Co-arrays in the next Fortran Standard

Robert W. Numrich, Minnesota Supercomputing Institute
and
John Reid, JKR Associates

Abstract

The WG5 committee, at its meeting in Delft, May 2005, decided to include co-arrays in the next Fortran Standard. A special issue of Fortran Forum in August 1998 explained the feature, but since many of the details of the version adopted by WG5 differ from the 1998 version, it seems appropriate to describe it afresh. This article will appear in the August 2005 issue of Fortran Forum. It is not an official document and has not been approved by J3 or WG5.

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

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

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

The extension includes intrinsic procedures to synchronize images, to return the number of images, to return the index of the current image, and to perform collective actions.

1 Introduction

Co-arrays were designed to answer the question ‘What is the smallest change required to convert Fortran into a robust and efficient parallel language?’. Our answer is a simple syntactic extension. It looks and feels like Fortran and requires Fortran programmers to learn only a few new rules. These rules are related to two fundamental issues that any parallel programming model must resolve, work distribution and data distribution.

First, consider work distribution. The co-array extension 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. Each replication of the program is called an **image.** The number of images may be the same as the number of physical processors, or it may be more, or it may be less. A particular implementation may permit the number of images to be chosen at compile time, at link time, or at execute time. Each image executes asynchronously and the normal rules of Fortran apply. The execution path may differ from image to image as specified by the programmer who, with the help of a unique image index, determines the actual path using normal Fortran control constructs and explicit synchronizations. For code between synchronizations, the compiler is free to use all its normal optimization techniques as if only one image were present.

Second, consider data distribution. The co-array extension allows the programmer to express data distribution by specifying the relationship among memory images in a syntax very much like normal Fortran array syntax. Objects that are declared without the new syntax exist independently on all the images; each is private, that is, it can be accessed only from its own image. Objects with the new syntax have an important additional property: as well as having access to the local object, each image may access the corresponding object on any other image. For example, the statement

```
real, dimension(1000)[*] :: x,y
```

declares two objects `x` and `y`, each as a whole called a **co-array.** Co-arrays have the same size on each image. In this example, each image has two local real arrays of size 1000. If an image executes the statement:

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

the array `y` from image `q` is copied into local array `x`.

Array indices in parentheses follow the normal Fortran rules within one memory image. Co-array indices in square brackets provide an equally convenient notation for accessing an object on another image. Bounds in square brackets in co-array declarations follow the rules of assumed-size arrays since a co-array exists on all the images. The upper bound for the last co-dimension is never specified, which allows the programmer to write code without knowing the number of images the code will eventually use.

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.

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 physical processor for each image, a co-array may be stored from the same virtual address in each physical processor. On any machine, a co-array may be implemented in such a way that each image can calculate the virtual address of an element on another image relative to the array start address on that other image. An implementation might arrange for each co-array to be stored from the same virtual address in each image, but this is not required.

To use co-array syntax for data structures with different sizes on different images, we may declare a co-array of a derived type with a component that is an allocatable array or a pointer. On each image, the component is allocated locally or is pointer assigned to a local target, so that it has the desired size for that image (or not allocated or pointer assigned, if it is not needed on that image). It is straightforward to access such data on another image, for example,

```
x(:) = z[p]%alloc(:)
```

where the square bracket is associated with the variable `z`, not with its component. In words, this statement means ‘Go to image `p`, obtain the address of the allocated component array, and copy the data in the array itself to the local array `x`’. Data manipulation of this kind is handled awkwardly, if at all, in other programming models. Its natural expression in co-array syntax gives the programmer power and flexibility for writing parallel code.

The co-array feature adopted by WG5 was formerly known as Co-Array Fortran, an informal extension to Fortran 95 by Numrich and Reid (1998). Co-Array Fortran itself was formerly known as F−−, which evolved from a simple programming model for the CRAY-T3D described only in internal Technical Reports at Cray Research in the early 1990s. The first informal definition (Numrich 1997) was restricted to the Fortran 77

language and used a different syntax to represent co-arrays. It was extended to Fortran 90 by Numrich and Steidel (1997) and defined more precisely for Fortran 95 by Numrich and Reid (1998).

