing on another image.

4.1 The `sync all` statement

The `sync all` statement provides a barrier for the important case where all images must synchronize before moving forward. Any statement executed before the barrier on image P is also executed before any statement executed after the barrier on any other image Q. The normal rules relating to execution order apply. In particular, if the value of a co-array variable is changed by any image before the barrier, it is accessible to all images after the barrier. The programmer must ensure that if one image changes a co-array value between two successive barriers, another image does not reference or define that value between the barriers.

Figure 4.1 shows two simple examples of the use of the `sync all` statement. In the code shown on the left side of the figure, image 1 reads data from standard input and broadcasts it to other images. The first `sync all` ensures that image 1 does not interfere with any previous use of `p` by another image. The second `sync all` ensures that another image does not access `p` before the new value has been set by image 1.

Alternatively, in the code shown on the right side of the figure, image 1 still reads data from standard input but does not broadcast the value to the other images. Instead, each other image makes a copy. The first and last `sync all` statements are there for the same reasons as before. The two new `sync all` statements ensure that no race condition occurs. The other images wait at their `sync all` statement for image 1 to finish. When image 1 reaches its `sync all` statement, the other images are released, and they copy the value of `p` from image 1. This alternative version illustrates that the `sync all` statement on one image may correspond to a different `sync all` statement on another image.

Figure 4.1: Read data on image 1 and define it on other images.

```
real :: p[*]                        |   real :: p[*]
  ...                               |     ...
sync all                            |   sync all
if (this_image()==1) then           |   if (this_image()==1) then
   read (*,*) p                     |     read(*,*) p
                                    |     sync all
   do i = 2, num_images()           |   else
      p[i] = p                      |     sync all
   end do                           |     p = p[1]
end if                              |   end if
sync all                            |   sync all
```

4.2 The `allocate` and `deallocate` statements

There is an implicit synchronization of all images in association with each `allocate` statement that involves one or more co-arrays. Images do not commence executing subsequent statements until all images finish executing the same `allocate` statement. The programmer is responsible for ensuring that the shape and co-shape are the same on all images. Similarly, for a `deallocate` statement involving one or more co-arrays, all images delay making the deallocations until they are all about to execute the same `deallocate` statement. Without these rules, an image might reference data on another image that has not yet been allocated or has already been deallocated.

For an allocatable co-array declared in a procedure without the `save` attribute, if the co-array is still allocated when a `return` statement or an `end` statement is executed, there is an implicit deallocation (and associated synchronization) before the procedure is exited.

Fortran 2003 allows the shapes to disagree in an intrinsic array assignment to an allocatable array; the system performs the appropriate reallocation. Such disagreement is not permitted for an allocatable co-array, since it would involve synchronization.

4.3 The `notify` and `query` statements

The `notify` and `query` statements support an asynchronous programming style to enable better load balancing between images. Rather than waiting for all the other images at a `sync all` statement, an image may proceed with calculations while other images catch up.

Each of these statements specifies an *image_set* in parentheses,

`notify(`*image_set*`)`
`query(`*image_set*`)`

as an integer scalar holding an image index, as a rank-one integer array holding distinct image indices, or as an asterisk to indicate all images. The `notify` statement is non-blocking; the image executing it continues execution without waiting for any of the images in its image set. The `query` statement is blocking; the image executing it waits until it receives notification from every image in its image set.

An alternative form of the `query` statement,

`query(`*image_set*`, ready=`*scalar_logical_variable*`)`

is non-blocking. A `query` statement is said to be **satisfied** on completion if it has no `ready=` specifier or if its `ready=` variable has been set to true. If the `ready=` value is false, we expect the image to execute further `query` statements until it executes a satisfied `query` statement.

These control statements interact with each other in the following way. Let $N_{P,Q}$ be the number of `notify` statements executed on image P with image Q in its image set. Let $S_{Q,P}$ be the number of satisfied `query` statements executed by image Q with image P in its image set. A `query` statement on image Q without a `ready=` variable delays until $N_{P,Q} > S_{Q,P}$ for all images P in its image set. A `query` statement on image Q with a `ready=` variable sets its value to true if and only if $N_{P,Q} > S_{Q,P}$ for all images P in its image set.

An execution of a `notify` statement on image P with image Q in its image set `corresponds` to a satisfied `query` statement on image Q with image P in its image set, if the number $N_{P,Q}$ of such `notify` statements on image P is the same as the number $S_{Q,P}$ of such satisfied `query` statements on image Q.

Any statement executed before a `notify` statement on image P is also executed before any statement executed after the corresponding `query` statement on image Q. The normal rules relating to execution order apply. In particular, if the value of a co-array variable is changed by image P before the `notify` statement, it is accessible to image Q after the `query` statement.

