ISO/IEC JTC1/SC22/WG5 N1669

Co-arrays in the next Fortran Standard

John Reid, JKR Associates, UK

January 17, 2007

Abstract

The WG5 committee, at its meeting in Delft, May 2005, decided to include co-arrays in the next Fortran Standard. A Fortran Forum article in August 2005 explained the feature, but since many of the details have been changed since then, it seems appropriate to describe it afresh. This article 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. Any occurrence of square brackets is a warning about communication between images.

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

The extension includes statements for synchronizing images and intrinsic procedures to return the number of images, to return the index of the current image, and to perform collective 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 sequence 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 with the new syntax have an important 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 **co-array**. A co-array is either named or a component of a scalar structure that is not a co-array. Note that a subobject of a co-array is not called a co-array. A co-array always has the same shape on each image. In this example, each image has two real co-arrays of size 1000. If an image executes the statement:

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

the co-array `y` on image `q` is copied into co-array `x` on the executing image.

Array indices in parentheses follow the normal Fortran rules within one 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 always 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 memory of the executing image. Since most references to data objects in a parallel code should be local, co-array syntax should appear only in isolated parts of the code. If not, the syntax acts as a visual flag to the programmer that too much communication among images may be taking place. It also acts as a flag to the compiler to generate code that avoids latency whenever possible.

On a shared-memory machine, a co-array on an image and the corresponding co-arrays on other images may be implemented as a sequence of arrays with evenly spaced starting addresses. 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.

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

The co-array feature adopted by WG5 was formerly known as Co-Array Fortran, an informal extension to Fortran 95 by Numrich and Reid (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, Numrich 2005a, and Numrich 2005b). 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 (2004).

Reid (2005) proposed that co-arrays be included in the revision of Fortran that is planned for 2008 (which we will call Fortran 2008 in this report). The ISO Fortran Committee agreed to include co-arrays in May 2005, but made some changes and further changes have been made since then by J3, the Primary Development Body for Fortran. 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).

2 Referencing images

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

Unless the co-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.

A co-array 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-indexed object**. 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 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 co-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 access its value on 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 9.

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, non-co-array 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`.

It is not permissible to add a co-array component by type extension unless the type already has one or more co-array components.

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 element of the co-array `z` on the executing image is placed in the co-array `zmax` on the executing image.

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 is undefined.

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[*]
  ...
sync all
if (this_image()==1) then
   read(*,*) z
   do image = 2, num_images()
      z[image] = z
   end do
end if
sync all
```

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

A co-indexed object 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. On a distributed-memory machine, it is likely that a local copy of the actual argument will be made before execution of the procedure starts and if necessary copied back on return. If the actual argument is a co-indexed object and is allocatable or has an allocatable ultimate component, the dummy argument must have the intent `in` or the `value` attribute; this ensures that allocatables are not allocated remotely. Also, because of the likelihood of copying, the dummy argument must not have the `asynchronous` or `volatile` attribute.

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 corresponding actual argument must be a co-array or a subject of a co-array. The association is with the whole or part of the co-array itself and not with a copy. Making a copy is undesirable because it would make synchronization necessary on entry and 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. 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 subscript, no selection of an allocatable component unless it is a co-array, no selection of a pointer component, and no image selection; and

2. if the dummy argument is a co-array that has the `contiguous` attribute (to be added in Fortran 2008) or is not of assumed shape, the actual argument must be contiguous (in a way that can be verified at compile time by the compiler).

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-array 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-array 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.

A function result is not permitted to be a co-array or to be of a type that has a co-array component at any level of component selection. 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 define a co-indexed object or contain any image control statements (Section 10.9), since these involve side effects. However, it may reference the value of a co-indexed object.

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 were 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 co-array is not required to have the `save` attribute because a recursive procedure may need separate allocatable arrays at more than one level of recursion.

Note that in Fortran 2008, all data declared in a module, including co-arrays, automatically have the `save` attribute.

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 two teams and the images of one team may calling the procedure with one co-array while the images of the other team are calling the procedure with another co-array or are executing different code.

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

6 Storage association

Co-arrays are not permitted in `common` and `equivalence` statements.

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 value of each bound, co-bound, or length type parameter is 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)
```

Furthermore, the dynamic types must 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 executing the same `allocate` statement. Similarly, for `deallocate`, all images delay making the deallocations until they are all about to execute the same `deallocate` statement. Without these rules, an image might reference data on another image that has not yet been allocated or has already been deallocated.