Portions of Co-Array Fortran have been incorporated into the Cray Fortran compiler and various applications have been converted to the syntax (see, for example, Numrich, Reid, and Kim 1998 and Numrich 2005). A portable compiling system for a subset of the extension has been implemented by Dotsenko, Coarfa, and Mellor-Crummey (2004). It is called `cafc` and performs source-to-source transformation of co-array code to Fortran 90 augmented with platform-specific communication. One instantiation uses the Aggregate Remote Memory Copy Interface (ARMCI) library for one-sided communication (Nieplocha and Carpenter 1999) and another uses loads and stores for communication on shared-memory machines. Experience with the use of `cafc` is related by Coarfa, Dotsenko, and Mellor-Crummey (2004) and Dotsenko, Coarfa, Mellor-Crummey, and Chavarría-Miranda, D. (2004).

Reid (2005) proposed that co-arrays be included in the revision of Fortran that is planned for 2008. The ISO Fortran Committee agreed in May 2005, subject to some changes (see WG5 (2005)). The rest of this article contains a complete description of the proposal as it now stands. For more extensive discussion and examples, see Numrich and Reid (1998). The appendix contains a summary of the changes since the 1998 definition.

2 Referencing images

Images are normally referenced by co-subscripts enclosed in square brackets. Each set of co-subscripts maps to an **image index**, which is an integer between one and the number of images, in the same way as a set of array subscripts maps to a position in array element order. The underlying run-time system maps an image index onto a physical processor and its associated memory.

The number of images may be retrieved through the intrinsic function `num_images()`. On each image, the image index is available from the intrinsic `this_image()` with no arguments. The set of subscript indices that correspond to the current image for a co-array `z` are available as `this_image(z)`. The image index that corresponds to a set of subscript indices `sub` for a co-array `z` is available as `image_index(z,sub)`. For example, on image 5, for the array declared as

```
real :: z(10,20)[10,0:9,0:*]
```

`this_image()` has the value `5` and `this_image(z)` has the value `(/ 5,0,0 /).` For the same example on image `213`, `this_image(z)` has the value `(/ 3,1,2 /).` On any image, the value of `image_index(z, (/ 5,0,0 /) )` is `5` and the value of `image_index(z, (/3,1,2 /)` is `213`.

3 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[20,*]
```

Unless the array is allocatable (Section 7), the form for the dimensions in square brackets is the same as that for the dimensions in parentheses for an assumed-size array. The total number of subscripts plus co-subscripts

is limited to 15.

The part of a co-array that resides on another image may be addressed by using subscripts in square brackets following any subscripts in parentheses, 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**. Each subscript in square brackets must be a scalar integer expression. Section subscripts are not permitted in square brackets. Subscripts in parentheses must be employed whenever the parent has nonzero local rank. For example, `a[2,3]` is not permitted as a shorthand for `a(:)[2,3]`.

The **rank**, **bounds**, **extents**, **size**, and **shape** of a co-array are given by the data in parentheses in its declaration or allocation. The **co-rank**, **co-bounds**, and **co-extents** are given by the data in square brackets in its declaration or allocation. The co-size of a co-array is always equal to the number of images. The syntax and semantics mirror those of assumed-size arrays: the final extent is always indicated with an asterisk, and a co-array has no final co-extent, no final upper bound, and no co-shape. For example, the co-array declared thus

```
real :: array(10,20)[10,-1:9,0:*]
```

has rank `2`, co-rank `3`, and shape `(/10,20/)`; its lower co-bounds are `1, –1, 0.`

A co-array is not permitted to be a constant. This restriction is not necessary, but the feature would be useless. Each image would hold exactly the same value so there would be no reason to read its value from another image.

To ensure that data initialization is local (the same on each image), co-subscripts 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
```

A co-array may be allocatable as discussed in Section 7.

A co-array is not permitted to be a pointer, but a co-array may be of a derived type with pointer or allocatable components as discussed in Section 8.

A derived type is not permitted to have a co-array component unless the component is allocatable. If an object has a co-array component at any level of component selection, each ancestor of the co-array component must be a non-allocatable, non-pointer scalar. Were we to allow a co-array of a type with 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`.