Figure 4.2 shows two examples of how to use `notify/query` pairs. On the left side, image `p` computes the value of some variable `z` and then notifies image `q`. Image `q` executes a blocking query to image `p` and waits for notification before reading the value of `z`. On the right side, image `p` does the same thing, but image `q` executes non-blocking `query` statements. Image `q` continues execution doing work not dependent on the value of `z` and loops back to check for a true value for its logical variable. Only then does it read the value of `z` from image `p`.

Figure 4.2: Using `notify` and `query` statements.

```
real :: z[*]                          |  real :: z[*]
  ...                                 |  logical :: ready=.false.
  ...                                 |     ...
sync all                              |  sync all
if(this_image()==p) then              |  if(this_image()==p) then
   z = some_function()                |     z = some_function()
   notify(q)                          |     notify(q)
 elseif(this_image()==q) then         |  elseif(this_image()==q) then
   query(p)                           |     do while(.not.ready)
   z = z[p]                           |        ...  ! Work not dependent on z
 end if                               |        query(p,ready=ready)
                                      |     end do
                                      |     z = z[p]
                                      |  end if
```

4.4 The `sync team` statement

A team of images is defined by the value of a scalar of type `image_team` of the intrinsic module `ISO_Fortran_env`. All the components of the type are `private`, and none is a co-array. A variable of this type has default initialization to a value that identifies an empty team.

A team is established by executing the intrinsic subroutine,

```
call form_team(image_team,list)
```

with a rank-one integer array `list` containing the image index for each member of the team. Every image of the team must call this procedure, and none of the team members continues beyond the call until all team members have made the call. The intrinsic function,

```
list = team_images(team)
```

returns the images in the team as a rank-one array of type default integer.

The intention is to allow the processor to calculate optimized communication patterns during the call of `form_team` and to store them for all subsequent team actions. It is quite likely that

different images would need different data, so it is inappropriate to copy values between images. For this reason, a co-array is not permitted to be of type `image_team`.

The statement,

```
sync team (team)
```

behaves like `sync all` except that it applies only to the images of the team. The executing image must be a member of the team.

5 Input/output

The main mechanism for parallel I/O is the direct-access file where the programmer arranges that the same record is never accessed by more than one image without an intervening synchronization. If two images write to the same record of a direct-access file, it is the programmer's responsibility to separate the writes by appropriate `flush` statements and image synchronizations.

The `open` statement has been augmented to allow several images to be connected on the same unit. The set of images connected to a unit is called a **connect team**, specified as `team=connect` in the `open` statement by a variable `connect` of type `image_team`. The executing image must be a member of the team, and each member of the team must execute the same `open` statement. There is an implicit team synchronization. If there is no `team=` specifier in an `open` statement, the connect team is the executing image. In that case, the normal rules for opening a file apply.

If an `open` statement specifies a connect team of more than one image, the connection must be either direct access or sequential access with `action='write'`. If the connection is for `write`, the processor ensures that once an image P commences transferring the data of a record to the file, no other image Q transfers data to the file until the whole record from image P has been transferred. In other words, each record in an external file arises from a single image. The `backspace`, `rewind`, and `endfile` statements are not permitted for such a unit.

The default unit for input, either `*` in a `read` statement or `input_unit` in the intrinsic module `iso_fortran_env`, has a connect team consisting of image one only. Any other preconnected unit has a connect team consisting of all the images, but a `read` statement for such a unit is permitted only on image one.

The processor is permitted to hold data in a buffer and to transfer several whole records on execution of a `flush` statement. Without a `flush`, all writes may be buffered locally until the file is closed. Execution of a `flush` statement is required only when a record written by one image is read by another or when the relative order of writes from images is important. The `flush` statement ensures that any changed records in buffers are copied to the file itself or to a replication of the file that other images access. Executing a `flush` statement also has the effect of requiring the reloading of I/O buffers in case the file has been altered by another image.

If an image executes a `close` statement, all images in the connect team must execute the same `close` statement for the unit with the same status. There is an implicit team synchronization.

6 An example: LU Factorization

As an example, we examine in some detail a parallel algorithm for LU factorization, with row pivoting, of a square matrix of order `n`. We decompose the matrix by columns into blocks of size

n×nb. If the program executes with p images, these blocks are held contiguously on images (1, 2, ..., p, 1, 2, ..., p, ...) as a block-column, cyclic-wrapped distribution.