When an image executes an `allocate` statement, no communication is necessarily involved apart from any required for synchronization. The image allocates the co-array and records how the corresponding co-arrays on other images are to be addressed. The compiler, except perhaps in debug mode, is not required to enforce the rule that the bounds and co-bounds are the same on all images. Nor is the compiler responsible for detecting or resolving deadlock problems.

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

8 Restrictions on intrinsic assignment for co-arrays

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 an intrinsic assignment for a scalar of a derived type with an allocatable co-array component, no disagreement of allocation status or shape is permitted for the co-array component. This, of course, can be checked by the system on the executing image.

For the same reason, intrinsic assignment is not permitted to a polymorphic co-array, a subobject of a polymorphic co-array, or a polymorphic co-indexed object.

Intrinsic assignment to a co-indexed object of a type with an allocatable component at any level of component selection is not permitted because it may cause an action (allocation) to occur on another image.

9 Array pointers

A co-array is not permitted to be a pointer. Furthermore, because an object of type `c_ptr` or `c_funptr` has the essence of a pointer, a co-array is not permitted to be of either of these types.

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.

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. Numrich used this technique to build an object-based parallel library for Co-Array Fortran (Numrich 2005b).

For example, if co-array `z` contains a pointer component `ptr`, `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)
```

If such an association would otherwise be implied, the pointer becomes undefined. For example, this happens with the derived-type intrinsic assignments

```
z[q] = z ! The pointer component of z[q] becomes undefined
z = z[q] ! The pointer component of z becomes undefined
```

when executed on an image with an index other than `q`.

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. A remote procedure reference is regarded as local; for example, the statement

