RAL-TR-1998-060

Co-Array Fortran for parallel programming[1]

by

R. W. Numrich[2] and J. K. Reid[3]

Abstract

Co-Array Fortran, formerly known as F−−, is a small extension of Fortran 95 for parallel processing. A Co-Array Fortran program is interpreted as if it were replicated a number of times and all copies were executed asynchronously. Each copy has its own set of data objects and is termed an image. The array syntax of Fortran 95 is extended with additional trailing subscripts in square brackets to give a clear and straightforward representation of any access to data that is spread across images.

References without square brackets are to local data, so code that can run independently is uncluttered. Only where there are square brackets, or where there is a procedure call and the procedure contains square brackets, is communication between images involved.

There are intrinsic procedures to synchronize images, return the number of images, and return the index of the current image.

We introduce the extension; give examples to illustrate how clear, powerful, and flexible it can be; and provide a technical definition.

Categories and subject descriptors: D.3 [PROGRAMMING LANGUAGES].
General Terms: Parallel programming.
Additional Key Words and Phrases: Fortran.

Department for Computation and Information,
Rutherford Appleton Laboratory,
Oxon OX11 0QX, UK

August 1998.

---

[1] Available by anonymous ftp from `matisa.cc.rl.ac.uk` in directory `pub/reports` in the file `nrRAL98060.ps.gz`

[2] Silicon Graphics, Inc., 655 Lone Oak Drive, Eagan, MN 55121, USA. Email: `rwn@cray.com`

[3] Email: `jkr@rl.ac.uk`

CONTENTS

1 Introduction

We designed Co-Array Fortran to answer the question 'What is the smallest change required to convert Fortran 95 into a robust, efficient parallel language?'. Our answer is a simple syntactic extension to Fortran 95. It looks and feels like Fortran and requires Fortran programmers to learn only a few new rules.

The few new rules are related to two fundamental issues that any parallel programming model must resolve, work distribution and data distribution. Some of the complications encountered with other parallel models, such as HPF (Koebel, Loveman, Schrieber, Steele, and Zosel 1994), CRAFT (Pase, MacDonald, and Meltzer 1994) or OpenMP (1997) result from the intermixing of these two issues. They are different and Co-Array Fortran keeps them separate.

First, consider work distribution. Co-Array Fortran adopts the Single-Program-Multiple-Data (SPMD) programming model. A single program is replicated a fixed number of times, each replication having its own set of data objects. Such a model is new to Fortran, which assumes a single program executing alone with a single set of data objects. Each replication of the program is called an **image.** Each image executes asynchronously and the normal rules of Fortran apply, so the execution path may differ from image to image. The programmer determines the actual path for the image with the help of a unique image index by using normal Fortran control constructs and by explicit synchronizations. For code between synchronizations, the compiler is free to use all its normal optimization techniques, as if only one image is present.

Second, consider data distribution. The co-array extension to the language allows the programmer to express data distribution by specifying the relationship among memory images in a syntax very much like normal Fortran array syntax. We add one new object to the language called a co-array. For example, the statement

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

declares that each image has two real arrays of size `n`. If the statement:

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

is executed on all images and if `q` has the same value on each image, the effect is that each image copies the array `y` from image `q` and makes a local copy in array `x`.

Array indices in parentheses follow the normal Fortran rules within one memory image. Indices in square brackets provide an equally convenient notation for accessing objects across images and follow similar rules. Bounds in square brackets in co-array declarations follow the rules of assumed-size arrays since co-arrays are always spread over all the images.

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 reference to the object in the local memory of the executing image. Since most references to data objects in a parallel code should be to the local part, 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.

If different sizes are required on different images, we may declare a co-array of a derived type with a component that is a pointer array. The pointer component is allocated on each image to have the desired size for that image (or not allocated at all, if it is not needed on the image). It is straightforward to access such data on another image, for example,

```
x(:) = a[p]%ptr(:)
```

In words, this statement means 'Go to image `p`, obtain the the pointer component of variable `a`, read from the corresponding target, and copy the data to the local array `x`'. The square bracket is associated with the variable

`a`, not with its components. Data manipulation of this kind is handled awkwardly, if at all, in other programming models. Its natural expression in co-array syntax places power and flexibility with the programmer, where they belong. This technique may be the key to such difficult problems as adaptive mesh refinement, which must be solved to make parallel processing on large machines successful.

Co-Array Fortran was formerly known as F−−, pronounced eff-minus-minus. The name was meant to imply a small addition to the language, but was often misinterpreted. It evolved from a simple programming model for the CRAY-T3D where it was first seen as a back-up plan while the real programming models, CRAFT, HPF, and MPI, were developed. This embryonic form of the extension was described only in internal Technical Reports at Cray Research (Numrich 1991, Numrich 1994a, Numrich 1994b). In some sense, Co-Array Fortran can be thought of as syntax for the one-sided get/put model as represented, for example, in the SHMEM Library on the CRAY-T3D/E and on the CRAY Origin 2000. This model has become the preferred model for writing one-sided message-passing code for those machines (Numrich, Springer, and Peterson 1994; Sawdey, O'Keefe, Bleck, and Numrich 1995). But since co-array syntax is incorporated into the language, it is more flexible and more efficient than any library implementation can ever be.

The first informal definition of F−− (Numrich 1997) was restricted to the Fortran 77 language and used a different syntax to represent co-arrays. To remove the limitations of the Fortran 77 language, we extended the F−− idea to the Fortran 90 language (Numrich and Steidel 1997a; Numrich and Steidel 1997b; Numrich, Steidel, Johnson, de Dinechin, Elsesser, Fischer, and MacDonald 1997). In these papers, the programming model was defined more precisely and an attempt was made to divorce the model from the subconscious association of the syntax with physical processors and to relate it more to the logical decomposition of the underlying physical or numerical problem. In addition, these papers started to work through some of the complexities and idiosyncrasies of the Fortran 90 language as they relate to co-array syntax. This current paper is the culmination of that effort.

In the meantime, portions of Co-Array Fortran have been incorporated into the Cray Fortran 90 compiler and various applications have been converted to the syntax (see, for example, Numrich, Reid, and Kim 1998).

In the next section, we illustrate the power of the language with some simple examples, introducing syntax and semantics as we go, without attempting to be complete. Section 3 contains a complete technical specification, Section 4 contains improved versions of the codes of Section 2, and in Section 5 we make some comparisons with other languages. We conclude with a summary of the features of Co-Array Fortran in Section 6. Appendices 1 and 2 explain possible extensions and Appendix 3 shows how the intrinsic `sync_memory` permits the intrinsics `sync_team` and `sync_all` to be constructed in Co-Array Fortran.

In Section 3, paragraphs labeled as notes are non-normative, that is, they they have no effect on the definition of the language. They are there to help the reader to understand a feature or to explain the reasoning behind it.

2 Simple examples

In this section, we consider some simple examples. By these examples, we do not mean to imply that we expect every programmer who uses co-array syntax to reinvent all the basic communication primitives. The examples are intended to illustrate how they might be written with the idea of including them in a library for general use and how easy it is to write application codes requiring more complicated communication.

2.1 Finite differencing on a rectangular grid

For our first example, suppose we have a rectangular grid with periodic boundary conditions; at each point, we want to sum the values at neighbouring points and subtract four times the value at the point (5-point approximation to the Laplacian operator). We suppose that the data is distributed as the co-array `u(1:nrow)[1:ncol]`, in both the local dimension and the co-dimension. If `ncol` is equal to the number of images, the following procedure is one way to perform the calculation.

```
subroutine laplace (nrow,ncol,u)
   integer, intent(in)  ::  nrow, ncol
   real, intent(inout)  ::  u(nrow)[*]
   real                 ::  new_u(nrow)
   integer              ::  i, me, left, right
   new_u(1) = u(nrow) + u(2)
   new_u(nrow) = u(1) + u(nrow-1)
   new_u(2:nrow-1) = u(1:nrow-2) + u(3:nrow)
   me = this_image(u) ! Returns the co-subscript within u
                      ! that refers to the current image
   left = me-1; if (me == 1) left = ncol
   right = me + 1; if (me == ncol) right = 1
   call sync_all( (/left,right/) )  ! Wait if left and right
                                    ! have not already reached here
   new_u(1:nrow)=new_u(1:nrow)+u(1:nrow)[left]+u(1:nrow)[right]
   call sync_all( (/left,right/) )
   u(1:nrow) = new_u(1:nrow) - 4.0*u(1:nrow)
end subroutine laplace
```

In the first part of the procedure, we add together neighbouring values of the array along the local dimension, being careful to enforce the periodic boundary conditions. We place the result in a temporary array because we cannot overwrite the original values until the averaging is complete. We obtain the co-subscript of the invoking image from the intrinsic function `this_image(u)` and then enforce the periodic boundary conditions across the co-dimension in the same way that we did for the local dimension. The Co-Array Fortran execution model is asynchronous SPMD, not synchronous SIMD, so we must synchronize explicitly with the intrinsic procedure `sync_all` to make sure the values of the array on neighbouring images are ready before using them. After adding the values from these images into the local temporary array, we synchronize a second time to make sure the neighbouring images have obtained the original local values before altering them. After the second synchronization, each image replaces the data in its local array with the averaged values corresponding to the finite difference approximation for the Laplacian operator and returns.