Figure 6.3: Program for LU factorization.

```
program LU
   integer                          :: n[*]        ! Matrix order
   integer                          :: nb[*]       ! Block size
   character(len=100)               :: file[*]     ! I/O File name
   double precision, allocatable :: A(:,:)[:]   ! Distributed matrix
   integer,allocatable              :: ipiv(:)[:]  ! Row indices of the pivots
   integer                          :: p           ! Number of images
   integer                          :: Iam         ! This image
   p = num_images()
   Iam = this_image()
   if(Iam == 1) then
       read(*,*) n,nb,file
       notify(*)
   else
       query(1)
       file=file[1]
       n=n[1]; nb=nb[1]
   end if
   allocate(A(n,nb*ceiling(real(n)/(nb*p)))[*])
   allocate(ipiv(nb*ceiling(real(n)/(nb*p)))[*])
   call dataIN(file,n,nb,A)
   call LUfactorize(n,nb,A,ipiv)
   call dataOUT(file,n,nb,A)
   deallocate(A,ipiv)
end program LU
```

Figure 6.3 shows a program that illustrates how we can tie together all the pieces of the co-array extension to carry out an LU factorization. Image 1 reads the problem size, the block size, and the input file name from standard input and notifies the other images once it has these values. The other images wait on a blocking `query` until they are released and then read the values of the input variables from image 1. All images then allocate memory for the matrix `A` and the pivot vector `ipiv`. These arrays are co-arrays so there is an implied synchronization for these two allocate statements.

The matrix `A` has been previously written to a direct access file column-by-column. The procedure `dataIN` shown in Figure 6.4 reads the data from that file. There is a corresponding procedure `dataOUT` that writes the data out to the file.

Our code for the actual factorization is based on the LAPACK procedure `dgetrf` (Anderson, Bai, Bischof, Blackford, Demmel, Dongarra, Du Croz, Greenbaum, Hammarling, McKenney and Sorensen 1999) that performs a blocked right-looking factorization. For each block column, `dgetrf` performs the following steps:

1. `dgetf2`: Apply interchanges within the block column and factor it.

2. `dlaswp`: Apply the interchanges to the previous columns, which belong to the factor L.

Figure 6.4: Parallel I/O from a file.

```
subroutine dataIN(inputFile,n,nb,A)
  character(len=*),intent(in)    :: inputFile   ! I/O file name
  integer, intent(in)            :: n           ! Matrix order
  integer, intent(in)            :: nb          ! Block size
  double precision,  intent(out) :: A(n,*)      ! Distributed matrix
  integer                        :: len         ! Length of the block column
  integer                        :: j           ! Loop index
  integer                        :: ja=1        ! Column in A
  integer                        :: p           ! Number of images
  integer,parameter              :: unit=999    ! Unit number
  type(image_team)               :: team        ! Team
  p = num_images()
  call form_team( all, (/ (j,j=1,p) /) )
  inquire (iolength=len) A(:,1)
  open(unit,file=inputFile,status='old',access='direct',recl=len,team=all)
  j = (this_image()-1)*nb+1
  do while(j<=n)
      read(unit,rec=j) A(:,ja)
      j = j+1; ja = ja+1
      if (mod(ja,nb)==1) j = j+nb*(p-1)
  end do
  close(unit)
end subroutine dataIN
```

3. `dlaswp`: Apply the interchanges to the subsequent columns.

4. `dtrsm`: Compute the block row of U.

5. `dgemm`: Update the submatrix of subsequent rows and columns.

For large matrix order `n` compared with the block size `nb`, most of the work is performed in the Level-3 BLAS code `dgemm`, which is likely to be highly optimized on every machine.

Figure 6.5 shows our co-array version. The major loop is over the block columns, as in `dgetrf`. The image that holds the pivot block column uses `dgetf2` to factor it, as in step 1. After a synchronization, each image uses co-array syntax to make a copy of the pivot block column and the interchanges. Each image then executes step 2 for those previous columns that it holds and steps 3 to 5 for those subsequent columns that it holds. Note that, for a large problem, the bulk of the work is in step 5 and is performed in parallel as each image executes `dgemm` independently.

Table 6.1 shows the execution times on a Cray-X1E for a matrix of order `n`=5000 with block size `nb`=50 as a function of the number of images. Each processor has two vector pipelines, with frequency 1.13 MHz, capable of two floating-point operations each for a peak computational power of 4.52 Gflop/s. Each image corresponds to four processors working together (called MSP mode) with a peak computational power of 18.1 Gflop/s.

With one image, our code obtains the fraction, $15.8/18.1 = 0.87$, of the peak theoretical power. As shown in Figure 6.6, our code scales quite well, as $p^{-2/3}$ although not perfectly as p^{-1}, as the number of images increases to 32. The wiggles in the curve are caused by details of the Cray-