```
call a[p]%proc(x)
```

finds the target procedure on image `p` and calls this procedure on the executing image.

10 Synchronization

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 and 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 program must execute synchronization statements.

10.1 `sync all` statement

In most cases, the programmer will use the `sync all` statement for synchronization. This provides a barrier for the important case where all images must synchronize before moving forward. Any statement executed before the barrier on image P is also executed before any statement executed after the barrier on image Q. The normal rules relating to execution order apply. In particular, if the value of a co-array variable is changed by image P before the barrier, it is accessible to image Q after the barrier.

Figure 1: Read data on image 1 and broadcast it.

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

Figure 1 shows a simple example of the use of `sync all`. Image 1 reads data and broadcasts it to other images. The first `sync all` ensures that image 1 does not interfere with any previous

use of `p` by another image. The second `sync all` ensures that another image does not access `p` before the new value has been set by image 1.

Although usually the synchronization will be at the same `sync all` statement on all images, this is not a requirement. The additional flexibility may be useful, for example, when different images are executing different code and need to exchange data from time to time.

There is no implicit `sync all` at the start of the main program. If the code relies on the initialization of a co-array variable on another image, a `sync all` statement is needed.

10.2 Execution segments

There are other statements that allow the programmer to control execution order between images. Each such statement is called an **image control statement** and they are listed in Section 10.9.

On each image, the sequence of statements executed before the first execution of an image control statement, between the execution of two image control statements, or after the last execution of an image control statement is known as a **segment**. The segment executed immediately before the execution of an image control statement includes the evaluation of all expressions within the statement.

For example, in Figure 1, each image executes a segment before executing the first `sync all` statement, executes a segment between executing the two `sync all` statements, and executes a segment after executing the second `sync all` statement.

On each image P, the statement execution order determines the segment order, P_i, i=1, 2, ... Between images, the execution of corresponding image control statements on images P and Q at the end of segments P_i and Q_j may ensure that either P_i precedes Q_{j+1}, or Q_j precedes Q_{i+1}, or both.

A consequence is that the set of all segments on all images is partially ordered: the segment P_i precedes segment Q_j if and only if there is a sequence of segments starting with P_i and ending with Q_j such that each segment of the sequence precedes the next either because they are on the same image or because of the execution of corresponding image control statements.

A pair of segments P_i and Q_j are called **unordered** if P_i neither precedes nor succeeds Q_j.

For example, if the middle segment of Figure 1 is P_i on image 1 and Q_j on another image Q, P_{i-1} precedes Q_{j+1} and P_{i+1} succeeds Q_{j-1}, but P_i and Q_j are unordered.

There are restrictions (see next paragraph) on what is permitted in a segment that is unordered with respect to another segment. These provide the compiler with scope for optimization. When constructing code for execution in a segment or part of a segment, it may assume that this image is the only one in execution and thus it may use all the optimization techniques normally available to a Fortran compiler.

A scalar co-variable that is of type default integer, default logical, default real, or default bits (a Fortran 2008 feature), and has the `volatile` attribute may be referenced during the execution

of a segment that is unordered relative to the execution of a segment in which the co-variable is defined. Otherwise,

- if a co-array variable is defined on an image in a segment, it must not be referenced, defined, or become undefined in a segment on another image unless the segments are ordered,

- if the allocation of an allocatable subobject of a co-array or the pointer association of a pointer subobject of a co-array is changed on an image in a segment, that subobject shall not be referenced or defined in a segment on another image unless the segments are ordered, and

- if a procedure invocation on image P is in execution in segments P_i, P_{i+1}, ..., P_k and defines a non-co-array dummy argument, the argument associated entity shall not be referenced or defined on another image Q in a segment Q_j unless Q_j precedes P_i or succeeds P_k (because a copy of the actual argument may be passed to the procedure).

The restriction on a volatile variable that is permitted to be used in this way to a scalar of one of four simple types is included because memory updates/references to such a variable need to be atomic: referencing the value on one image concurrently with an update on another will either get the previous value or the new value. Such atomic memory operations cannot be guaranteed in general.

10.3 Working in teams

A team of images is defined by the value of a scalar of type `image_team` of the intrinsic module `ISO_Fortran_env`. It must be established by executing the intrinsic subroutine `form_team` (Section 13.3) on every image of the team. The intrinsic `team_images` (Section 13.4) can be used to determine the images in the team of a scalar of type `image_team`. All the components of the type are `private` and none is a co-array. It has default initialization to a value that identifies an empty team. The intention is to allow the processor to calculate optimized communication patterns during the call of `form_team` and store them for all subsequent team actions. It is quite likely that different images would need different data, so it is inappropriate to copy values between images. For this reason, a co-array is not permitted to be of type `image_team`.

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

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.

10.4 The `sync images` statement

For greater flexibility, the `sync images` statement performs a synchronization of the image that executes it with each of the other images in a set of images that it specifies. Executions of `sync images` statements on images M and T correspond if the number of times image M has executed a `sync images` statement with T in its image set is the same as the number of times image T has executed a `sync images` statement with M in its image set. The segments that executed before the `sync images` statement on either image precede the segments that execute after the corresponding `sync images` statement on the other image. Here is an example that imposes the fixed order 1, 2, ... on images:

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

The image set is specified in parentheses as an integer scalar holding an image index, an integer array of rank one holding distinct image indices, or an asterisk to indicate all images.

Execution of a `sync images(*)` statement is not equivalent to the execution of a `sync all` statement. `sync all` causes all images to wait for each other. `sync images` statements are not required to specify the same image set on all the images participating in the synchronization. In the following example, image one will wait for each of the other images to reach the `sync images(1)` statement. The other images wait for image one to set up the data, but do not wait on any of the other images.

```
if (this_image() == 1) then
    ! Set up co-array data needed by all other images
   sync images(*)
else
   sync images(1)
   ! Use the data set up by image one
end if
```

10.5 The `notify` and `query` statements

The `notify` and `query` statements support a non-blocking, asynchronous programming style. They enable better load balancing by allowing one image to proceed with calculations while another is catching up. Each specifies an image set in the same way as a `sync images` statement.

Figure 2 shows an alternative to the code of Figure 1. Instead of image 1 doing the broadcast, it notifies the other images once it has read `p` and each copies the value after it has been notified. Meanwhile, image 1 continues to execute. Here, we have assumed that in the previous and subsequent code each image accesses only its own value of `p`.

Figure 2: Using `notify` and `query` to read data on image 1 and broadcast it.

