ISO/IEC JTC1/SC22/WG5 – **N1317**

Co-array Fortran for parallel programming

Robert W. Numrich, Silicon Graphics, Inc.

and

John Reid, Rutherford Appleton Laboratory

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 that 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 call to a procedure that 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.

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.

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. Array indices in square brackets provide an equally convenient notation for accessing objects across images and follow similar rules.

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 data on a single 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.

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 want to apply an approximate Laplacian operator using a five-point stencil to data distributed as the co-array `u(1:nrow)[1:ncol]`, with periodic boundary conditions 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_images() ! Synchronize all images
   new_u(1:nrow)=new_u(1:nrow)+u(1:nrow)[left]+u(1:nrow)[right]
   call sync_images()
   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 neighboring 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 all 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_images` to make sure the values of the array on other images are ready before using them. After adding the values from left and right neighboring images into the local temporary array, we synchronize a second time to make sure all images have obtained the original values before altering the local values. 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
   a(ix,:) = b(:,iz)[ix]
end do
```

The do loop is executed on each image and it ranges over all the images from which data is needed. The one statement in the body is a clear representation of the data transfer.

In the cycle of the loop with `ix=iz`, only local data transfer takes place. This is not an error, 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.

```
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_images()
   if(this_image(great)==1)then
      work = great[:]    ! Gather local maxima
      great[:]=maxval(work) ! Scatter global maximum
   end if
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 operation. 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_images()
      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_images()
      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_images()
   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 scattering 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)[*]
      call sync_images()
      ! Replace x by the result of summing over the co-dimension
      integer p
      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_images()
   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_images( (/(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_images()
   do ! Iterate
      if(this_image()<=num_images()/2) then
         call hydro
      else
         call radiation
      end if
      call sync_images()
        : ! Code that accesses data from both modules and calculates residual
      if(residual<threshold)exit
   end do
end program main
```

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 procedure called `sync_images`.

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 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,*]
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. On a distributed-memory machine with one processor for each image, a co-array may be stored from the same memory address in each 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 are defined to include both the specifications in parentheses and those in square brackets. The **local rank**, **local extents**, **local size**, and **local shape** are defined from the specifications in parentheses. The **co-rank**, **co-extents**, **co-size,** and **co-shape** are defined from the specifications 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 local rank and the co-rank are each limited to seven. The syntax automatically ensures that these are the same on all images.

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 of deferred-shape:

```
subroutine solve(n,a,b)
integer :: n
real :: a(n)[*], b(n)
real,allocatable :: work(:)[:]  ! Deferred shape co-array
```

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. If the co-rank is greater than one, the co-array has no final extent and no final upper bound.

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.

A deferred-shape co-array must be allocatable. 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 co-arrays always commence at one, even for dummy arguments, independent of the bounds specified in their dimension statements. 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).

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.

The **rank**, **extents**, **size,** and **shape** of a co-array subobject are given as for a Fortran 95 array subobject except that we include both the subscript triplets and vector subscripts in parentheses and those in square brackets. The **local rank**, **local extents**, **local size**, and **local shape** are given by ignoring those in square brackets. The **co-rank**, **co-extents**, **co-size**, and **co-shape** are given from those in square brackets. 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 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, subscripts in parentheses are required if the parent has nonzero local rank. For example, `a[:]` is not an acceptable alias for `a(:)[:]`. Furthermore, square brackets may never precede parentheses.

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] = 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_images()
```

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

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`.

3.4 Procedures

A co-array subobject is permitted only in an intrinsic operation or assignment (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:3) = 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 or component selection; any subscript expressions must have the same value on all images.

Note: These rules are intended to ensure that copy-in or copy-out is not needed for a co-array, 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 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
```

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. 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.

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,

```
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.

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_images()`.

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 hidden references to other images, a pointer is permitted to be associated only with targets on its own image. Neither the pointer nor the target in a pointer assignment statement is permitted to be a co-array subobject. For example,

```
z[i]%p => q ! Not allowed
q => z[i]%p ! Not allowed
```

are 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
z = z[i] ! has a pointer component
```

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
```

3.8 Execution control

Since Co-array Fortran is based an SPMD execution model, synchronization plays a vital role in a Co-array Fortran program. If one image is to access data calculated by another, it must wait until the calculation is complete. The intrinsic subroutine `sync_images` is provided for this purpose. In its simplest form, it has no arguments and all images wait until they are all executing such a call. Alternatively, if two images are collaborating, each may call the subroutine with a scalar argument whose value is the index of the other. Finally, if a bigger group is collaborating, each may call the subroutine with an array argument whose value is the set of indices of all the images of the group.

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 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.

`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.