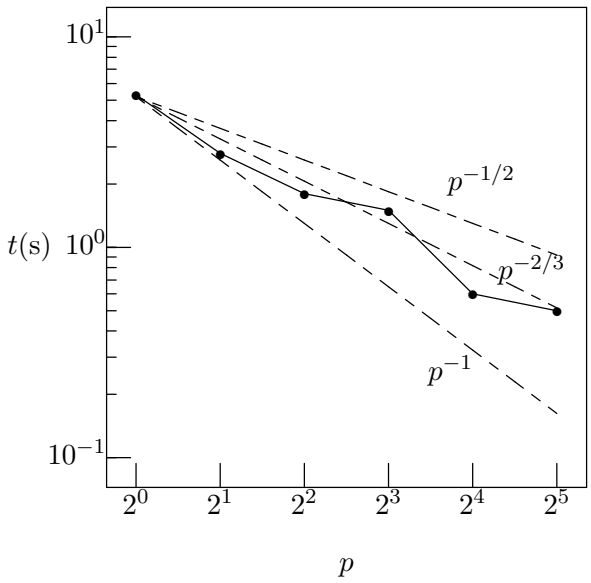
Figure 6.5: Code for LU factorization.

```
subroutine luFactorize(n, nb, A, ipiv)
  integer,intent(in)    :: n                  ! Matrix order
  integer,intent(in)    :: nb                 ! Block size
  double precision,intent(inout) :: A(n,*)[*] ! Matrix to factorized
  integer,intent(inout) :: ipiv(*)[*]         ! Row indices of the pivots
  integer               :: me                 ! This image
  integer               :: p                  ! Number of images
  integer               :: pivimage=1         ! Index of the image handling current pivot block
  integer               :: info               ! Temporary variable
  integer               :: i                  ! Row index
  integer               :: j                  ! Pivot column index
  integer               :: jb                 ! Size of the pivot block
  integer               :: lastcol            ! The local index of the last column on this image
  integer               :: lpiv(nb)           ! Local copy of ipiv for pivot block
  integer               :: pivcol=1           ! The local index of the pivot column on its image
  integer               :: nextpiv=1          ! The local index of the next column on this image
  double precision      :: pivblock(n,nb)     ! Copy of pivot block column
  double precision      :: one=1.0
  p  = num_images()
  me = this_image()
  lastcol = nb*ceiling(real(n/nb+1-me)/p)
  if ( mod(n-1,p*nb)/nb+1 == me ) lastcol = lastcol + mod(n,nb)
  Loop: do j = 1, n, nb
          jb = min(nb,n+1-j) ! Size of the pivot block
          ! Factor diagonal and subdiagonal blocks on pivimage.
          if (pivimage==me)  call dgetf2( n-J+1, jb, A( j, pivcol ), n, ipiv( pivcol ), info )
  SYNC ALL
          ! Copy the pivot columns from pivimage
  PIVBLOCK(J:N,1:JB) = A(J:N,PIVCOL:PIVCOL+JB-1)[PIVIMAGE]
  LPIV(1:JB) = IPIV(PIVCOL:PIVCOL+JB-1)[PIVIMAGE]
          ! Apply interchanges to columns 1:nextpiv-1.
          call dlaswp( nextpiv-1, A(j,1), n, 1, jb, lpiv, 1 )
          if (pivimage==me) nextpiv=nextpiv+jb
          ! Apply interchanges to columns nextpiv:lastcol.
          if (nextpiv <= lastcol) then
            call dlaswp( lastcol-nextpiv+1, A(j,nextpiv), n, 1, jb, lpiv, 1 )
          ! Compute block row of U
            call dtrsm('l','l','n','u',jb,lastcol-nextpiv+1,one,pivblock(j,1),n,A(J,nextpiv),n)
          ! Update trailing submatrix
            call dgemm('n','n',n-j-jb+1,lastcol-nextpiv+1,jb,-one, pivblock(j+jb,1), &
                       n,  A(J,nextpiv),n,one,A(J+jb,nextpiv),n)
          end if
  SYNC ALL
          ! Adjust the pivot indices.
          if (pivimage==me) then
            do i = pivcol, pivcol+jb-1; ipiv(i) = j - 1 + ipiv(i);  end do
          end if
          ! Update pivimage, wrapping if necessary.
          pivimage = pivimage + 1
          if (pivimage>p) then; pivimage = 1; pivcol = pivcol + nb; end if
  end do Loop
end subroutine luFactorize
```

Table 6.1: Execution time and computational power for a Cray-X1E for matrix size $n = 5000$ and block size $nb = 50$.

images	time(s)	Gflop/s
1	5.3	15.8
2	2.8	29.5
4	1.8	45.8
8	1.5	55.5
16	0.61	136
32	0.51	162

Figure 6.6: Execution time for a Cray-X1E as a function of the number of images. The matrix size is $n = 5000$ and the block size is $nb = 50$. Perfect scaling corresponds to the dashed line marked p^{-1}. Our code scales approximately as $p^{-2/3}$.



X1E architecture (Numrich 2007). We are aware that better scaling may be obtained by a full decomposition in both rows and columns rather than decomposing by columns alone (Dongarra, van de Geijn and Walker 1994, Dongarra and Walker 1995, Numrich 2005*a*, Numrich 2005*b*, Numrich 2007), but the additional synchronization required and the additional complications required for pivoting make it run slower for modest values of `n`. Besides, such a code is more complicated and would not be appropriate to show in full here.