```
real :: p[*]
  ...
if (this_image()==1) then
   read (*,*) p
   notify(*)
 else
   query(1)
   p = p[1]
 end if
```

This form of the `query` statement is called **blocking**: the image waits for the notification before continuing execution. An alternative has the form

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

which allows the image to do other work instead of waiting idle. A `query` statement is said to be **satisfied** on completion of its execution if it has no `ready=` specifier or if it set its `ready=` variable to true. If the `ready=` value is false, we expect the image to execute further `query` statements, continuing until it executes a satisfied `query` statement.

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

A `query` statement on image Q with a `ready=` variable sets its value to true if and only if $N_{P,Q}>S_{Q,P}$ for all images P in its image set. A `query` statement on image Q without a `ready=` variable delays until $N_{P,Q}>S_{Q,P}$ for all images P in its image set.

Segments on an image executed before the execution of a `notify` statement precede the segments on other another image that follow the corresponding `query` statement on that other image.

10.6 The `sync memory` statement

The `sync memory` statement provides a means of dividing a segment on an image into two segments, each of which can be ordered in some user-defined way with respect to segments on other images. Unlike the other image control statements, it does not have any in-built synchronization effect. In case there is some user-defined ordering between images, the compiler will probably avoid optimizations involving moving statements across the `sync memory` statement and will ensure that any changed data that the image holds in temporary memory such as cache or registers or even packets in transit between images, is made visible to other images. Also, any data from other images that is held in temporary memory will be treated as undefined until it is reloaded from its host image.

For example, consider the code in Figure 3, executed on images `p` and `q`. The do loop is known

Figure 3: Spin-wait loop

```
logical,volatile :: locked[*] = .true.
integer :: iam, p, q
:
iam = this_image()
if (iam == p) then
   sync memory
   locked[q] = .false.
   sync memory
else if (iam == q) then
   do while (locked); end do
   sync memory
end if
```

as a spin-wait loop. Once image `q` starts executing it, it will continue until it finds the value `.false.` for `locked`. The `volatile` attribute ensures that the value is retested on each loop execution. The effect is that the segment on image `p` ahead of the first `sync memory` statement precedes segment on image `q` that follows the third `sync memory` statement. The second `sync memory` statement encourages the compiler to release the lock at once rather than later in the segment that follows.

Note that the segment P_i in which the lock is released is unordered with respect to the segment Q_j in which it is tested. This is permissible by the rules in the penultimate paragraph of Section 10.2 since its type is default logical. The reason for this is discussed in the final paragraph of Section 10.2.

Given the fundamental `sync memory` statement, the effects of the other synchronizations can be programmed in Fortran as procedures (see WG5 (2005), Appendix 1), but the statements are likely to be more efficient. In addition, the programmer may use the `sync memory` statement to express customized synchronization operations in Fortran.

10.7 `stat=` and `errmsg=` specifiers in synchronization statements

All the synchronization statements, that is, `sync all`, `sync team`, `sync images`, `sync memory`, `notify`, and `query`, have optional `stat=` and `errmsg=` specifiers. They have the same role for these statements as they do for `allocate` and `deallocate` in Fortran 2003.

10.8 Critical sections

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**. There is a new construct to delimit a critical section:

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

No image control statement may be executed during the execution of a critical construct, that is, the code executed is a segment.

If image T is the next to execute the construct after image M, the segment in the critical section on image M precedes the segment in the critical section on image T.

10.9 The image control statements

The full list of image control statements is

- `sync all` statement;
- `sync team` statement;
- `sync images` statement;
- `sync memory` statement;
- `notify` statement;
- `query` statement;
- `allocate` or `deallocate` statement involving a co-array;
- `critical` or `end critical` statement;
- `open` statement with a `team=` specifier;
- `close` statement;
- `end`, `end block`, or `return` statement that involves an implicit deallocation of a co-array;
- `end program` statement;
- `call` statement for a collective subroutine (13.5).
- `call` statement for the subroutine `form_team` (13.3).

All of the image control statements include the effect of executing a `sync memory` statement.

11 Program termination

It seems natural to allow all images to continue executing until they have all executed a `stop` or `end program` statement, provided none of them encounters an error condition that may be expected to terminate its execution. On the other hand, if such an error condition occurs on one image, the computation is flawed and it is desirable to stop the other images as soon as is practicable.

For this reason, termination is separated into 'normal termination' and 'error termination'. An image initiates normal termination if it executes a `stop` or `end program` statement. It does not complete termination until all other images have initiated normal or error termination, which allows its data to remain accessible to the other images.

On the other hand, an image initiates error termination if it executes a statement that would cause the termination of a single-image program and is not a `stop` or `end program` statement. This causes all other images that have not already initiated termination to initiate error termination. Within the performance limits of the processor's ability to send signals to other images, this propagation of error termination should be immediate. The exact details are intentionally left processor dependent.

The statement