4 Accessing data objects

Any object reference without square brackets is always a reference to the object on the invoking image. For example, in

```
real :: z(20)[20,*], zmax[*]
    :
zmax = maxval(z)
```

the value of the largest local element of the co-array `z` is placed in the local part of the co-array `zmax`.

For a reference with square brackets, the co-subscript list must map to a valid image index. For example, if there are 16 images and the co-array `z` is declared thus

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

then a reference to `z(:)[1,4]` is valid, since it has co-subscript order value `16,` but a reference to `z(:)[2,4]` is invalid, since it has co-subscript order value `17.` The programmer is responsible for generating valid co-subscripts. The behaviour of a program that generates an invalid co-subscript depends on the implementation by the underlying run-time system.

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

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

```
z[6] = 1
```

is executed by every image that encounters it, not just image 6. If code is to be executed selectively, the Fortran `if` or `case` statement is needed. For example, the code

```
real :: z[*]
  ...
if (this_image()==1) then
   read(*,*) z
   do image = 2, num_images()
      z[image] = z
   end do
end if
call sync_all()
```

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

A co-array subobject is permitted in intrinsic operations, intrinsic assignments, and input/output lists. It is also permitted in non-intrinsic operations and as an actual argument in a procedure call provided the interface is explicit and the dummy argument is not a co-array and has `intent(in)` or the `value` attribute. A local copy of the actual argument is made before execution of the procedure starts.

5 Co-arrays in procedures

A dummy argument of a procedure is permitted to be a co-array.

It may be of explicit shape, assumed size, assumed shape, or allocatable:

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

When the procedure is called, the dummy argument is associated either with the whole of a co-array or with a co-array subobject. The association is with the whole or part of co-array itself and not with a copy. Making a copy is undesirable because it would make synchronization necessary on entry or return to ensure that remote access was not to a copy that does not yet exist or has already been deallocated. Restrictions have been introduced so that copy-in and/or copy-out is never needed. Furthermore, the interface is required to be

explicit so that the compiler can check adherence to the restrictions.

Just the local part of the co-array actual argument is specified and this is associated with the local part of the dummy co-array. It is assumed that the rest of the actual co-array consists of the corresponding parts on the other images. Here is an example

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

The restrictions that avoid copy-in and/or copy-out are:

1. the actual argument must have a designator that has no vector-valued subscripts, allocatable component selection, pointer component selection, or image selection;
2. if an assumed-shape array or a subobject of an assumed-shape array appears as an actual argument corresponding to a dummy co-array, the dummy co-array must be of assumed shape; and
3. if an array section appears as an actual argument corresponding to a dummy co-array that is not of assumed shape, the section must have elements whose subscript order values in its parent array consist of a sequence without gaps.

If a dummy argument is an allocatable co-array, the corresponding actual argument must be an allocatable co-array of the same rank and co-rank. This allows the array to be allocated in the procedure and accessed in the caller after return.

Automatic co-arrays are not permitted. For example, the following code fragment is not permitted

```
subroutine solve3(n)
integer :: n
real :: work(n)[*]  ! Not permitted
```

Were automatic co-arrays permitted, it would be necessary to require image synchronization, both after memory is allocated on entry and before memory is deallocated on return. We would also need rules to ensure that the sizes are the same in all images. Effectively the arrays would be just like allocatables, except for not needing to write allocate syntax.

A function result is not permitted to be a co-array. A co-array function result is like an automatic co-array and is disallowed for the same reasons.

The rules for resolving generic procedure references are based on the local properties and are therefore unchanged. The rules cannot be extended to allow overloading of array and co-array versions since the syntactic form of an actual argument would be the same in the two cases.

A pure or elemental procedure is not permitted to contain any co-array syntax, since this involves side effects.

Unless it is allocatable or a dummy argument, a co-array that is declared in a procedure must be given the `save` attribute. If a co-array is declared in a procedure, with a fixed size but without the `save` attribute, there would need to be an implicit synchronization on entry to the procedure and return from it. Without this, there might be a reference from one image to non-existent data on another image. An allocatable array is not required to have the `save` attribute because a recursive procedure may need separate allocatable arrays at each level of recursion.