7 Co-arrays of derived type

By definition, co-arrays are the same size on each image. This restriction is not sufficient for handling applications where the sizes of arrays are determined dynamically and different images may hold different amounts of data. But a co-array may be of a derived type with pointer or allocatable components. The targets of such components are always local with shapes that may vary from image to image. On each image, the component is allocated locally or is pointer assigned to a local target with the desired size for that image. If an image has no data assigned to it, it need not allocate the component or associate the pointer.

For example, the derived type,

```
type  VelocityField
  real,allocatable  :: x(:),y(:),z(:)
  type(FieldMap),pointer :: map
end type
```

contains allocatable components that hold the values of a velocity field. It also contains a pointer of another derived type that holds a map describing how the velocity field is distributed across images.

A variable of this type may be declared as a co-array,

```
type(VelocityField),allocatable :: v[:]
```

Allocation of an instance of this type as a co-array,

```
allocate(v[*])
```

is a collective allocation with implied synchronization across images.

Each image allocates memory for its own data components,

```
allocate(v%x(0:n+1),v%y(0:n+1),v%z(0:n+1))
```

using the normal Fortran `allocate` statement. These memory allocations are independent of each other, there is no synchronization involved, and the memory allocated may be, and probably will be, at different locations on the local heap for each image. The programmer must supply synchronization following these allocations before referencing data across images.

Pointer assignment takes place locally on each image. For example, if `map` and `localMap` are of type `fieldMap`, the code,

```
type(fieldMap),target  :: map
type(fieldMap),pointer :: localMap
v%map => map
localMap => v%map
```

represents pointer assignment of the pointer component to a map on the invoking image, in the first case, and a local pointer assignment to the pointer component, in the second case.

Pointers are not allowed to be co-arrays, and there is no concept of pointer assignment across images. For example, the derived-type intrinsic assignments,

```
v[q] = v     ! Pointer component of v[q] becomes undefined
v    = v[q]  ! Pointer component of v    becomes undefined
```

result in these pointer components becoming undefined when such assignments are executed by an image other than the one with co-subscript `q`. The data components of the structure, of course, are copied from one image to the other.

Co-array syntax supplies a straightforward way to access data from the components of such structures. The co-subscript in square brackets is associated with the variable `v`, not with its component, for example,

```
vX(1:n) = v[p]%x(1:n)
```

This statement requires the image that executes it to obtain the address, on image `p`, of the allocated component `v%x` and then to copy the data in the array itself to the local array `vX`. A typical example for using such data structures is the exchange of halo cells from neighboring images (Numrich et al. 1998*a*, Numrich et al. 1998*b*),

```
v%x(n+1)=v[me+1]%x(1)
v%x(0)  =v[me-1]%x(n)
```

Data manipulation of this kind is handled awkwardly, if at all, in other programming models. Its natural expression in co-array syntax gives the programmer power and flexibility for writing parallel code.

Note that the presence of co-arrays with pointer components may lead to variables that are not co-arrays being accessed from other images or having their values changed by other images. For example, the variable `map` may be accessed from another image through the co-array `v`. Of course, this cannot happen for a variable that does not have the `target` attribute.

Co-arrays of derived type can be combined with the object-oriented features of Fortran 95 (Akin 2003, Norton, Decyk and Szymanski 1997) and Fortran 2003 (Cohen 2004, Reid 2003) to emulate object-oriented class libraries. Numrich used this technique to build an object-based parallel numerical library (Numrich 2005*a*, Numrich 2005*b*, Numrich 2006). This library defines distributed data structures as derived types such as `BlockR8Matrix` for a blocked matrix with double precision data. Constructors create these distributed objects based on information contained in another structure called a `VectorMap`, and they allocate memory space for the correct amount of data for each image. The library contains procedures that perform various operations on these data structures including the usual linear algebra operations like matrix multiplication, matrix transpose, and LU decomposition.

Figure 7.7 contains a code sample that shows how to use this object-based library for solving a system of linear equations. The second line of the program gives access to the library through the module `CafLib`. The next four lines of code declare some co-arrays: the block matrix `A` holds the distributed matrix, and the block vector `X` holds the right-hand-side vector, which will be replaced by the solution vector. The pivot vector `pivot` holds pivot information associated with