2.2 Data redistribution

Our second example comes from the application of fast Fourier transforms. Suppose we have a 3-dimensional co-array with one dimension spread across images: `a(1:kx,1:ky)[1:kz]` and need to copy data into it from a co-array with a different dimension spread across images: `b(1:ky,1:kz)[1:kx]`. We assume that the arrays are declared thus:

```
real :: a(kx,ky)[*], b(ky,kz)[*]
```

Of course, the number of images must be at least `max(kx,kz)`. If the number of images is exactly `kz`, the following Co-Array Fortran code does what is needed:

```
iz = this_image(a)
do ix = 1, kx
   do iy = 1, ky
      a(ix,iy) = b(iy,iz)[ix]
   end do
end do
```

The outer do construct is executed on each image and it ranges over all the images from which data is needed. The inner do construct is a clear representation of the data transfer; it can also be written as the array statement

```
a(ix,:) = b(:,iz)[ix]
```

In the cycle with `ix=iz`, only local data transfer takes place. This is permitted, but the statement

```
a(iz,:) = b(:,iz)
```

might be more efficient in execution than the statement

```
a(iz,:) = b(:,iz)[iz]
```

If the number of images is greater than `kz`, the code will be erroneous on the additional images because of an out-of-range subscript. We therefore need to change the code to:

```
iz = this_image(a)
if (iz<=kz) then
   do ix = 1, kx
      a(ix,:) = b(:,iz)[ix]
   end do
end if
```

2.3 Maximum value of a co-array

Our next example finds the maximum value of a co-array and broadcasts the result to all the images. On each image, the code begins by finding the local maximum. Synchronization is then needed to ensure that all images have performed this task before the maximum of the results is computed. We then use the first image to gather the local maxima, find the global maximum and scatter the result to the other images. Another synchronization is needed so that none of the images leave the procedure with the result in its memory.

```
subroutine greatest(a,great) ! Find maximum value of a(:)[*]
   real, intent(in)  :: a(:)[*]
   real, intent(out) :: great[*]
   real :: work(num_images()) ! Local work array
   great = maxval(a(:))
   call sync_all ! Wait for all other images to reach here
   if(this_image(great)==1)then
      work(:) = great[:]    ! Gather local maxima
      great[:]=maxval(work) ! Broadcast the global maximum
   end if
   call sync_all
end subroutine greatest
```

The work array is needed only on the first image, so storage will be wasted on the other images. If this is important, we may use an allocatable array that is allocated only on the first image (see the enhanced version in Section 4.3).

Note that we have used array sections with square brackets in an intrinsic assignment. These may also be used

in intrinsic operations. Although the processor does not need to do it this way, the effect must be as if the data were collected into a temporary array on the current image and the operation performed there. For simplicity of implementation, we have restricted this feature to the intrinsic operations and intrinsic assignment. However, round brackets can always be employed to create an expression and have the effect of a copy being made on the current image. For example, a possible implementation of the example of this section is

```
if(this_image(a)==1)then
   great[:]=maxval( (a(:)[:]) )
end if
```

but this would probably be slow since all the data has to be copied to image 1 and all the work is done by image 1.

2.4 Finite-element example

We now consider a finite-element example. Suppose each image works with a set of elements and their associated nodes, which means that some nodes appear on more than one image. We treat one of these nodes as the principal and the rest as 'ghosts'. For each image, we store pairs of indices `(prin(i),ghost(i))` of principals on the image and corresponding ghosts on other images. We group them by the image indices of the ghosts.

In the assembly step for a vector **x**, we first add contributions from the elements in independent calculations on all the images. Once this is done, for each principal and its ghosts, we need to add all the contributions and place the result back in all of them. The following is suitable:

```
subroutine assemble(start,prin,ghost,neib,x)
! Accumulate values at nodes with ghosts on other images
    integer, intent(in) :: start(:), prin(:), ghost(:), neib(:)
! Node prin(i) is the principal for ghost node ghost(i) on image neib(p),
!     i = start(p), ... start(p+1)-1, p=1,2,...,size(neib).
   real, intent(inout) :: x(:)[*]
   integer k1,k2,p
   call sync_all (neib)
   do p = 1,size(neib) ! Add in contributions from the ghosts
      k1 = start(p); k2 = start(p+1)-1
      x(prin(k1:k2)) = x(prin(k1:k2)) + x(ghost(k1:k2))[neib(p)]
   end do
   call sync_all (neib)
   do p = 1,size(neib) ! Update the ghosts
      k1 = start(p); k2 = start(p+1)-1
      x(ghost(k1:k2))[neib(p)] = x(prin(k1:k2))
   end do
   call sync_all
end subroutine assemble
```

2.5 Summing over the co-dimension of a co-array

We now consider the problem of summing over the co-dimension of a co-array and broadcasting the result to all images. A possible implementation is as follows, using the first image to do all the work:

```
   subroutine sum_reduce(n,x)
      integer, intent(in) :: n
      real, intent(inout) :: x(n)[*]
      integer p
      call sync_all
      ! Replace x by the result of summing over the co-dimension
      if ( this_image(x)==1 ) then
         do p=2,num_images()
            x(:) = x(:) + x(:)[p]
         end do
         do p=2,num_images()
            x(:)[p] = x(:)
         end do
      end if
      call sync_all
   end subroutine sum_reduce
```

2.6 Grouping the images into teams

If most of a calculation naturally breaks into two independent parts, we can employ two sets of images independently. To avoid wasting storage, we use an array of derived type with pointer components that are allocated only on those images for which they are needed. For example, the following module is intended for a hydrodynamics calculation on the first half of the images.

```
module hydro
   type hydro_type
      integer :: nx,ny,nz
      real, pointer :: x(:),y(:),z(:)
   end type hydro_type
   type(hydro_type) :: hyd[*]
contains
   subroutine setup_hydro
      :
     allocate ( hyd%x(lx), hyd%y(ly), hyd%z(lz) )
     call sync_team( (/(i,i=1,num_images()/2)/) )
      :
   end subroutine setup_hydro
    :
end module hydro
```

The subroutine `setup_hydro` is called only on the images 1, 2, ..., `num_images()/2`. It allocates storage and performs other initializations. The corresponding main program might have the form:

```
program main
   use hydro
   use radiation
   real :: residual, threshold = 1.0e-6
   if(this_image()<=num_images()/2) then
      call setup_hydro ! Establish hydrodynamics data
   else
      call setup_radiation ! Establish radiation data
   end if
   call sync_all
   do ! Iterate
      if(this_image()<=num_images()/2) then
         call hydro
      else
         call radiation
      end if
      call sync_all
        : ! Code that accesses data from both modules and calculates residual
      if(residual<threshold)exit
   end do
end program main
```

2.7 Writing a tiled array to a direct-access file

Suppose an array is distributed over the images so that each image holds a sub-array of the same size, a tile, and that a single element halo is added around each tile. The simplest way to write out the file is:

```
real :: a(0:ml+1,0:nl+1)[mp,*]  ! num_images()==mp*np
:
inquire(iolength=la) a(1:ml,1:nl)
open(unit=11,file='fort.11',status='new',action='write',&
     form='unformatted',access='direct',recl=la, &
     team=(/ (i, i=1,num_images()) /)
write(unit=11,rec=this_image()) a(1:ml,1:nl)
```

The keyword team specifies which images are connected to the unit. All the images are writing to distinct records in the same file, so there is no need for synchronization.

3 Technical specification

3.1 Program images

A Co-Array Fortran program executes as if it were replicated a number of times, the number of replications remaining fixed during execution of the program. Each copy is called an **image** and each image executes asynchronously. A particular implementation of Co-Array Fortran may permit the number of images to be chosen at compile time, at link time, or at execute time. The number of images may be the same as the number of physical processors, or it may be more, or it may be less. The programmer may retrieve the number of images at run time by invoking the intrinsic function `num_images()`. Images are indexed starting from one and the programmer may retrieve the index of the invoking image through the intrinsic function `this_image()`. The programmer controls the execution sequence in each image through explicit use of Fortran 95 control constructs and through explicit use of an intrinsic synchronization procedures.

3.2 Specifying data objects

Each image has its own set of data objects, all of which may be accessed in the normal Fortran way. Some objects are declared with **co-dimensions** in square brackets immediately following dimensions in parentheses or in place of them, for example:

```
real, dimension(20)[20,*]  :: a
real  :: c[*], d[*]
character :: b(20)[20,0:*]
integer :: ib(10)[*]
type(interval) :: s
dimension :: s[20,*]
```

Unless the array is allocatable (Section 3.6), the form for the dimensions in square brackets is the same as that for the dimensions in parentheses for an assumed-size array. The set of objects on all the images is itself an array, called a **co-array,** which can be addressed with array syntax using subscripts in square brackets following any subscripts in parentheses (round brackets), for example:

```
a(5)[3,7] = ib(5)[3]
d[3] = c
a(:)[2,3] = c[1]
```

We call any object whose designator includes square brackets a **co-array subobject;** it may be a **co-array element**, a **co-array section**, or a **co-array structure component**. The subscripts in square brackets are mapped to images in the same way as Fortran array subscripts in parentheses are mapped to memory locations in a Fortran 95 program. The subscripts within an array that correspond to data for the current image are available from the intrinsic `this_image` with the co-array name as its argument.