A procedure with a non-allocatable co-array dummy argument will usually be called simultaneously on all images with the same actual co-array, but this is not a requirement. For example, the images may be grouped into teams, each of which is either calling the procedure with the same actual co-array or is executing other code.

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

We expect that such a procedure will usually be called simultaneously on all the images of a team with the same actual co-array and the same bounds and co-bounds. We recommend this practice.

6 Storage association

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

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

is not permitted.

Appearance in a `common` or `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. This ban on associating a co-array with an object that is not a co-array is not necessary, but we see it as desirable to keep some separation between objects that are co-arrays and those that are not.

A `common` block that contains a co-array must have 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`.

7 Allocatable co-arrays

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

```
real, allocatable :: a(:)[:], s[:,:]
:
allocate ( a(10)[*], s[-1:34,0:*] )
```

The co-bounds must always be included in the allocate statement and the upper bound for the final co-dimension must always be an asterisk. For example, the following are not permitted (compile-time constraints):

```
allocate( a(n) )     ! Not allowed (no co-bounds)
allocate( a(n)[p] )  ! Not allowed (co-bound not *)
```

Also, the values of all the local bounds are required to be the same on all images. For example, the following is not permitted (run-time constraint)

```
allocate( a(this_image())[*] ) ! Not allowed (varying local bound)
```

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 executing the `allocate` statement. Similarly, for `deallocate,` all images delay making the deallocations until they are all about to execute the `deallocate` statement. This synchronization is independent of those obtained by calling `sync_all` and `sync_team` (see Sections 9 and 11). Without these rules, an image might reference data on another image that has not yet been allocated or has already been deallocated.

When an image executes an allocate statement, no communication is necessarily 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 an allocatable co-array without the `save` attribute there is an implicit deallocation (and associated synchronization) before the procedure in which it is declared is exited by execution of a `return` statement or an `end` statement.

An allocatable co-array may be given the `save` attribute unless separate arrays are needed at each level of recursion in a recursive procedure. For allocation of such a co-array, each image must descend to the same level of recursion or deadlock may occur.

Fortran 2003 allows the shapes to disagree in an intrinsic array assignment to an allocatable array; the system performs the appropriate reallocation. Such disagreement is not permitted for an allocatable co-array, since it would involve synchronization. Similarly, in a Fortran 2003 intrinsic assignment to an object of a type that has an allocatable component at any level of component selection, the component shapes are not required to agree. Such disagreement is not permitted in an intrinsic assignment to a remote object of such a type since this would involve a statement on one image causing an action (allocation) to occur on another image.

8 Array pointers

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

However, a co-array may be of a derived type with pointer or allocatable components. The targets of such components are always local with shapes that may vary from image to image. 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 an allocatable component of a co-array and allocated only on the images on which it is needed.

For example, if co-array `z` contains a pointer component `ptr,` then `z[q]%ptr` is a reference to the target of component `ptr` of `z` on image `q.` This target must reside on image `q` and must have been established by an allocate statement executed on image `q` or a pointer assignment executed on image `q,` for example,

```
z%p => r ! Local association
```

A local pointer can be associated with a target component on the local image,

```
r => z%p ! Local association
```

but cannot be associated with a target component on another image,

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

There is no requirement that the components of a co-array of derived type be allocated or associated or for them to have the same shape on all images.

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

Similarly, for a co-array of a derived type that has a pointer or allocatable component, it is illegal to allocate one of those 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)
```

A co-array is permitted to be of a type that has a procedure pointer component or a type bound procedure. Such a procedure must not be invoked remotely; for example, the statement