Figure 7.7: Program for solving a system of linear equations using CafLib.

```
program LU
  use CafLib                                    ! Access object-based library
  type(BlockR8Matrix)        :: A[*]            ! Distributed matrix
  type(BlockR8Vector)        :: X[*]            ! Distributed vector
  type(R8PivotVector)        :: pivot[*]        ! Distributed permutation
  integer                    :: k[*],n[*]       ! Block size and matrix size
  type(DirectAccessDiskFile) :: fileA           ! Description of file for A
  type(DirectAccessDiskFile) :: fileX           ! Description of file for X
  character,parameter        :: myA='myA'       ! File name for A
  character,parameter        :: myX='myX'       ! File name for X
  integer,parameter          :: unit=888        ! Unit number
  integer                    :: Iam             ! Index of executing image
  integer                    :: p,q,coDims(2)   ! Co-dimensions
  Iam = this_image()
  coDims(:) = factorNumImages(2)         ! Factor the image-index space
  p = coDims(1); q = coDims(2)           ! Co-dimensions; p*q<=num_images()
  if(Iam == 1) then
    open(unit=unit,file="input"); read(unit,*) n,k; close(unit)
  end if
  SYNC ALL                               ! Barrier
  n = n[1]; k = k[1]
  call newBlockMatrix(A,n,n,k,k,p,q)     ! Construct block matrix
  call newBlockVector(X,n,k,p*q)         ! Construct block vector
  call newPivotVector(A,pivot)           ! Construct pivot vector:     collective
  call newDiskFile(fileA,unit,myA,n)     ! Describe I/O for matrix A
  call newDiskFile(fileX,unit,myX,n)     ! Describe I/O for vector X
  call readBlockMatrix(A,fileA)          ! Read matrix A from a file:  collective
  call readBlockVector(X,fileX)          ! Read vector X from a file:  collective
  call luDecomposition(A,pivot)          ! Perform LU decomposition:   collective
  call solve(A,X,pivot)                  ! Solve the system:           collective
    :                                    ! Make use of the solution
  call writeBlockVector(X,fileX)         ! Write vector X to a file:   collective
  call deletePivotVector(pivot)          ! Remove block matrix:        collective
  call deleteBlockMatrix(A)              ! Remove block vector
  call deleteBlockVector(X)              ! Remove block vector
  call deleteDiskFile(fileA)             ! Remove disk file A
  call deleteDiskFile(fileX)             ! Remove disk file X
end program LU
```

the matrix `A` used during the decomposition and solution. The integers `n` and `k` contain the global matrix size and the local block size. They are declared as co-arrays so that all images can obtain their values from the first image, which reads them from an input file.

The first executable statement determines the image number for each image. The second executable statement invokes the function `factorNumImages`, which returns an array of two integers that factor the image-index space into a two-dimensional grid. These indices `p` and `q` are used in the constructors at run-time to determine how objects are distributed across images.

The `if` statement controls execution so that image 1 opens the input file, reads the problem size and local block size, and then closes the file. The other images wait at their barrier `sync all` until the values have been defined in the co-array variables `n` and `k`, and they all copy the values from image 1 after passing the barrier.

Each image independently invokes the constructor `newBlockMatrix` to create a distributed matrix of global size `n`×`n`. Inside the library, the co-array is declared with co-dimensions `A[p,*]`. The constructor decomposes both rows and columns into a cyclic-wrapped distribution (Dongarra and Walker 1995) with local blocks of size `k`×`k` held as allocatable array components. The constructor allocates memory for these blocks and initializes them to zero. Next, each image invokes the constructor `newBlockVector` to create a distributed vector of global size `n` with a cyclic-wrapped distribution of local blocks of size `k` again held as allocatable array components. The constructor allocates memory for these blocks and initializes them to zero. Inside the library, the co-array is declared with co-dimension `X[*]`.

Each image invokes the constructor `newPivotVector` to create a pivot vector associated with the matrix `A`. It is a collective subroutine with internal synchronization. The pivot vector is used internally by the library, and all its components are private. Its co-dimensions inside the library match those of the matrix `A`. The library function `getPivotVector` returns an array containing the usual pivot vector, like our array `ipiv` in Figure 6.5, if the programmer needs it.

The library contains I/O routines for matrix and vector objects. To use them, each image invokes the constructor `newDiskFile` to create objects of type `DirectAccessDiskFile` that enable them to read the matrix `A` and the vector `X` from direct access files. Each image invokes the collective subroutines `readBlockMatrix` and `readBlockVector` that read the data from the files. They then invoke the collective subroutine `luDecomposition` that performs the LU decomposition followed by the collective subroutine `solve` that solves the system. They all invoke the collective I/O subroutine `writeBlockVector` that saves the solution vector to a file. After using the solution, each image comes to the second barrier `sync all` where they wait for all images to arrive before deleting the data structures.

Reference (Numrich 2005*a*) describes the performance of this code on the Cray-T3E and reference (Numrich 2007) describes the performance on the Cray-X1E.