Note: On a shared-memory machine, we expect a co-array to be implemented as if it were an array of higher rank. The implementation would need to support the declaration of arrays of rank up to 14. On a distributed-memory machine with one physical processor for each image, a co-array may be stored from the same memory address in each physical processor. On any machine, a co-array may be implemented in such a way that each image can calculate the memory address of an element on any other image.

The **rank**, **extents**, **size,** and **shape** of a co-array or co-array subobject are given as for Fortran 95 except that we include both the data in parentheses and the data in square brackets. The **local rank**, **local extents**, **local size**, and **local shape** are given by ignoring the data in square brackets. The **co-rank**, **co-extents**, **co-size**, and **co-shape** are given from the data in square brackets. For example, given the co-array declared thus

```
real, dimension(10,20)[20,5,*]  :: a
```

`a(:,:)[:,:,1:15]` has rank 5, local rank 2, co-rank 3, shape (/10,20,20,5,15/), local shape (/10,20/), and co-shape (/20,5,15/).

The co-size of a co-array is always equal to the number of images. If the co-rank is one, the co-array has a co-extent equal to the number of images and it has co-shape `(/num_images()/)`. If the co-rank is greater than one, the co-array has no final extent, no final upper bound, and no co-shape (and hence no shape).

Note: We considered defining the final extent when the co-rank is greater than one as the number of images divided by the product of the other extents, truncating towards zero. We reject this, since it means, for example, that `a(:,:)[:,:,:]` would not always refer to the whole declared co-array.

The local rank and the co-rank are each limited to seven. The syntax automatically ensures that these are the same on all images. The rank of a co-array subobject (sum of local rank and co-rank) must not exceed seven.

Note: The reason for the limit of seven on the rank of a co-array subobject is that we expect early implementations to make a temporary local copy of the array and then rely on ordinary Fortran 95 mechanisms.

For a co-array subobject, square brackets may never precede parentheses.

Note: For clarity, we recommend that subscripts in parentheses are employed whenever the parent has nonzero local rank. For example, `a[:]` is not as clear as `a(:)[:]`.

A co-array must have the same bounds (and hence the same extents) on all images. For example, the subroutine

```
subroutine solve(n,a,b)
integer :: n
real :: a(n)[*], b(n)
```

must not be called on one image with `n` having the value 1000 and on another with `n` having the value 1001.

A co-array may be allocatable:

```
subroutine solve(n,a,b)
integer :: n
real :: a(n)[*], b(n)
real,allocatable :: work(:)[:]
```

Allocatable arrays are discussed in Section 3.6.

There is no mechanism for assumed-co-shape arrays (but see Appendix 1, which describes a possible extension). A co-array is not permitted to be a pointer (but see Appendix 2, which describes another possible extension). Automatic co-arrays are not permitted; for example, the co-array `work` in the above code fragment is not permitted to be declared thus

```
subroutine solve(n,a,b)
integer :: n
real :: a(n)[*], b(n)
real :: work(n)[*]  ! Not permitted
```

Note: Were automatic co-arrays permitted, for example, in a future revision of the language, they would pose problems to implementations over consistent memory addressing among images. It would probably be necessary to require image synchronization, both before and after memory is allocated on entry and both before and after memory is deallocated on return.

A co-array is not permitted to be a constant.

Note: This restriction is not necessary, but the feature would be useless since each image would hold exactly

the same value. We see no point in insisting that vendors implement such a feature.

A DATA statement initializes only local data. Therefore, co-array subobjects are not permitted in DATA statements. For example:

```
real :: a(10)[*]
data a(1)    /0.0/ ! Permitted
data a(1)[2] /0.0/ ! Not permitted
```

Unless it is allocatable or a dummy argument, a co-array always has the SAVE attribute.

The image indices of a co-array always form a sequence, without any gaps, commencing at one. This is true for any lower bounds. For example, for the array declared as

```
real :: a(10,20)[20,0:5,*]
```

`a(:,:)[1,0,1]` refers to the rank-two array `a(:,:)` in image one.

Note: If a large array is needed on a subset of images, it is wasteful of memory to specify it directly as a co-array. Instead, it should be specified as a pointer component of a co-array and allocated only on the images on which it is needed (we expect to use an allocatable component in Fortran 2000).

Co-arrays may be of derived type but components of derived types are not permitted to be co-arrays.

Note: Were we to allow co-array components, we would be confronted with references such as `z[p]%x[q]`. A logical way to read such an expression would be: go to image `p` and find component `x` on image `q`. This is logically equivalent to `z[q]%x`.

3.3 Accessing data objects

Each object exists on every image, whether or not it is a co-array. In an expression, a reference without square brackets is always a reference to the object on the invoking image. For example, `size(b)` for the co-array `b` declared at the start of Section 3.2 returns its local size, which is 20.

The subscript order value of the co-subscript list must never exceed the number of images. For example, if there are 16 images and the the co-array `a` is declared thus

```
real :: a(10)[5,*]
```

`a(:)[1,4]` is valid since it has co-subscript order value 16, but `a(:)[2,4]` is invalid.

Two arrays conform if they have the same shape. Co-array subobjects may be used in intrinsic operations and assignments in the usual way, for example,

```
b(:,1:m) = a[:,1:m]*c(:)[1:m] ! All have rank two.
b(j,:)   = a[:,k]             ! Both have rank one.
c[1:p:3] = d(1:p:3)[2]        ! Both have rank one.
```

Square brackets attached to objects in an expression or an assignment alert the reader to communication between images. Unless square brackets appear explicitly, all expressions and assignments refer to the invoking image. Communication may take place, however, within a procedure that is referenced, which might be a defined operation or assignment.

The rank of the result of an intrinsic operation is derived from the ranks of its operands by the usual rules, disregarding the distinction between local rank and co-rank. The local rank of the result is equal to the rank. The co-rank is zero. Similarly, a parenthesized co-array subobject has co-rank zero. For example `2.0*d(1:p:3)[2]` and `(d(1:p:3)[2])` each have rank 1, local rank 1, and co-rank 0.

Note: Whether the executing image is one of those selected in square brackets has no bearing on whether the executing image evaluates the expression or assignment. For example, the statement

```
p[6] = 1
```

is executed by every image, not just image 6. If code is to be executed selectively, the Fortran IF or CASE statement is needed. For example, the code

```
real :: p[*]
  ...
if (this_image(p)==1)then
   read(6,*)p
   p[:] = p
end if
call sync_all
```

employs the first image to read data and broadcast it to other images.

3.4 Procedures

A co-array subobject is permitted only in intrinsic operations, intrinsic assignments, and input/output lists (but see Appendix 1 for a possible extension).

If a dummy argument has co-rank zero, the value of a co-array subobject may be passed by using parentheses to make an expression, for example,

```
c(1:p:2) = sin( (d[1:p:2]) )
```

Note: The behaviour is as if a copy of the section is made on the local image and this copy is passed to the procedure as an actual argument.

If a dummy argument has nonzero co-rank, the co-array properties are defined afresh and are completely independent of those of the actual argument. The interface must be explicit. The actual argument must be the name of a co-array or a subobject of a co-array without any square brackets, vector-valued subscripts, or pointer component selection; any subscript expressions must have the same value on all images. If the dummy argument has nonzero local rank and its local shape is not assumed, the actual argument shall not be an array section, involve component selection, be an assumed-shape array, or be a subobject of an assumed-shape array.

Note: These rules are intended to ensure that copy-in or copy-out is not needed for an argument of nonzero co-rank, and that the actual argument may be stored from the same memory address in each image.

A function result is not permitted to be a co-array.

A pure or elemental procedure is not permitted to contain any Co-Array Fortran extensions.

The rules for resolving generic procedure references remain unchanged.

3.5 Sequence association

COMMON and EQUIVALENCE statements are permitted for co-arrays and specify how the storage is arranged on each image (the same for every one). Therefore, co-array subobjects are not permitted in an EQUIVALENCE statement. For example

```
equivalence (a[10],b[7]) ! Not allowed (compile-time constraint)
```

is not permitted. Appearing in a COMMON and EQUIVALENCE statement has no effect on whether an object is a co-array; it is a co-array only if declared with square brackets. An EQUIVALENCE statement is not

permitted to associate a co-array with an object that is not a co-array. For example

```
integer :: a,b[*]
equivalence (a,b) ! Not allowed (compile-time constraint)
```

is not permitted. A COMMON block that contains a co-array always has the SAVE attribute. Which objects in the COMMON block are co-arrays may vary between scoping units. Since blank COMMON may vary in size between scoping units, co-arrays are not permitted in blank COMMON.

3.6 Allocatable arrays

A co-array may be allocatable. The ALLOCATE statement is extended so that the co-extents can be specified, for example,

```
real, allocatable :: a(:)[:], s[:,:]
:
allocate ( array(10)[*], s[34,*] )
```

The upper bound for the final co-dimension must always be given as an asterisk and values of all the other bounds are required to be the same on all images. For example, the following are not permitted

```
allocate(a(num_images()))     ! Not allowed (compile-time constraint)
allocate(a(this_image())[*]) ! Not allowed (run-time constraint)
```

There is 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 execution of an ALLOCATE statement for the same set of co-arrays. Similarly, for DEALLOCATE, all images delay making the deallocations until they are all about to execute a DEALLOCATE statement for the same set of co-arrays.