`sync_images(`*[*`image`*]*`)` is a subroutine that synchronizes images. `image` may be an integer scalar, an integer array, or may be absent. This means that there are four cases:

Case (i); If `image` is scalar with a value in the range 1 ≤ `image` ≤ `num_images()` and the image making the call has index `i`, image `i` waits for the image with index `image` to invoke `sync_images` with a scalar argument of value `i`, unless it is already making such an invocation. If it is not making an invocation, the first invocation that it makes must be of this form. It shall not be making a different invocation of `sync_images.` The case with `i` and `image` having the same value is valid.

Case (ii); If `image` is an array with an element of value `this_image()`, the image making the call waits for the images with indices given by the other elements of `image` that lie in the range 1 ≤ `image(`*i*`)` ≤ `num_images()` to invoke `sync_images` with an array argument having an element of value `this_image()`.

Case (iii); If `image` is a scalar with a value outside the range 1 ≤ `image` ≤ `num_images()`, an array without an element of value `this_image()`, or an array with an element of value `this_image()` but without another element with a value in the range 1 ≤ `image(`*i*`)` ≤ `num_images()`, the subroutine has no effect.

Case (iv); if `image` is absent, the image making the call waits for all other images to invoke

`sync_images` with no arguments, unless they are all already making such invocations. If any of them is not making such an invocation, the first invocation of `sync_images` that it makes must be of this form. None of them shall be making a different invocation of `sync_images.`

Note: Implementations that employ optimization techniques involving maintaining copies of variables in registers or other forms of readily accessible storage shall arrange for each image to replace any changed values in the storage that other images access before indicating to other images that it has reached a call of `sync_images`.

`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.

3.10 Input/output

It is assumed that all images reference the same file system. Each image has its own set of independent input/output units. A file may be opened on one image when it is already open on another; only the BLANK=, DELIM=, PAD=, ERR=, and IOSTAT= specifiers may have values different from those in effect on the other image. A file may not be closed with status DELETE if the file is to be connected to a unit on another image. For a unit identified by * in a READ or WRITE statement, there is a single position for all images. Only one image executes a statement for such a unit at any one time, the system introducing delays when necessary. Otherwise, each image positions each file independently. If the access order is important, the program should employ synchronization statements to ensure that two images do not access the same record at the same time. For safe input-output, control constructs may be employed to ensure that no file is ever connected on more than one image at the same time.

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 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_images()
   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_images()
   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
```

The code is slightly more complicated but now each image begins by accessing its neighbour (with wrap-around). 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.

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_images( (/(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_images( (/(i, i=first,last)/) )
   end if
end subroutine greatest
```

Here, we have used the intrinsic `sync_images` 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 processors in $\log_2 np$ stages, where np is the number of processors. 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,mypartner,me,span
   me =  this_image(x)
   span = 1
   do k=1,log2_images()
      if(mod(me-1,2*span)<span)then
         mypartner = me + span
      else
         mypartner = me - span
      end if
      call sync_images()
      work(:) = x(:)[mypartner]
      call sync_images() ! 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()
call sync_images( )
if(me <= rem_images()) then
    x(:) = x(:) + x(:)[m+me]
end if
```

The main loop takes the form:

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

```
call sync_images( )
if(me <= rem_images()) then
    x(:)[m+me] = x(:)
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 *etal.* 1994) 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.

HPF directives on a distributed memory computer 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 adds 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 single extension to the language.

In some cases, especially when code does not translate naturally into data parallel array syntax, this technique 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 *etal.* 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.

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.

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 have its co-shape specified explicitly, or be allocatable. There is implicit synchronization of all images in association with each allocation or deallocation of a co-array.
4. 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.

5. A reference to a co-array without square brackets is a reference to the local object.
6. 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 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.
7. 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.
8. There is an intrinsic subroutine for synchronizing the executing image with another image, a set of other images, or all other images.
9. Co-arrays are not permitted to be pointers, but a co-array may be of a derived type with pointer components. A pointer is not permitted to become associated with a target on another image.
10. Execution of a STOP on any images causes all images to cease execution.
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. Each image has its own set of independent input/output units. For a unit identified by * in a READ or WRITE statement, there is a single position for all images. Otherwise, each image positions each file independently.

7 Acknowledgements

We would like to express our thanks to Dick Hendrickson of Imagine1, Inc. for his careful reading of early drafts of this paper and his many helpful suggestions; to Alan Wallcraft of the Stennis Space Center, Mississippi for his enthusiastic support and many helpful email exchanges; and to Mike Metcalf formerly of CERN, Ian Gladwell of SMU, Dallas and Bill Long of Silicon Graphics for their suggestions after reading 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.

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 within the F^{--} programming model, Accepted for 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 Progamming 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.

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:

```
interface sub
    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. Foe 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.