8 Collective subroutines

A **collective subroutine** is a member of a new category of intrinsic subroutines that perform operations across co-dimensions in a way similar to intrinsic procedures that perform operations across normal dimensions.

```
co_all    (mask,result[,team])      True if all values are true
co_any    (mask,result[,team])      True if any value is true
```

```
co_count  (mask,result[,team])      Numbers of true elements
co_maxloc (co_array,result[,team])  Image indices of maximum values
co_maxval (co_array,result[,team])  Maximum values
co_minloc (co_array,result[,team])  Image indices of minimum values
co_minval (co_array,result[,team])  Minimum values
co_product(co_array,result[,team])  Products of elements
co_sum    (co_array,result[,team])  Sums of elements
```

Each subroutine has a leading `intent(in)` co-array argument with the same shape on every image of the team, an `intent(out)` result argument with the same shape as the first argument, and an optional scalar `intent(in)` argument of type `image_team`. If the optional argument is absent, the team consists of all images. The same statement must be used to call the collective on all images of the team, and each call involves synchronization of the images of the team. Upon completion, each image of the team receives the same `result`, each element of which is calculated from the values of the corresponding elements on all the images of the team.

The following invocation, for example,

```
real :: x(n)[*],y(n)
call co_product(x,y)
```

returns to local array `y(:)` on each image the products,

$$\mathtt{y(i)} = \prod_{\mathtt{p}} \mathtt{x(i)[p]} \ . \tag{8.1}$$

Roundoff effects may cause the result of `co_product`, as well as the result of `co_sum`, to vary between images.

An important case is for a scalar co-array. For example, the code,

```
real :: x(n), y(n), prod
real :: local_prod[*]
   :
local_prod = dot_product(x(1:n),y(1:n))
call co_sum(local_prod,prod)
```

computes the inner-product of two rank-one arrays `x` and `y`. Each image computes its own local inner-product using the normal `dot_product` intrinsic function. No team is specified in the call of `co_sum`, so the team is taken to be all the images. There is an implied synchronization so the programmer need not supply it explicitly. The collective procedure returns the sum over all the images to them all in the result variable `prod`.

9 Summary

The co-array programming model is the first official parallel extension to the Fortran language. We designed co-arrays, from the beginning, to be a natural extension that makes sense to a Fortran programmer. Arrays are the rock bottom essence of the language for expressing numerical algorithms in a very natural way. Co-arrays provide an equally natural way for expressing parallel numerical algorithms.

The underlying philosophy of our design is to make the smallest number of changes to the language required to obtain a robust and efficient parallel language without requiring the programmer to learn very many new rules. Co-array syntax looks and feels like normal Fortran syntax and, for the most part, the rules that apply to co-dimensions are the same as the rules that apply to normal dimensions.

Of equal importance in our design, the compiler is required to implement very little new technology. Because the programmer is responsible for data distribution, for remote data communication, and for explicit synchronization, the compiler is free to concentrate on local code optimization using all the compiler technology developed over the past decades. Only where it sees explicit co-array syntax does it need to generate new code to use a specific communication protocol for a particular machine. Square bracket syntax in a statement acts as an explicit flag to the compiler, as well as to the programmer, that remote memory operations are taking place. The compiler optimizes such statements locally using the same optimization techniques it uses to optimize local memory operations.

In this paper, we have explained the more important aspects of co-arrays for parallel programming as they have been specified for the next Fortran standard. An image is a replication of the program, and the number of images is fixed. Each image behaves as a uni-processor code except that some data objects are declared as co-arrays, and each image has access to co-arrays on other images. That access is provided in a very natural way though an additional set of subscripts.

There is no attempt to treat the set of co-arrays with a given name as a huge shared array because that would place a huge burden on the system to provide memory coherence. Instead, the programmer provides explicit synchronizations, and the compiler is free, for code between synchronizations, to use all its normal optimization techniques as if only one image were present.

Co-arrays have the same shape on every image. For irregular data, a co-array may be of a derived type that has allocatable or pointer components that may vary in size between images. The allocated arrays or pointer targets are always on the same image so that implementation is straightforward. We have outlined how to use such data structures to design a library for numerical linear algebra that emulates an object-oriented class library.

We have concentrated on the more important mechanisms for synchronization, illustrated how to form and use subsets of images as teams, described the new category of collective subroutines, and demonstrated the principal mechanism for parallel I/O using direct-access files. Our code for parallel LU factorization shows the simplicity of using the co-array model and how to combine it with well established numerical libraries that have been used as the workhorses for performance for many decades.

10 Acknowledgements

We express our special thanks to Bill Long of Cray Inc for his help with many of the detailed changes made since the 1998 report and for his advocacy of co-arrays in the US Fortran Committee J3. The United States Department of Energy supported this research in part through Grant No. DE-FC02-01ER25505 from the Office of Science.

References

J. E. Akin. *Object-oriented programming via Fortran 90/95*. Cambridge University Press, 2003.

E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Phialadelphia, 3rd edn, 1999.

Cristian Coarfa, Yuri Dotsenko, and John Mellor-Crummey. Experiences with Sweep3D implementations in Co-Array Fortran. *in* 'Proceedings of the Los Alamos Computer Science Institute 5th Annual Symposium, Santa Fe, NM, USA', 2004.