Note: These rules are needed to permit all images to store and reference the data consistently. Depending on the implementation, images may need to synchronize both before and after memory is allocated and both before and after memory is deallocated. This synchronization is completely separate from those obtained by calling `sync_all` and `sync_team` (see Sections 3.8 and 3.9).

An allocatable co-array without the SAVE attribute must not have the status of currently allocated if it goes out of scope when a procedure is exited by execution of a RETURN or END statement.

When an image executes an allocate statement, no communication is involved apart from any required for synchronization. The image allocates the local part and records how the corresponding parts on other images are to be addressed. The compiler, except perhaps in debug mode, is not required to enforce the rule that the bounds are the same on all images. Nor is the compiler responsible for detecting or resolving deadlock problems. For allocation of a co-array that is local to a recursive procedure, each image must descend to the same level of recursion or deadlock may occur.

3.7 Array pointers

A co-array is not permitted to be a pointer (but see Appendix 2, where a possible extension is described).

A co-array may be of a derived type with pointer components. For example, if `p` is a pointer component, `z[i]%p` is a reference to the target of component `p` of `z` on image `i`. To avoid references with co-array syntax to data that is not in a co-array, we limit each pointer component of a co-array to the behaviour of an allocatable component of a co-array:

1. a pointer component of a co-array is not permitted on the left of a pointer assignment statement

(compile-time constraint),

2. a pointer component of a co-array is not permitted as an actual argument that corresponds to a pointer dummy argument (compile-time constraint), and
3. if an actual argument of a type with a pointer component is part of a co-array and is associated with a dummy argument that is not a co-array, the pointer association status of the pointer component must not be altered during execution of the procedure (this is not a compile-time constraint).

To avoid hidden references to co-arrays, the target in a pointer assignment statement is not permitted to be any part of a co-array. For example,

```
q => z[i]%p ! Not allowed (compile-time constraint)
```

is not permitted. Intrinsic assignments are not permitted for co-array subobjects of a derived type that has a pointer component, since they would involve a disallowed pointer assignment for the component:

```
z[i] = z ! Not allowed if Z has a pointer
z = z[i] ! component (compile-time constraint)
```

Similarly, it is legal to allocate a co-array of a derived type that has pointer components, but it is illegal to allocate one of those pointer components on another image:

```
type(something), allocatable ::  t[:]
  ...
allocate(t[*])          !  Allowed
allocate(t%ptr(n))      !  Allowed
allocate(t[q]%ptr(n))   !  Not allowed (compile-time constraint)
```

3.8 Execution control

Most of the time, each image executes on its own as a Fortran 95 program without regard to the execution of other images. It is the programmer's responsibility to ensure that whenever an image alters co-array data, no other image might still need the old value. Also, that whenever an image accesses co-array data, it is not an old value that needs to be updated by another image. The programmer uses invocations of the intrinsic synchronization procedures to do this, and the programmer should make no assumptions about the execution timing on different images. This obligation on the programmer provides the compiler with scope for optimization. When constructing code for execution on an image, it may assume that the image is the only image in execution until the next invocation of one of the intrinsic synchronization procedures and thus it may use all the optimization techniques available to a standard Fortran 95 compiler.

In particular, if the compiler employs temporary memory such as cache or registers (or even packets in transit between images) to hold co-array data, it must copy such data to memory that can be accessed by another image to make it visible to it. Also, if another image changes the co-array data, the executing image must recover the data from global memory to the temporary memory it is using. The intrinsic procedure `sync_memory` is provided for both purposes. It is concerned only with data held in temporary memory on the executing image for co-arrays in the local scope. Given this fundamental intrinsic procedure, the other synchronization procedures can be programmed in Co-Array Fortran (see Appendix 3), but the intrinsic versions, which we describe next, are likely to be more efficient. In addition, the programmer may use it to express customized synchronization operations in Co-Array Fortran.

Note: A compiler can hold co-arrays in temporary storage, such as cache or registers, between calls to `sync_memory`.

If data calculated on one image are to be accessed on another, the first image must call `sync_memory` after the calculation is complete and the second must call `sync_memory` before accessing the data. Synchronization is needed to ensure that `sync_memory` is called on the first before `sync_memory` is called on the second.

Note: If the local part of a co-array or a subobject of it is an actual argument corresponding to a dummy argument of zero co-rank, a copy may be passed to the procedure. To avoid the possibility of the original being altered by another image after the copy has been made, a synchronization may be needed ahead of the procedure invocation. Similarly, a synchronization is needed after return before any other image accesses the result.

The subroutine `sync_team` provides synchronization for a team of images. The subroutine `sync_all` (see Section 3.10) provides a shortened call for the important case where the team contains all the images. Each invocation of `sync_team` or `sync_all` has the effect of `sync_memory`. The subroutine `sync_all` is not discussed further in this section.

For each invocation of `sync_team` on one image of a team, there shall be a corresponding invocation of `sync_team` on every other image of the team. The n^{th} invocation for the team on one image corresponds to the n^{th} invocation for the team on each other image of the team, n = 1, 2, ... The team is specified in an obligatory argument `team`.

Note: Corresponding calls usually arise from a single statement, but this is not always the case. Some images may be executing different code that still needs synchronization. An example is given in Section 4.5.

The subroutine also has an optional argument `wait`. If this argument is absent from a call on one image it must be absent from all the corresponding calls on other images of the team. If `wait` is absent, each image of the team waits for all the other images of the team to make corresponding calls. If `wait` is present, the image is required to wait only for the images specified in `wait` to make corresponding calls.

Note: No information is available about whether an action on one image occurs before or after an action on another image unless one is executed ahead of a synchronization call and the other is executed behind the corresponding synchronization call on the other. For example, while one image executes the statements between two invocations of `sync_all`, another image might be out of execution. Here is a example that imposes the fixed order 1, 2, ... on images:

```
me = this_image()
if(me>1)             call sync_team( me-1 )
  p[6] = p[6] + 1
if(me<num_images()) call sync_team( me+1 )
```

Without a further call of `sync_memory,` the full result is available only on the last image.

Teams are permitted to overlap, but the following rule is needed to avoid any possibility of deadlock. If a call for one team is made ahead of a call for another team on a single image, the corresponding calls shall be in the same order on all images in common to the two teams.

Note: The presence of the optional argument `wait` allows the invoking image to continue execution without waiting for all the others in cases where it does not need data from all the others. Judicious use of this optional argument can improve the overall efficiency substantially. Implementations, however, are not required to cause immediate continued execution. Implementations for machines with an efficient hardware barrier, for example, may choose to wait for the whole team, which certainly more than satisfies the requirement that the members of `wait` have arrived.

The intrinsic `sync_file` plays a similar role for file data to that of `sync_memory` for co-array data. Because of the high overheads associated with file operations, `sync_team` does not have the effect of

`sync_file`. If data written by one image to a file is to be read by another image without closing the connection and re-opening it on the other image, calls of `sync_file` on both images are needed (details in Section 3.9).

To avoid the need for the programmer to place invocations of `sync_memory` around many procedure invocations, these are implicitly placed around any procedure invocation that might involve any reference to `sync_memory`. Formally, we define a **caf procedure** as

1. an external procedure;
2. a dummy procedure;
3. a module procedure that is not in the same module;
4. `sync_all`, `sync_team`, `sync_file`, `start_critical`, `end_critical`; or
5. a procedure whose scoping unit contains an invocation of `sync_memory` or a caf procedure reference.

Invocations of `sync_memory` are implicitly placed around every caf procedure reference.

Exceptionally, it may be necessary to limit execution of a piece of code to one image at a time. Such code is called a critical section. We provide the subroutine `start_critical` to mark the commencement of a critical region and the subroutine `end_critical` to mark its completion. Both have the effect of `sync_memory`. Each image maintains an integer called its **critical count**. Initially, all these counts are zero. On entry to `start_critical`, the image waits for the system to give it permission to continue, which will only happen when all other images have zero critical counts. The image then increments its critical count by one and returns. Having these counts permits nesting of critical regions. On entry to `end_critical`, the image decrements its critical count by one and returns. We have not found a way to program these subroutines in Co-Array Fortran.

Note: Actions inside a critical region on one image are always separated from actions inside a critical regions on another, but one image could be inside a critical region while another is simultaneously executing statements outside a critical region. In the following

```
me = this_image()
call start_critical
  p[6] = p[6] + 1
call end_critical
if (me==1) then
  call sync_all( (/ (i, i=1,num_images()) /) )
else
  call sync_all( me )
endif
```

the critical region guarantees atomic update of `p[6]`, but the `sync_all` is required to make the full result available on image 1.

The effect of a STOP statement is to cause all images to cease execution. If a delay is required until other images have completed execution, a synchronization statement should be employed.

3.9 Input/output

Most of the time, each image executes its own read and write statements without regard for the execution of other images. However, Fortran 95 input and output processing cannot be used from more than one image without restrictions unless the images reference distinct file systems. Co-Array Fortran assumes that all images reference the same file system, but it avoids the problems that this can cause by specifying a single set of I/O units shared by all images and by extending the file connection statements to identify which images have access to the unit.

It is possible for several images to be connected on the same unit for direct-access input/output. The intrinsic `sync_file` may be used to ensure that any changed records in buffers that the image is using are copied to the file itself or to a replication of the file that other images access. This intrinsic plays the same role for I/O buffers as the intrinsic `sync_memory` does for temporary copies of co-array data. Execution of `sync_file` also has the effect of requiring the reloading of I/O buffers in case the file has been altered by another image. Because of the overheads of I/O, `sync_file` applies to a single file.