```
call a[p]%proc(x)     !  Not allowed
```

is not permitted.

9 Execution control

Most of the time, each image executes on its own as a Fortran program without regard to the execution of other images. It is the programmer's responsibility to ensure that, whenever an image alters the contents of a co-array, no other image might still need the old value or that, whenever an image accesses the contents of a co-array, it is not an old value that has been subsequently updated by another image.

To avoid such memory race conditions, the programmer must invoke intrinsic synchronization procedures. In almost all cases, the programmer will use the procedure `sync_all` or `sync_team` (see Section 11). The subroutine `sync_all` provides a barrier for the important case where all images must synchronize before moving forward. The subroutine `sync_team` provides synchronization for a team of images that need not consist of all the images.

The programmer must make no assumptions about the execution timing on different images. 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 some of the statements between two invocations of `sync_all`, another image might be out of execution.

This obligation has been placed on the programmer to provide the compiler with scope for optimization. When constructing code for execution on an image, it may assume that this 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 compiler.

In particular, if the compiler employs temporary memory to hold an image's co-array data temporarily, such as cache or registers or even packets in transit between images, it must eventually make that data visible to other images. 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 `flush_memory` is provided for both purposes, but direct calls are rarely needed since each invocation of `sync_team` or `sync_all` also has the effect of `flush_memory.`

Note that if data calculated on one image are to be accessed on another, the images must synchronize after the

calculation is complete on the first and before the second accesses the data. Doing this with `sync_team` or `sync_all` will also ensure that the necessary call to `flush_memory` is made on the first before the necessary call to `flush_memory` is made on the second.

Given the fundamental `flush_memory` intrinsic procedure, the other synchronization procedures can be programmed in Fortran (see WG5 (2005), Appendix 1), but the intrinsic versions are likely to be more efficient. In addition, the programmer may use it to express customized synchronization operations in Fortran.

For the important case of one image synchronizing with one other image, `sync_team` may be given the index of the other image as a scalar argument. Here is an example that imposes the fixed order 1, 2, ... on images:

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

If the local part of a co-array or a subobject of it is an actual argument corresponding to a dummy argument that is not a co-array, 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.

Teams are permitted to overlap, but the programmer must ensure that the following rule holds, so that the synchronizations correspond correctly. 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.

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 may improve the overall efficiency. Implementations, however, are not required to cause immediate continued execution. Implementations may choose to wait for the whole team, which certainly more than satisfies the requirement that the members of `wait` have arrived.

Two procedures have been added to support a non-blocking, asynchronous programming style. They effectively split the `sync_all` and `sync_team` procedures into two phases. The procedure `notify_team` registers each team member at a synchronization point but returns without blocking. The image may execute other work and them check the status of the other team members by calling the non-blocking procedure `ready_team,` to see if the other team members have arrived, or the blocking procedure `wait_team,` to wait for the other team members.

The `flush` statement for file I/O plays a similar role for file data to that of `flush_memory` for co-array data. Because of the high overheads associated with file operations, `sync_team` and `sync_all` do not also have the effect of a `flush` statement. 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, `flush` statements on both images are needed.

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 a new construct to delimit a critical section:

```
critical
    :   ! code that is executed on one image at a time
end critical
```

There is an implicit `flush_memory` at the beginning and end of each critical section.

The effect of a `stop` statement or an `end` statement in the main program 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 ahead of the `stop` or `end` statement.

10 Input/output

The co-array feature assumes that all images reference the same file system. Most of the time, each image executes its own read and write statements without regard for the execution of other images. With more than one active image, however, Fortran I/O processing cannot be used without restrictions unless the images reference distinct file systems. We avoid 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 `flush` statement 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 statement plays the same role for I/O buffers as the intrinsic `flush_memory` does for temporary copies of co-array data. Executing a `flush` statement also has the effect of requiring the reloading of I/O buffers in case the file has been altered by another image.

It is possible for several images to be connected on the same unit for sequential output. The processor ensures 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 `flush`.

Invocation of `flush` 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 `flush,` 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 programmer's responsibility to separate the writes by appropriate `flush` statements and image synchronizations. 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 indices of 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 `connect_team.` 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` must have `access=direct` or `action=write`.

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

A file must 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 only on the image with index one. All other pre-connected units have a `connect_team` containing all the images. The input unit identified by *, therefore, is only available on the image with index one.

Before execution of a `close` statement, there is an implied `flush` statement for the unit. There are implied calls to `sync_team` with single argument `connect_team` before and after the implied `flush` statement and before and after the `close`.

Before execution of a `backspace`, `rewind`, and `endfile` statement, there is an implied `flush.` There are implied calls to `sync_team` with single argument `connect_team` before and after the implied `flush` statement and before and after the file positioning statement.

11 Intrinsic procedures

The following intrinsic procedures are added. Only `num_images` is permitted in a specification expression. None are permitted in an initialization expression. We use square brackets [ ] to indicate optional arguments.

11.1 Inquiry function

`num_images()` returns the number of images.

11.2 Image index functions

`image_index(array,sub)` returns the index of the image corresponding to the set of co-subscripts `sub` for co-array `array`.

`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.

`array` is a co-array of any type.

`dim` is scalar integer whose value is in the range $1 \le$ `dim` $\le n$ where n is the co-rank of `array`.

If `array` is absent, the result is the index of the invoking image. If `array` is present and `dim` is absent, the result is an array holding the set of co-subscripts of `array` for data on the invoking image. If `array` and `dim` are present, the result is a scalar holding co-subscript `dim` of `array` for data on the invoking image.

11.3 Synchronization procedures

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

`notify_team(team)` is a subroutine that notifies all team members that it has been called on this image. It also has the effect of `flush_memory`.

`team` is an integer array of rank one with intent `in`. Its values must be in the range $1 \le$ `team(i)` $\le$

`num_images()` with no repetitions.

`wait_team(team)` is a subroutine that causes the image to wait until, for every team member, the team member has called `notify_team` with the image in its team at least as many times as the image has called `notify_team` with the team member in its team. It also has the effect of `flush_memory`.

`team` is an integer array of rank one with intent `in`. Its values must be in the range $1 \le$ `team(i)` $\le$ `num_images()` with no repetitions.

`ready_team(team)` is a function of type default logical that returns the value true it and only if, for every team member, the team member has called `notify_team` with the image in its team at least as many times as the image has called `notify_team` with the team member in its team. It also has the effect of `flush_memory`.

`team` is an integer array of rank one with intent `in`. Its values must be in the range $1 \le$ `team(i)` $\le$ `num_images()` with no repetitions.

`sync_team(team` [,wait]) is a subroutine that synchronizes images. It also has the effect of `flush_memory`.

`team` is of intent `in` and is scalar or of rank one. The scalar case is treated as if the argument were the array `(/ this_image(),team /);` here, `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()` with no repetitions. One element of `team` must have the value `this_image()`.

`wait` is of type integer and is scalar or of rank one. The scalar case is treated as if the argument were the array `(/wait/)`. If `wait` is absent, the effect is as if wait were present and equal to `team`.

The effect of `sync_team(team,wait)` with both arguments of rank one is as if the statements

```
call notify_team(team)
call wait_team(wait)
```

were executed.

`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()) /).` It also has the effect of `flush_memory`.

11.4 Collective functions

A new category of intrinsic function has been introduced. A **collective** function is one that has a co-array argument and, on each image of a team, returns a result of the same shape, each element of which is calculated from the values of the corresponding elements on all the images of the team. The most important case is for a scalar co-array. For example, suppose the rank-one co-arrays `x` and `y` hold vectors and their inner-product is required. It may be found thus:

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

Here, the intrinsic `co_sum` has not been given a team, so the team is taken to be all the images. The intrinsic returns the sum over all the images to them all. The details of how best to do this will vary according to the architecture; for example, it might be done in $\log_2$(`num_images()`) steps, each involving synchronization and exchange of data between pairs of images.

On the other hand, the following invocation of `co_sum`