Cristian Coarfa, Yuri Dotsenko, Jason Lee Eckhardt, and John Mellor-Crummey. Co-array Fortran Performance and Potential: An NPB Experimental Study. *The 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), College Station, Texas, October 2003*, October 2003.

Malcolm Cohen. Fortran 2003: Into the Future. *Dr. Dobb's Journal*, pp. 50–56, July 2004.

Jack J. Dongarra and David W. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, **37**(2), 151–180, June 1995.

Jack J. Dongarra, Robert A. van de Geijn, and David W. Walker. Scalability issues affecting the design of a dense linear algebra library. *Journal of Parallel and Distributed Computing*, **22**, 523–537, 1994.

Yuri Dotsenko, Cristian Coarfa, and John Mellor-Crummey. A multi-platform co-array fortran compiler. *in* 'Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT 2004, Antibes Juan-les-Pins, France)', 2004*a*.

Yuri Dotsenko, Cristian Coarfa, John Mellor-Crummey, and D. Chavarría-Miranda. Experiences with Co-Array Fortran on hardware shared memory platforms. *in* 'Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2004, West Lafayette, IN, USA)', 2004*b*.

ISO/IEC Fortran Standards Committee, ISO/IEC JTC 1/SC22/WG5. http://www.nag.co.uk/sc22wg5/.

Jarek Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. *in* 'Lecture Notes in Computer Science', Vol. 1586. Springer-Verlag, 1999.

C.D. Norton, V.K. Decyk, and B.K. Szymanski. High performance object-oriented scientific programming in Fortran 90. *in* 'Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing'. SIAM Activity Group on Supercomputing, Society for Industrial and Applied Mathematics, March 1997. CD ROM format.

Robert W. Numrich. F^{--}: A Parallel Extension to Cray Fortran. *Scientific Programming*, **6**(3), 275–284, Fall 1997.

Robert W. Numrich. A Parallel Numerical Library for Co-Array Fortran. *in* 'Parallel Processing and Applied Mathematics: Proceedings of the Sixth International Conference on Parallel Processing and Applied Mathematics (PPAM05)', pp. 960–969, Poznan, Poland, September 11-14 2005*a*. Springer Lecture Notes in Computer Science, LNCS 3911.

Robert W. Numrich. Parallel numerical algorithms based on tensor notation and Co-Array Fortran syntax. *Parallel Computing*, **31**, 588–607, 2005*b*.

Robert W. Numrich. CafLib Users' Manual: Release 1.2. Available from the author, 2006.

Robert W. Numrich. A Note on Scaling the Linpack Benchmark. in press, Journal of Parallel and Distributed Computing, 2007.

Robert W. Numrich and John Reid. Co-Array Fortran for parallel programming. *ACM Fortran Forum*, **17**(2), 1–31, 1998.

Robert W. Numrich and John Reid. Co-arrays in the next Fortran Standard. *ACM Fortran Forum*, **24**(2), 2–24, 2005.

Robert W. Numrich and Jon L. Steidel. F^{--}: A Simple Parallel Extension to Fortran 90. *SIAM News*, **30**(7), September 1997.

Robert W. Numrich, John Reid, and Kieun Kim. Writing a multigrid solver using Co-Array Fortran. *in* B. K. gström, J. Dongarra, E. Elmroth and J. Waśniewski, eds, 'Applied Parallel Computing: Large Scale Scientific and Industrial Problems', pp. 390–399. 4th International Workshop, PARA98, Umeå, Sweden, June 1998, Springer, 1998*a*. Lecture Notes in Computer Science 1541.

Robert W. Numrich, Jon L. Steidel, Brian H. Johnson, Benoit Dupont de Dinechin, Gary Elsesser, Greg Fischer, and Tom MacDonald. Definition of the F^{--} Extension to Fortran 90. *in* 'Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computers', pp. 292–306. Lectures on Computer Science Series, Number 1366, Springer-Verlag, 1998*b*.

John Reid. The future of Fortran. *Computing in Science and Engineering*, **5**(3), 59–67, July/August 2003.

John Reid. Co-array Fortran for parallel programming. ISO/IEC/JTC1/SC22/WG5-N1626, requirement UK-001 (ftp://ftp.nag.co.uk/sc22wg5/N1601-N1650/N1626.txt), 2005*a*.

John Reid. Revision of Requirement UK-001. ISO/IEC/JTC1/SC22/WG5-N1639 (ftp://ftp.nag.co.uk/sc22wg5/N1601-N1650/N1639.txt), 2005*b*.

John Reid. Co-arrays in the new Fortran Standard. ISO/IEC/JTC1/SC22/WG5-N1669, (ftp://ftp.nag.co.uk/sc22wg5/N1601-N1650/N1669.pdf), 2007.

US Fortran Standards Committee J3, a technical subcommittee of the InterNational Committee for Information Technology Standards (INCITS). http://www.j3-fortran.org/.