It is possible for several images to to be connected on the same unit for sequential output. The processor shall ensure that once an image commences transferring the data of a record to the file, no other image transfers data to the file until the whole record has been transferred. Thus, each record in an external file arises from a single image. The processor is permitted to hold the data in a buffer and transfer several whole records on execution of `sync_file`.

Note: `sync_file` 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. Without a `sync_file`, all writes could be buffered locally until the file is closed. If two images write to the same record of a direct-access file, it is the programmers responsibility to separate the writes by appropriate `sync_file` calls and image synchronization. This is a consequence of the need to make no assumptions about the execution timing on different images.

The I/O keyword TEAM is used to specify an integer rank-one array, `connect_team`, for the images that are associated with the given unit. All elements of `connect_team` shall have values between 1 and `num_images()` and there shall be no repeated values. One element shall have the value `this_image()`. The default `connect_team` is `(/this_image()/)`.

The keyword TEAM is a connection specifier for the OPEN statement. All images in `connect_team`, and no others, shall invoke OPEN with an identical *connection-spec-list*. There is an implied call to `sync_team` with the single argument `connect_team` before and after the OPEN statement. The OPEN statement connects the file on the invoking images only, and the unit becomes unavailable on all other images. If the OPEN statement is associated with a processor dependent file, the file is the same for all images in `connect_team`. If `connect_team` contains more than one image, the OPEN shall have `ACCESS=DIRECT` or `ACTION=WRITE`.

An OPEN on a unit already connected to a file must have the same `connect_team` as currently in effect.

A file shall not be connected to more than one unit, even if the `connect_team`s for the units have no images in common.

Pre-connected units that allow sequential read shall be accessible on the first image only. All other pre-connected units have a `connect_team` containing all the images.

Note: The input unit identified by * is therefore only available on image 1.

CLOSE has a TEAM= specifier. If the unit exists and is connected on more than one image, the CLOSE statement must have the same `connect_team` as currently in effect. There is an implied call to

`sync_file` for the unit before CLOSE. There are implied calls to `sync_team` with single argument `connect_team` before and after the implied `sync_file` and before and after the CLOSE.

BACKSPACE, REWIND, and ENDFILE have a TEAM= specifier. If the unit exists and is connected on at least one image, the file positioning statement must have the same `connect_team` as currently in effect. There is an implied call to `sync_file` for the unit before the file positioning statement. There are implied calls to `sync_team` with single argument `connect_team` before and after the implied `sync_file` and before and after the file positioning statement.

A companion paper (Wallcraft, Numrich and Reid 1998) is in preparation to discuss Co-Array Fortran I/O in more detail.

3.10 Intrinsic procedures

Co-Array Fortran adds the following intrinsic procedures. Only `num_images`, `log2_images`, and `rem_images` are permitted in specification expressions. None are permitted in initialization expressions. We use italic square brackets *[ ]* to indicate optional arguments.

`end_critical()` is a subroutine for limiting synchronous execution. Each image holds an integer called its critical count. On entry, the count for the image shall be positive. The subroutine decrements this count by one. `end_critical` has the effect of `sync_memory`.

`log2_images()` returns the base-2 logarithm of the number of images, truncated to an integer. It is an inquiry function whose result is a scalar of type default integer.

`num_images()` returns the number of images. It is an inquiry function whose result is a scalar of type default integer.

`rem_images()` returns `mod(num_images(),2**log2_images())`. It is an inquiry function whose result is a scalar of type default integer.

`start_critical()` is a subroutine for limiting synchronous execution. Each image holds an integer called its critical count. Initially all these counts are zero. The image waits for the system to give it permission to continue, which will only happen when all other images have zero critical counts. The image then increments its critical count by one and returns. `start_critical` has the effect of `sync_memory`.

`sync_all(`*[*`wait`*]*`)` is a subroutine that synchronizes all images. `sync_all()` is treated as `sync_team(all)` and `sync_all(wait)` is treated as `sync_team(all,wait)`, where `all` has the value `(/ (i,i=1,num_images()) /)`.

`sync_all(`*[*`wait`*]*`)` has the effect of `sync_memory`.

`sync_file(unit)` is a subroutine for marking the progress of input-output on a unit. `unit` is an INTENT(IN) scalar argument of type integer and specifies the unit. The subroutine affects only the data for the file connected to the unit. If the unit is not connected on this image or does not exist, the subroutine has no effect. Before return from the subroutine, any file records that are held by the image in temporary storage and for which WRITE statements have been executed since the previous call of `sync_file` on the image (or since execution of OPEN in the case of the first `sync_file` call) shall be placed in the file itself or a replication of the file that other images access. The first subsequent access by the image to file data in temporary storage shall be preceded by data recovery from the file itself or its replication. If the unit is connected for sequential access, the previous WRITE statement shall have been for advancing input/output.

`sync_team(team [,wait])` is a subroutine that synchronizes images. `team` is an INTENT(IN) argument that is of type integer and is scalar or of rank one. The scalar case is treated as if the argument were the array `(/this_image(),team/)`; in this case, `team` must not have the value `this_image()`. All elements of `team` shall have values in the range $1 \le$ `team(i)` $\le$ `num_images()` and there shall be no repeated values. One element of `team` shall have the value `this_image()`. `wait` is an optional INTENT(IN) argument that is of type integer and is scalar or of rank one. Each element, if any, of `wait` shall have a value equal to that of an element of `team`. The scalar case is treated as if the argument were the array `(/wait/)`.

The argument `team` specifies a team of images that includes the invoking image. For each invocation of `sync_team` on one image, there shall be a corresponding invocation of `sync_team` for the same team on every other image of the team. The n^{th} invocation for the team on one image corresponds to the n^{th} invocation for the team on each other image of the team, $n = 1, 2, \ldots$. If a call for one team is made ahead of a call for another team on a single image, the corresponding calls shall be in the same order on all images in common to the two teams.

If `wait` is absent on one image it must be absent in all the corresponding calls on the other images of the team. In this case, `wait` is treated as if it were equal to `team` and all images of the team wait until all other images of the team are executing corresponding calls. If `wait` is present, the image waits for all the images specified by `wait` to execute corresponding calls.

`sync_team(team[,wait])` has the effect of `sync_memory`.

`sync_memory()` is a subroutine for marking the progress of the execution sequence. Before return from the subroutine, any co-array data that is accessible in the scoping unit of the invocation and is held by the image in temporary storage shall be placed in the storage that other images access. The first subsequent access by the image to co-array data in this temporary storage shall be preceded by data recovery from the storage that other images access.

Note: Temporary storage includes registers and cache, but could also include network packets in transit between nodes of a distributed memory machine.

`this_image([array[,dim]])` returns the index of the invoking image, or the set of co-subscripts of `array` that denotes data on the invoking image. The type of the result is always default integer. There are four cases:

Case (i). If `array` is absent, the result is a scalar with value equal to the index of the invoking image. It is in the range 1, 2, ..., `num_images()`.

Case (ii). If `array` is present with co-rank 1 and `dim` is absent, the result is a scalar with value equal to co-subscript of the element of `array` that resides on the invoking image.

Case (iii). If `array` is present with co-rank greater than 1 and `dim` is absent, the result is an array of size equal to the co-rank of `array`. Element k of the result has value equal to co-subscript k of the element of `array` that resides on the invoking image.

Case (iv). If `array` and `dim` are present, the result is a scalar with value equal to co-subscript `dim` of the element of `array` that resides on the invoking image.

4 Improved versions of the examples

In this section, we revisit some of the examples of Section 2, to illustrate how co-array syntax may be used to extend the scope or to improve the execution performance.

4.1 Finite differencing on a rectangular grid

We begin by extending the example of Section 2.1 to three dimensions, with two dimensions spread over images. This illustrates the convenience of having co-arrays with more than one co-dimension.

```
subroutine laplace (nrow,ncol,nlevel,u)
   integer, intent(in)  ::  nrow, ncol, nlevel
   real, intent(inout)  ::  u(nrow)[ncol,*]
   real                 ::  new_u(nrow)
   integer              ::  i, me(2), left, right, up, down
   new_u(1) = u(nrow) + u(2)
   new_u(nrow) = u(1) + u(nrow-1)
   new_u(2:nrow-1) = u(1:nrow-2) + u(3:nrow)
   me = this_image(u)
   left = me(1)-1; if (me(1) == 1) left = ncol
   right = me(1) + 1; if (me(1) == ncol) right = 1
   down = me(2)-1; if (me(2) == 1) down = nlevel
   up = me(2) + 1; if (me(2) == nlevel) up = 1
   call sync_all
   new_u(1:nrow)=new_u(1:nrow)+u(1:nrow)[left,me(2)]+u(1:nrow)[right,me(2)]&
                              +u(1:nrow)[me(1),down)]+u(1:nrow)[me(1),up]
   call sync_all
   u(1:nrow) = new_u(1:nrow) - 6.0*u(1:nrow)
end subroutine laplace
```

4.2 Data redistribution

The code of Section 2.2 will lead to bottlenecks since all images will begin by accessing data on the first image. We can circumvent this:

```
iz = this_image(a)
if (iz<=kz) then
   do i = 1, kx-1
      ix = iz + i; if (ix>kx) ix = ix - kx
      a(ix,:) = b(:,iz)[ix]
   end do
   a(iz,:) = b(:,iz)
end if
```

Each image begins by accessing its upper neighbour and then continues in ascending order, with wrap-around, until all required images are accessed. Also, we have taken the opportunity to avoid square brackets for the local transfer. Note that we again have a very clear representation of the action required.