```
y = co_sum(x)  !  y(i) = sum_p x(i)[p]
```

returns to array `y` on each image the sums $\sum_{p}$ `x(i)[p]`.

If the team does not consist of all the images, it may be specified in the usual way through an integer array of image indices with no repeated values.

The case where the co-array has non-zero rank allows for greater communication efficiency when the same collective operation is required over many vectors each of which is spread across images.

The full list of collective functions is as follows:

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

They correspond in the obvious way to the array intrinsics `all`, `any`, etc.

All the collectives involve synchronization of the images of the team. Roundoff effects may cause the result of `co_sum` and `co_product` to vary between images.

12 Acknowledgements

We would like to express our special thanks to Bill Long of Cray, for his help with many of the detailed changes made since the 1998 report and for his advocacy of co-arrays in the US Fortran Committee J3.

13 References

Coarfa, C., Dotsenko, Y., and Mellor-Crummey, J. (2004). Experiences with Sweep3D implementations in Co-Array Fortran. In Proceedings of the Los Alamos Computer Science Institute 5th Annual Symposium (LACSI, 2004), Santa Fe, USA.

Dotsenko, Y., Coarfa, C., and Mellor-Crummey, J. (2004). A multi-platform Co-Array Fortran compiler. In Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT 2004), Antibes Juan-les-Pins, France.

Dotsenko, Y., Coarfa, C., Mellor-Crummey, J., and Chavarría-Miranda, D. (2004). Experiences with Co-Array Fortran on hardware shared memory platforms. In Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2004), West Lafayette, Indiana, USA.

Nieplocha, J. and Carpenter, B. (1999). ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems, Vol. 1586 of Lecture Notes in Computer Science, pp. 533-546, Springer-Verlag.

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

Numrich, R.W. (2005). Parallel numerical algorithms based on tensor notation and Co-Array Fortran syntax.

Parallel Computing, in press.

Numrich, R. W. and Reid, J. K. (1998). Co-Array Fortran for parallel programming. ACM Fortran Forum (1998), 17, 2 (Special Report) and Rutherford Appleton Laboratory report RAL-TR-1998-060 available as `ftp://ftp.numerical.rl.ac.uk/pub/reports/nrRAL98060.pdf`

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. and Steidel, J. L. (1997). F^{--}: A simple parallel extension to Fortran 90. SIAM News, **30**, 7, 1-8.

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

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

Appendix: Changes from the definition of 1998

Here is a summary of the changes since the definition of Numrich and Reid (1998).

1. WG5 decided that the limit on the rank of arrays should be lifted to 15 and that in the case of a co-array, would apply to the sum of the rank and co-rank.
2. Co-subscripts are now limited to scalars. This is a big change that substantially simplifies the proposal and is consistent with Cray's implementation. We now use 'rank' for what was 'local rank', and similarly for the other terms.
3. The intrinsic function `image_index` has been added.
4. We previously required a dummy co-array to have the same bounds on all images and correspond to the same actual co-array.
5. The `save` attribute was previously acquired automatically when this was a requirement. Now the programmer must declare it.
6. A co-array component of a derived type was previously not allowed.
7. Co-array subobjects are now allowed as actual arguments corresponding to non-co-array arguments of a non-intrinsic procedure.
8. There are additional rules to ensure that copy-in or copy-out does not take place for a co-array.
9. Allocatable dummy co-arrays are permitted in view of allocatable dummy arguments being permitted in Fortran 2003.
10. Co-arrays of a type with an allocatable component permitted since such a type is permitted in Fortran 2003.
11. All the collective functions are new.
12. To simplify the rules, we now require that if one image executes an `allocate` statement the others execute the same statement in synchronization. A similar rule applies to the `deallocate` statement.
13. For an allocatable co-array without the `save` attribute there is an implicit deallocation (and associated

synchronization) before exit from the procedure in which it is declared.

14. Actual arguments associated with co-array dummy arguments were more severely restricted.
15. A procedure pointer component or a type bound procedure is not permitted to invoke a remote procedure.
16. The critical section replaces the two intrinsic subroutines, `start_critical` and `end_critical`.
17. Previously, invocations of `flush_memory` were implicitly placed around any procedure invocation that might involve any reference to `flush_memory`. This has been removed for the sake of efficiency since it is often not required.
18. The `flush` statement was not part of Fortran 95, so there was an intrinsic `sync_file` for this purpose.
19. `close`, `backspace`, `rewind`, and `endfile` used to have a `team` specifier that was required to be the same as the connect team.
20. The functions `log2_images()` and `rem_images()` have been removed since these can be coded in Fortran, for example:

```
integer function log2_images()
! Returns the base-2 logarithm of the number of images,
! truncated to an integer.
   log2_images = log(num_images()+0.5)/log(2.0)
end function log2_images
```

21. `flush_memory` was previously called `sync_memory`.
22. `notify_team`, `wait_team`, and `image_index` are new.