```
all stop
```

has been introduced. When executed on one image, it initiates error termination there and hence causes all other images that have not already initiated termination to initiate error termination. It thus causes the whole calculation to stop as soon as is practicable.

The full list of causes for the initiation of error termination is

- an `all stop` statement is executed,
- an error condition occurs during execution of an `allocate`, `deallocate`, `sync all`, `sync team`, `sync images`, `sync memory`, `notify`, or `query` statement without a `stat=` specifier,
- an error occurs during execution of an `open`, `close`, `read`, `write`, `backspace`, `endfile`, `rewind`, `flush`, `wait`, or `inquire` statement without an `iostat=`, `end=`, or `err=` specifier,
- an error occurs during execution of a `print` statement,
- an error occurs during execution of the `execute_command_line` intrinsic subroutine and the optional `cmdstat` argument is not present,
- an error occurs during execution of the `form_team` intrinsic subroutine and the optional `stat` argument is not present,
- an error condition occurs by means outside Fortran.

12 Input/output

Most of the time, each image executes its own read and write statements without regard for the execution of other images.

It is possible for several images to have the same unit connected to the same file. The set of images with the common connection is called a **connect team**. It may be specified in the `open` statement by a variable of type `image_team` in a `team=` specifier. The value must have been set by an invocation of the intrinsic `form_team` and the executing image must be a member of the team. The same `open` statement must be executed on all images of the team and there is an implicit team synchronization. If there is no `team=` specifier in an `open` statement, the connect team is the executing image.

If an `open` statement specifies a connect team of more than one image, the connection must either be direct access or be sequential access with `action='write'`. The main mechanism for parallel i/o is the direct-access file where the programmer has arranged that the same record is never accessed by more than one image without an intervening synchronization.

If the connection is not for direct access, 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 `backspace`, `rewind`, and `endfile` statements are not permitted for such a unit.

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

The processor is permitted to hold the data in a buffer and transfer several whole records on execution of `flush`. The `flush` statement ensures that any changed records in buffers are copied to the file itself or to a replication of the file that other images access. This statement plays the same role for I/O buffers as the statement `sync 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.

Execution of a `flush` statement is required only when a record written by one image is read by another or when the relative order of writes from images is important. 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.

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 teams for the units have no images in common.

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

13 Intrinsic procedures

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

13.1 Inquiry functions

`num_images()` returns the number of images as a default integer scalar.

`co_lbound(co_array[,dim,kind])` returns the lower co-bounds of a co-array in just the same way as `lbound` returns the lower bounds of an array.

`co_ubound(co_array[,dim,kind])` returns the upper co-bounds of a co-array in just the same way as `ubound` returns the upper bounds of an assumed-size array (no upper co-bound in the final co-dimension).

13.2 Image index functions

`image_index(co_array,sub)` returns the index of the image corresponding to the set of co-subscripts for `co_array` as a default integer scalar.

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

`sub` is a rank-one integer array of size equal to the co-rank of `co_array`.

`this_image()` returns the index of the invoking image as a default integer scalar.

`this_image(co_array[,dim])` returns the set of co-subscripts of `co_array` that denotes data on the invoking image.

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

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

If `dim` is absent, the result is a default integer array of rank one and size equal to the co-rank of `co_array`; it holds the set of co-subscripts of `co_array` for data on the invoking image. If `dim` is present, the result is a default integer scalar holding co-subscript `dim` of `co_array` for data on the invoking image.

13.3 Subroutine that forms a team of images

`form_team(team,images[,stat,errmsg])` is a subroutine that forms a team of images. There is an implicit team synchronization after the team has been formed.

`team` is a scalar of type `image_team` with intent `out`.

`images` is an integer array of rank one with intent `in`. Its values must be in the range 1, 2, ..., `num_images()` with no repetitions. It must not be of size zero. It must specify the same set of images on all images of the team and this must include the executing image.

`stat` is a scalar default integer with intent `out`. It is assigned the value -1 if any of the images of the team has initiated image termination, or a processor-dependent positive value if an error condition occurs. Otherwise it is assigned the value 0.

`errmsg` is a scalar of type default character with intent `inout`. If an error condition occurs, it is assigned a processor-dependent explanatory message. Otherwise, it is unchanged.

13.4 Transformational function

`team_images(team)` returns a list of images in a team.

`team` is a scalar of type `image_team`.

The result is a default integer array of rank one and size equal to the number of images in the team. It holds the image indices in increasing order.

13.5 Collective subroutines

A new category of intrinsic subroutine, the **collective subroutine**, has been introduced. Each has a leading 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. An optional argument specifies 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))
call co_sum(local_prod,prod)
```