Statements like those that set `ix` in this example, or the equivalent `ix=mod(iz+i-1,kx)+1`, are often used in Co-Array Fortran programs to load balance remote memory operations. The value of `kx` is often `num_images()`.

4.3 Maximum value of a co-array section

Dummy arguments may be co-arrays, but to ease the implementation task no co-rank or co-shape information is transferred as part of the co-array itself. Where such information is required, other Fortran mechanisms must be used. For example, we may alter the example of Section 2.3 to

```
subroutine greatest(first,last,a,great)
                 ! Find maximum value of a(:)[first:last]
   integer, intent(in) :: first, last
   real, intent(in)  :: a(:)[*]
   real, intent(out) :: great[*]
                  ! Place result in great[first:last]
   real, allocatable :: work(:) ! Local work array
   integer :: i, this
   this = this_image(great)
   if (this>=first .and. this<=last) then
      great = maxval(a)
      call sync_team( (/(i, i=first,last)/) )
      if(this==first)then
         allocate (work(first:last))
         work = great[first:last]    ! Gather local maxima
         great[first:last]=maxval(work) ! Scatter global maximum
         deallocate (work)
      end if
      call sync_team( (/(i, i=first,last)/) )
   end if
end subroutine greatest
```

Here, we have used the intrinsic `sync_team` to synchronize a subset of images by using an array constructor to give it a list of image indices. If we were to attempt to synchronize all images, deadlock would result since the call is made only for the selected images.

4.4 Summing over the co-dimension of a co-array (1)

A better implementation of the example of Section 2.5 involves all the images in $\log_2 np$ stages, where np is the number of images. At the beginning of step k of the new algorithm, the images will be in evenly-spaced groups of length $\sigma = 2^{k-1}$. Each image will hold the sum over its a group. New groups are formed from groups 1 and 2, 3 and 4, etc. Each image is partnered by an image in the same new group at a distance σ, exchanges data with its partner, then performs a summation. After this, each image will hold the sum over images of its new group. This continues until all images are in one group. Assuming that the number of images is an integral power of 2, the following code suffices:

```
subroutine sum_reduce(x)
   real, intent(inout) :: x(:)[*]
   ! Replace x by the result of summing over the co-dimension
   real work(size(x))
   integer k,my_partner,me,span
   me =  this_image(x)
   span = 1
   do k=1,log2_images()
      if(mod(me-1,2*span)<span)then
         my_partner = me + span
      else
         my_partner = me - span
      end if
      call sync_team(my_partner)
      work(:) = x(:)[my_partner]
      call sync_team(my_partner) ! Do not change x until we are sure
```

```
                          ! that the partner has the old value
      x(:) = x(:) + work(:)
      span = span*2
   end do
end subroutine sum_reduce
```

4.5 Summing over the co-dimension of a co-array (2)

If the number of images is not an integral power of 2, the code is more complicated. Let the number be $2^p + r$, with $r < 2^p$. The value p is returned by `log2_images()` and the value r is returned by `rem_images()`. The main loop is similar, but is restricted to the first $m = 2^p$ images. We begin by adding the data of the last r images into the first r:

```
m = num_images() - rem_images()
if(me <= rem_images()) then
    call sync_team(me+m)
    x(:) = x(:) + x(:)[me+m]
else if (me>m) then
    call sync_team(me-m)
end if
```

The invocation of `sync_team` in the else clause is needed to keep numbers of invocations on all the images in phase.

The main loop takes the form:

```
if(me <= m) then
  do k=1,log2_images()
      if(mod(me-1,2*span)<span)then
         my_partner = me + span
      else
         my_partner = me - span
      end if
      call sync_team( my_partner )
      work(:) = x(:)[my_partner]
      call sync_team( my_partner )
      x(:) = x(:) + work(:)
      span = span*2
   end do
end if
```

and there needs to be a final step where the last r images get their results:

```
if(me <= rem_images()) then
    call sync_team(me+m)
    x(:)[me+m] = x(:)
else if (me>m) then
    call sync_team(me-m)
end if
```

5 Comparison with other parallel programming models

Most other proposals for parallel programming models for the Fortran language are based either on directives (for example, HPF, see Koebel *et al.* 1994 and OpenMP 1997) or on libraries (for example, MPI 1995 and MPI-2 1997). What are the advantages of the co-array extension?

Compiler directives were originally designed as temporary expedients to help early compilers recognize vectorizable code, but have become 'languages' superimposed on languages. Some directives are really executable statements, which we see as misleading.

Such a programming model throws the burden onto the programmer's shoulders as much as the message-passing model does. The programmer has to decide what parts of the code are serial and what variables to distribute and how. The programmer makes sure the directives are consistent throughout the program. The programmer identifies any memory race conditions and inserts appropriate directives to eliminate them.

On a distributed memory computer, HPF directives tempt the programmer to underestimate the importance of localizing computation and minimizing communication. It often becomes clear that good performance requires the same attention to detail as for the message-passing model. HPF includes extrinsic procedures so that the programmer can drop out of the directive-based model into the message-passing model where all the details of distributed memory must be handled explicitly. After mastering all the intricacies of compiler directives, the programmer throws away most of the global information contained in them to obtain good performance.

Some programmers, encountering co-array syntax for the first time, point out that compiler directives can be used to mimic the co-array programming style. Given a Co-Array Fortran program, it is sometimes easy to add directives to produce an artificial directive-based solution. To distribute data and work using directives, the programmer may add artificial extra dimensions, which propagate through the entire code carrying along global information that is unimportant in most of the code. The co-array programming model shows that the directives and the extra dimensions are not only cumbersome but also superfluous. Directive-based models require compilers to recognize and to implement long lists of directives, which may or may not behave the same way on all platforms. Co-Array Fortran requires compilers to recognize only a simple extension to the language.

In some cases, especially when code does not translate naturally into data parallel array syntax, this technique of adding extra dimensions for emulating Co-Array Fortran syntax may be the only technique that works efficiently. But this technique does not use the data parallel programming model in the way that it was intended. It mimics message-passing, which is hard enough with all its bookkeeping, and then adds another layer of difficulty with directives. Co-Array Fortran requires the same bookkeeping but removes the artificial directives.

What about using a library-based model? The Co-Array Fortran philosophy is that the foundation for a parallel programming model should be simple with more complicated libraries erected on top of the foundation. Basing a parallel model on a complicated library followed by ever more complicated libraries on top of libraries makes the programmer's job more difficult not simpler.

Even a simple library like the one-sided SHMEM library, which has become the model of choice for writing parallel applications for the CRAY-T3D and the CRAY-T3E, has a number of limitations. (Sawdey *et al.* 1995). Data used for communication must be allocated statically, normally in common blocks, making dynamic memory management difficult. Default variable sizes change from machine to machine causing portability problems. The programmer is restricted to fixed communication patterns supported by the library and is allowed to communicate with only one memory image at a time. Memory images must be linearized

starting at zero. Library functions normally block until transfers complete and the overhead from subroutine calls lowers efficiency for fine-grained communication.

Co-array syntax removes the requirement for static memory allocation. The compiler knows variable sizes. The programmer can write arbitrary communication patterns for arbitrary variable types. The programmer defines the memory grid and the numbering scheme and can change them anywhere that it makes sense to do so. The compiler generates in-line code that it can optimize to support fine-grained communication patterns. Execution need not block until communication completes and communication with more than one memory image at a time is possible.

Another library-based model is BSP (Bulk Synchronous Programming) (Valiant 1990; Hill, McColl, Stefanescu, Goudreau, Lang, Rao, Suel, Tsantilas, and Bisseling 1997), which is close to Co-Array Fortran. The computation proceeds in discrete steps, alternating between a step of independent computation involving no communication and a step of communication involving no computation. Each step is terminated with a global barrier, equivalent to `sync_all()`.

MPI (MPI 1995, MPI-2 1997) is a de-facto standard for the two-sided message-passing model (Gropp, Lusk, and Skjellum 1994). It is actually a C library that may be called from Fortran, but there are serious inconsistencies with Fortran 90:

1. Some MPI subroutines accept arguments with differing types (choice argments).
2. Some MPI subroutines accept arguments with differing array properties (sometimes arrays and sometimes scalars that are not array elements).
3. Many MPI routines assume that actual arguments are passed by address and that copy-in copy-out does not occur.
4. An MPI implementation may read or modify user data (e.g. communication buffers used by nonblocking communications) after return.

Further, the calls to MPI procedures require many arguments, which leads to code that is hard to understand and therefore liable to bugs. A simple communication pattern may turn into an arcane code sequence. Co-array syntax, on the other hand, requires no include files, no status buffers, no model initialization, no message tags, no error codes, no library-specific variables, and no variable size information. Moving an arbitrarily complicated Fortran 90 data structure creates no problem for co-array syntax, but there is no simple equivalent in MPI.

The following example is due to Alan Wallcraft of the Naval Research Laboratory, Stennis Space Center, Mississippi, and comes from an actual halo exchange subroutine. He comments: ‘ The co-array version is much easier for Fortran 90 programmers to understand, and it is typically 2-3 times faster than any MPI version on SMP/DSM systems’.

Co-Array Fortran:

```
call sync_all(imgi(nproc-1:nproc+1))
ai(:,:,4:4)=ai(:,:,1:1)[imgi(nproc-1)]
ai(:,:,5:6)=ai(:,:,2:3)[imgi(nproc+1)]
call sync_all(imgi(nproc-1:nproc+1))
```

The fastest MPI version depends on the machine, but the following is usually fast:

```
if (nfirst.eq.0) then
   nfirst = 1
   call mpi_send_init(ai(1,1,1),  ihp*ip,mpi_real,imgi(nproc+1),9905, &
                      mpi_comm_world,mpireq(1),mpierr)
   call mpi_send_init(ai(1,1,2),2*ihp*ip,mpi_real,imgi(nproc-1),9906, &
```

```
                    mpi_comm_world,mpireq(2),mpierr)
   call mpi_recv_init(ai(1,1,4),  ihp*ip,mpi_real,imgi(nproc-1),9905, &
                    mpi_comm_world,mpireq(3),mpierr)
   call mpi_recv_init(ai(1,1,5),2*ihp*ip,mpi_real,imgi(nproc+1),9906, &
                    mpi_comm_world,mpireq(4),mpierr)
endif
call mpi_startall(4, mpireq, mpierr)
call mpi_waitall( 4, mpireq, mpistat, mpierr)
```

6 Summary

We conclude with a summary of the features of Co-Array Fortran.

1. A Co-Array Fortran program is replicated to images with indices 1, 2, 3, ... and runs asynchronously on them all. The number of images is fixed throughout execution.
2. Variables, but not structure components, may be declared with trailing dimensions in square brackets. These are called co-arrays and the trailing dimensions, called co-dimensions, provide access from one image to data on any other image. The co-size of a co-array is always equal to the number of images.
3. A co-array may be of assumed co-size or be allocatable.
4. There is implicit synchronization of all images in association with each allocation or deallocation of a co-array.
5. Normal array syntax is extended to include square brackets and the resulting objects are called co-array subobjects. A co-array subobject may appear in an expression within parentheses or as an operand of an intrinsic operation; in both cases, the behaviour is as if a temporary array were allocated on the local image and the data copied to it from other images. A co-array subobject may appear on the left of an intrinsic assignment. The rank of a co-array subobject is the sum of its local rank and its co-rank and must not exceed seven. The shape is the concatenation of its local shape and its co-shape. Conformance in array expressions and assignments is by shape.
6. A reference to a co-array without square brackets is a reference to the local object.
7. Dummy arguments, but not function results, may be co-arrays. The corresponding actual argument must be the name of co-array or a subobject of a co-array without any square brackets or component selection. The interface must be explicit. The co-array properties are specified afresh without regard to the co-array properties of the actual argument. The co-array properties are disregarded in applying the rules for argument association and resolution of generic invocations.
8. There are intrinsic functions that return the number of images, the base-2 logarithm of the number of images, and the remainder after the largest integer power of 2 is subtracted from the number of images.
9. There is an intrinsic subroutine for synchronizing the executing image with another image, a set of other images, or all other images.
10. Co-arrays are not permitted to be pointers, but a co-array may be of a derived type with pointer components. Such a component is restricted to the role of an allocatable component. A pointer is not permitted to become associated with a co-array.
11. There is an intrinsic function that returns the index of the executing image or the co-subscripts of a

co-array that denote data on the executing image.

12. Input and output is performed independently by each image. However, there is a single file system and a single set of units shared by all images. The new keyword TEAM is used to identify which images may perform I/O on a given unit. There is an intrinsic for marking the progress of input-output on a unit.
13. Execution of a STOP on any images causes all images to cease execution.

At the beginning of this paper, we asked the question 'What is the smallest change required to convert Fortran 95 into a robust, efficient parallel language?'. The answer to our question is contained in just nine pages of text in Section 3, which describes a simple syntactic extension to Fortran 95. The rest of the paper contains examples to illustrate the simplicity and elegance of co-array syntax, which places enormous power and flexibility in the hands of the programmer. We feel that once Fortran programmers begin to use co-array syntax, it will become the model of choice for parallel programming.

7 Acknowledgements

We would like to express our special thanks to Alan Wallcraft of the Naval Research Laboratory, Stennis Space Center, Mississippi for his enthusiastic support and many helpful email exchanges, particularly in connection with the design of the synchronization procedures and of the I/O features. We would also like to thank Dick Hendrickson of Imagine1, Inc. for his careful reading of early drafts of this paper and his many helpful suggestions and to Mike Metcalf formerly of CERN, Ian Gladwell of SMU Dallas, and Bill Long of Silicon Graphics for their suggestions after reading more recent drafts.

8 References

Gropp, W., Lusk, E., and Skjellum, A. (1994). Using MPI, portable parallel programming with the Message-Passing Interface. The MIT Press.

Hill, J. M. D., McColl, W., Stefanescu, D. C., Goudreau, M. W., Lang, K., Rao, S. B., Suel, T., Tsantilas, T. and Bisseling, R. (1997). The BSP programming library. To appear in *Parallel Computing*.

Koebel, C. H., Loveman, D. B., Schrieber, R. S., Steele, G. L., and Zosel, M. E. (1994). The High Performance Fortran Handbook, M. I. T. Press, Cambridge, Massachusetts.

MPI (1995). A message-passing interface standard. http://www.mcs.anl.gov/mpi/index.html

MPI-2 (1997). Extensions to the message-passing interface. http://www.mcs.anl.gov/mpi/index.html

Numrich, R. W. (1991). An explicit node-oriented programming model. Private Report, Cray Research, Inc., March 1991. Available from the author.

Numrich, R. W. (1994a). The Cray T3D address space and how to use it. Private Report, Cray Research, Inc., April 1994. Available from the author.

Numrich, R. W. (1994b). F^{--}: A parallel Fortran language, Private Report, Cray Research, Inc., April 1994. Available from the author.

Numrich, R. W. (1997). F−−: A parallel extension to Cray Fortran. *Scientific Programming* **6**, 275-284.

Numrich, R. W., Reid, J. K., and Kim, K. (1998). Writing a multigrid solver using Co-array Fortran. To appear in the Proceeding of the fourth International Workshop on Applied Parallel Computing (PARA98), Umeå University, Umeå Sweden, June, 1998.

Numrich, R. W., Springer, P. L. and Peterson, J. C. (1994). Measurement of communication rates on the Cray T3D interprocessor network. In *High-Performance Computing and Networking, International Conference and Exhibition, Munich, Germany, April 18-20, 1994, Proceedings, Volume 2: Networking and Tools,* Eds Wolfgang Gentzsch and Uwe Harms, Springer-Verlag, pp. 150-157.

Numrich, R. W. and Steidel, J. L. (1997a). F−−: A simple parallel extension to Fortran 90. SIAM News, **30**, 7, 1-8.

Numrich, R. W. and Steidel, J. L. (1997b). Simple parallel extensions to Fortran 90. Proc. eighth SIAM conference of parallel processing for scientific computing, Mar. 1997.

Numrich, R. W., Steidel, J. L., Johnson, B. H., de Dinechin, B. D., Elsesser, G., Fischer, G., and MacDonald, T. (1998). Definition of the F−− extension to Fortran 90. Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computers, Lectures on Computer Science Series, Number 1366, Springer-Verlag.

OpenMP (1997). OpenMP: a proposed industry standard API for shared memory programming. http:www.openmp.org

Pase, D. M., MacDonald, T., and Meltzer, A. (1994). The CRAFT Fortran Programming Model. Scientific Programming, **3**, 227-253.

Sawdey, A., O'Keefe, M., Bleck, R. and Numrich, R. W. (1995). The design, implementation, and performance of a parallel ocean circulation model. Proceedings of the Sixth ECMWF Workshop on the Use of Parallel Processors in Meteorology, Reading, England, November 1994, World Scientific Publishers, pp. 523-550.

Valiant, L. G. (1990). A bridging model for parallel computation. *Communications ACM* **33**, 103-111.

Wallcraft, A.J., R.W. Numrich and J.K. Reid (1998). An I/O model for Co-Array Fortran. In preparation.

Appendix 1. Extension to allow co-array subobjects as actual arguments

A possible extension of the language would allow co-array subobjects as actual arguments. In this appendix, we give the additional rules that such an extension would imply.

If a co-array subobject of nonzero co-rank is associated as an actual argument with a dummy argument of a procedure with no co-array dummy arguments, the behaviour is as if copy-in copy-out is employed. This allows all Fortran 95 intrinsics and application code procedures to be referenced with actual arguments that are co-array subobjects.

To avoid hidden references to other images, a co-array subobject is not permitted as an actual argument corresponding to a pointer dummy argument.

In a generic procedure reference, including a defined operation or a defined assignment, the ranks and the co-ranks of the actual and corresponding dummy arguments must match if any dummy argument is a co-array and the actual argument is a co-array subobject.