Here, no team is specified in the call of `co_sum`, 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`

```
call co_sum(x,y)  !  y(:) = sum_p x(:)[p]
```

returns to array `y` on each image the sum $\sum_p$`x(:)[p]`.

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 same statement must be used to call a collective on all images of the team. Each call involves synchronization of the images of the team.

The full list of collective subroutines is as follows:

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

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

`mask` or `co_array` is an intent `in` co-array argument with the same shape on every image of the team.

`result` is an intent `out` argument with the same shape as `mask` or `co_array`.

`team` is a scalar intent `in` argument of type `image_team` that specifies the team. The value must have been formed by a call of `form_team` on the executing image. If absent, the team consists of all images.

Roundoff effects may cause the result of `co_sum` and `co_product` to vary between images.

14 Acknowledgements

I would like to express 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; and I would like to thank Bill Long, Aleks Donev, and Bob Numrich for carefully reading drafts of this document and suggesting corrections and improvements.

15 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. (2005a). Parallel numerical algorithms based on tensor notation and Co-Array Fortran syntax. *Parallel Computing*, 31, pp. 588-607.

Numrich, R.W. (2005b). A Parallel Numerical Library for Co-Array Fortran. *Parallel Processing and Applied Mathematics: Proceedings of the Sixth International Conference on Parallel Processing and Applied Mathematics (PPAM05)*, pp. 960-969, Springer Lecture Notes in Computer Science, LNCS 3911.

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. and Reid, J. K. (2005). Co-arrays in the next Fortran Standard. ACM Fortran Forum (2005), 24, 2, 2-24 and WG5 paper

`ftp://ftp.nag.co.uk/sc22wg5/N1601-N1650/N1642.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), Umea University, Umea 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`

16 Appendix: Changes from the Fortran Forum article of 2005

Here is a summary of the main changes since the report of Numrich and Reid (2005).

1. The term 'co-array' now refers to the object on the executing image, rather than the collection of objects on all the images. The Standard was not consistent and this interpretation simplifies the text.

2. All the collective functions are now subroutines. This change was made because of difficulties with synchronization when more than one collective appears in a statement.

3. All the synchronization intrinsic procedures have been replaced by corresponding statements. These statements all have `stat=` and `errmsg=` specifiers. `flush_memory` is now called `sync memory`.

4. The default input file is connected on image 1 only. Other preconnected files have a connect team of all images, but reading is permitted only on image 1.

5. It is no longer assumed that all images access the same file system and I/O units are no longer shared by all images.

6. An explicit interface is no longer needed when a co-indexed object is an actual argument. If it is allocatable or has an allocatable ultimate component, the dummy argument must have intent `in` or the `value` attribute.

7. A co-indexed object is allowed to be of a type with pointer components. A pointer component becomes undefined if its value is copied from a associated pointer on another image. The old rules did not stop a pointer appearing to have a target on another image. It was felt that it would be better to be less restrictive and define the effect.

8. Intrinsic assignment to a co-indexed object of a type with an allocatable component at any level of component selection is not permitted. Previously, the component shapes were required to agree which is at variance for the rule for arrays.

9. The derived type `image_team` has been introduced, with the intrinsic `form_team`. It is used to specify a team in a more controlled way than with an integer array.

10. Significant changes re termination have been made. Section 11 has been added to this report to explain how it now works.

11. Co-arrays are not permitted in `common` and `equivalence` statements.

12. `co_lbound` and `co_ubound` have been added.

13. A remote procedure reference is regarded as local.

14. The concepts of image control statments and the execution segment have been introduced and restrictions added for a segment that is unordered with respect to another segment.