Two procedures are permitted to be overlaid in a generic interface if a non-optional argument differs in rank or co-rank. For example, the following interface is valid:

```
        subroutine sub1(a)
           real a(:)
        end subroutine sub1
        subroutine sub2(b)
           real b(:)[:]
        end subroutine sub2
        subroutine sub3(c)
           real c(:,:)
        end subroutine sub3
        subroutine sub4(d)
           real d[:,:]
        end subroutine sub4
        subroutine sub5(e)
           real e[:]
        end subroutine sub5
    end interface sub
```

Given the array

```
real :: c(10)[*]
```

the following calls are valid:

```
call sub(c)       ! Calls sub1
call sub(c(:))    ! Calls sub1
call sub(c(1)[:]) ! Calls sub5
call sub(c(:)[:]) ! Calls sub2
```

In resolving a generic call, a specific procedure that matches in rank and co-rank is given preference to a procedure with no co-array arguments that matches only in rank. For example, if `sub5` were removed from the above interface,

```
call sub(c(1)[:])
```

would result in a call to `sub1`.

The following intrinsic procedure is added:

`image_indices(source)` is an inquiry function that returns the indices of the images on which a

co-array subobject resides.

A2.1 Sequence association

For a co-array subobject actual argument of nonzero co-rank in a procedure reference that is not generic, the rules of sequence association are applied separately to the local and co-array parts. The co-ranks of the actual and dummy arguments are permitted to differ. For example, the following is permitted:

```
call sub(a[:,:])
   :
subroutine sub(a)
real :: a[*]
```

The local ranks are allowed to differ, except that if one is zero, so is the other. The two correspondences are independent. One says how the storage is arranged on each image (and it is the same on each) and the other says how the image addressing will be arranged. We therefore need something for images that is akin to the array element order in Fortran 95.

The images of a co-array or co-array subobject form a sequence called the **image order**. For a co-array that is not a dummy argument associated with a co-array subobject, the images are 1, 2, 3, ... For a co-array subobject, the images are selected from those of its parent. For a dummy argument associated with a co-array subobject, the images are selected from those of the associated actual argument. The position of an arbitrary image element is determined by the subscript order value of the subscript list designating the image element using the same formulas as those for computing ordinary subscript order values.

Note that the image indices of a dummy co-array associated with a co-array subobject need not commence at one and need not be contiguous. Consider for example,

```
real a(100)[32]
...
call sub(a(:)[5:30:5], ... )
```

Here, the first dummy argument of `sub` will be a co-array that resides on images with indices 5, 10, 15, ... The intrinsic procedure `image_indices` is available to provide lists of image indices when needed. For example, `image_indices(a(:)[5:30:5])` has the value `(/5,10,15,20,25,30/)`.

A restriction is needed to avoid the possibility of data redistribution across a procedure interface. Hence, if the parent of the actual argument is of assumed co-shape, the local ranks and the co-ranks must both agree. Also, restrictions are needed so that the implementation nevers needs to perform copy-in copy-out when the dummy argument is a co-array; for example, if the dummy argument has explicit local shape, the actual argument must not have assumed local shape.

Appendix 2. Extension to allow co-array pointers

A possible extension of the language would allow co-array pointers. In this section, we give the additional rules that such an extension would imply.

A co-array may be a pointer or a target. In the absence of pointers, the whole of a co-array can be referenced as an actual argument by use of array syntax such as `P(:)[:]`. If `P` is a co-array pointer, this would refer to the target. We also need a notation for the whole of the pointer, and have chosen to use the name followed an empty pair of square brackets, for example, `P[]`.

The ALLOCATE statement is available for a pointer and the rules of the Section 2.6 apply in this case too. In order not to hide communication inside a pointer, pointer assignment for a co-array pointer must be limited to co-array targets. The pointer must provide both a means to access local data without any square brackets and a means to address data on other images through square brackets. To ensure that this is achieved, we require the pointer and its target to have the same local rank and the same co-rank. For example, the pointer assignment in the code

```
REAL, POINTER :: P(:)[:], T(:,:)[:,:]
 ...
P[] => T(1,:)[:,2]
```

tells us that the sections `P(:)` and `T(1,:)` are identical, as are the sections `P(1)[:]` and `T(1,1)[:,2]`. There is no need for automatic synchronization at a pointer assignment statement, since the statement involves only local data and affects the later action only of the current image.

A co-array pointer is not permitted in a pointer assignment statement without the empty pair of square brackets, that is, both the array and co-array parts must be pointer assigned together.

Appendix 3. The synchronization intrinsics in Co-Array Fortran

In this appendix, we explain how a simple module for `sync_team` and `sync_all` may be constructed in Co-Array Fortran, assuming that the intrinsic `sync_memory` is available. Improved code is accessible by ftp from `jkr.cc.rl.ac.uk` in the directory `pub/caf`.

We identify corresponding calls of `sync_team` by counting, for each pair p and q, the number of invocations $c(p,q)$ for a team including image p on image q. If p and q belong to more than one team, their calls for each team are required to be in the same order, so the calls correspond if and only if $c(p,q)$ and $c(q,p)$ have the same value.

In a long run on a powerful machine, the counts $c(p,q)$ may get very large. We therefore use long integers, declared as with `kind=selected_int_kind(18)`, to hold these counts. On a machine without such integers, extra code will be needed; for example, the counts might be held as a pair of integers.

The basic mechanism we use to force an image to wait for another image is to make it execute a tight loop of code until co-array data is altered by the other image. Here is an example, where we hold the count $c(p,q)$ in the co-array element `c(p)[q]`.

```
do ! Wait for the count on image q to be big enough
   call sync_memory
   if( c(me)[q] >= c(q) ) exit
```

```
  end do
```

Note the presence of `sync_memory` in the loop. This is essential. Without it, the compiler could legitimately make a single local copy of c(me)[q] and execute the loop for ever. On some systems, the loop may slow the execution of other images by the continual reference to data on image `q`. This may be alleviated by inserting some code ahead of the `end do` statement to slow the loop. We will not do this here, but it illustrates that tuning of the code may be needed for efficient working on a particular machine.

We begin with a simple module for `sync_team` with `wait` present. The counts $c(p,q)$ are held in the co-array `c`. This has to be allocatable because the number of images is not a Fortran constant. It means that the initialization subroutine `sync_start` is needed to allocate the co-array and initialize its value. This must be called on all images. We must not allow any image to return until all images have initialized this array. We do this here by another allocate statement, since this causes synchronization.

```
module sync
   implicit none
   save
   integer, private :: this_image, num_images
   integer, parameter :: long=selected_int_kind(18)
   integer(long), allocatable, private :: c(:)[:],dummy(:)[:]
   integer, private :: k,me,nimg,q
contains
   subroutine sync_start ! This must be called initially on all images
      me   = this_image()
      nimg = num_images()
      allocate ( c(nimg)[*] )
      c(:) = 0
      call sync_memory
      allocate ( dummy(0)[*] ) ! Synchronize
  end subroutine sync_start

   subroutine sync_team(team,wait)
        integer, intent(in) :: team(:),wait(:)
        call sync_memory
        c(team) = c(team) + 1
        do k= 1,size(wait)
           q = wait(k)
           do  ! spin waiting for image wait(k)
              call sync_memory
              if( c(me)[q] >= c(q) ) exit
           enddo
        enddo
        call sync_memory
   end subroutine sync_team

 end module sync
```

The following additions provide for the cases where `team` is scalar or `wait` is absent

```
   interface sync_team
      module procedure sync_team, sync_team0, sync_team1,  &
                       sync_team2, sync_team3, sync_team4
   end interface

   subroutine sync_team0(team)
      integer, intent(in) :: team
      call sync_team ((/me,team/))
   end subroutine sync_team0
```

```
   subroutine sync_team1(team)
      integer, intent(in) :: team(:)
      call sync_team(team,team)
   end subroutine sync_team1

   subroutine sync_team2(team,wait)
      integer, intent(in) :: team(:)
      integer, intent(in) :: wait
      call sync_team(team,(/wait/))
   end subroutine sync_team2

   subroutine sync_team3(team,wait)
      integer, intent(in) :: team
      integer, intent(in) :: wait
      call sync_team ((/me,team/),(/wait/))
   end subroutine sync_team3

   subroutine sync_team4(team,wait)
      integer, intent(in) :: team
      integer, intent(in) :: wait(:)
      call sync_team ((/me,team/),wait)
   end subroutine sync_team4
```

The following additions provide for `sync_all`

```
   integer, allocatable, private :: all(:)
   interface sync_all
      module procedure sync_all, sync_all1, sync_all2
   end interface

! In sync_start
   allocate (all(nimg))
   all = (/ (k,k=1,nimg) /)

   subroutine sync_all
      call sync_team (all,all)
   end subroutine sync_all

   subroutine sync_all1(wait)
      integer, intent(in) :: wait(:)
      call sync_team(all,wait)
   end subroutine sync_all1

   subroutine sync_all2(wait)
      integer, intent(in) :: wait
      call sync_team(all,(/wait/))
   end subroutine sync_all2
```

The subroutine `sync_team` may also include checks that `me` is in `team`, that the element of `team` are in range, that there are no repeated values, and that each value in `wait` is in `team`. These checks may be under the control of a public flag in the module, so that they can be omitted in production code when the developer is confident that the errors will not occur.

The number of cycles of the do construct for the wait may be limited, so that deadlock can be detected. If the limit is reached, we recommend that a message be printed showing the index `p` of the executing image, the index `q` of the image for which it is waiting, the team, and the counts $c(p,q)$ and $c(q,p)$.