ISO/IEC JTC1/SC22/WG5 N1824

Coarrays in the next Fortran Standard

John Reid, JKR Associates, UK

April 21, 2010

Abstract

Coarrays will be included in the next Fortran Standard, known informally as Fortran 2008. This is about to be submitted for international ballot as a Draft International Standard (see WG5, 2010) and is expected to be published later this year. Only extremely minor changes, such as the correction of typos, are permitted after such a ballot. This article provides an informal description of the core coarray features that will be included. There are plans for further features to be defined in a Technical Report. Only the core set is described here.

A Fortran program containing coarrays 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 objects, so code that can run independently is uncluttered. Any occurrence of square brackets indicates communication between images, which might be slow.

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, among other things, the number of images and the index of the current image.

This article is not an official document and has not been approved by PL22.3 (formerly J3) or WG5.

Contents

1 Introduction

The coarray programming model is 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 coarray 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 almost all its normal optimization techniques as if only one image were present. In particular, code motion optimizations may be applied within such code.

Although this is not required, it is anticipated that in early implementations each image will execute the same executable code (e.g., `a.out` or `.exe` file) on identical hardware.

Second, consider data distribution. The coarray 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 statements

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

declare three objects, each as a **coarray**. `x` and `y` are array coarrays and `z` is a scalar coarray. A coarray always has the same shape on each image. In this example, each image has two real array coarrays of size 1000 and a scalar coarray. If an image executes the statement:

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

the coarray `y` on image `q` is copied into coarray `x` on the executing image.

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

The programmer uses coarray syntax only where it is needed. A reference to a coarray with no square brackets attached to it is a reference to the object in the memory of the executing image. Since it is desirable for most references to data objects in a parallel code to be local, coarray syntax should appear only in isolated parts of the code. Coarray syntax acts as a visual flag to the programmer that communication among images might 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 coarray on an image and the corresponding coarrays on other images might be implemented as a sequence of objects with evenly spaced addresses. On a distributed-memory machine with one physical processor for each image, a coarray might be stored at the same virtual address in each physical processor. On any machine, a coarray may be implemented in such a way that each image can calculate the virtual address of the coarray on another image. If it is an array coarray, each image can calculate the virtual address of an element on another image relative to the array start address on that other image.

Because coarrays are integrated into the language, remote references automatically gain the services of Fortran's basic data capabilities, including

- the typing system,
- automatic type conversions in assignments,
- information about structure layout, and
- object-oriented features with some restrictions.

The coarray 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 coarrays. 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 2005, and Numrich 2006). Since October 2008, the g95[1] compiler has supported coarrays as an extension of Fortran 95. This extension is as defined by the First Committee Draft of Fortran 2008 (WG5 2008). Coarray programs run directy on shared-memory symmetric-multiprocessor (SMP) machines. Coarray programs may also be run across networks with the support of the "G95 Coarray Console" program.

Reid (2005) proposed that co-arrays be included in the next revision of Fortran, now known as Fortran 2008. The ISO Fortran Committee agreed to include co-arrays in May 2005, but made some changes (see Numrich and Reid, 2005) and further changes have been made since then (see, for example, Numrich and Reid, 2007). The hyphen has since been removed from "co-array" (by analogy with cosine, cotangent, etc.) and the features are now split between a core set that is intended to be part of Fortran 2008 and further features that will be defined by

[1] `http://www.g95.org/`

a Type 2 Technical Report. The rest of this article contains a complete description of the core set.

2 Referencing images

Data objects on other images are referenced by cosubscripts enclosed in square brackets. Each valid set of cosubscripts maps to an **image index**, which is an integer between one and the number of images, in the same way as a valid 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()`. The intrinsic function `this_image()` with no arguments returns the image index of the invoking image. The set of cosubscripts that corresponds to the invoking image for a coarray `z` are available as `this_image(z)`. The image index that corresponds to an array `sub` of valid cosubscripts for a coarray `z` is available as `image_index(z,sub)`. For example, on image 5, for the array coarray 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 The properties of coarrays

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 **codimensions** in square brackets, for example:

```
real, dimension(20), codimension[20,*] :: a ! An array coarray
real  :: c[*], d[*]                         ! Scalar coarrays
character :: b(20)[20,0:*]
integer :: ib(10)[*]
type(interval) :: s[20,*]
```

Unless the coarray is allocatable (Section 9), 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 cosubscripts is limited to 15.

A subobject of a coarray is regarded as a coarray if and only if it has no cosubscripts, no vector subscripts, no allocatable component selection, and no pointer component selection. For example, `a(1)` and `a(2:10)` are coarrays if `a` is the coarray declared at the start of this section. The restrictions make a coarray subobject suitable for associating as an actual argument with a dummy coarray (see Section 5) since they ensure that copy-in copy-out is not needed (this would require synchronization). The term **whole coarray** is used for the whole of an object that is declared as a coarray or the whole of a coarray component of a structure.

If a whole coarray is an array, its **rank**, **bounds**, **extents**, **size**, and **shape** are given by the data in parentheses in its declaration or allocation. The lack of such data indicates a scalar coarray, which has rank zero and no bounds, extents, size, or shape.

The **corank**, **cobounds**, and **coextents** of a whole coarray are given by the data in square brackets in its declaration or allocation. Any subobject of a whole coarray that is a coarray has the corank, cobounds, and coextents of the whole coarray. The cosize of a coarray is always equal to the number of images. The syntax mirrors that of an assumed-size array in that the final upper bound is always indicated with an asterisk, but a coarray has a final coextent, a final upper cobound, and a coshape, all of which depend on the number of images. For example, when the number of images is 128, the coarray declared thus

```
real :: array(10,20)[10,-1:8,0:*]
```

has rank `2`, corank `3`, shape `(/10,20/)`; coshape `(/10,10,2/)`; its lower cobounds are `1, -1, 0` and its upper cobounds are `10, 8, 1`.

A coarray is not permitted to be a named constant, because this 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 each image initializes its own data, cosubscripts 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 coarray may be allocatable, see Section 9.

A coarray is not permitted to be a pointer, but a coarray may be of a derived type with pointer or allocatable components, see Section 10. Furthermore, because an object of type `c_ptr` or `c_funptr` has the essence of a pointer, a coarray is not permitted to be of either of these types.

4 Accessing coarrays

A coarray on another image may be addressed by using cosubscripts 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 **coindexed 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 coarray has nonzero rank. For example, `a[2,3]` is not permitted as a shorthand for `a(:)[2,3]`.

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

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

the value of the largest element of the array coarray `z` on the executing image is placed in the scalar coarray `zmax` on the executing image.

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

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

then a reference to `z(:)[1,4]` is valid, since it refers to image `16`, but a reference to `z(:)[2,4]` is invalid, since it refers to image `17`. The programmer is responsible for generating valid cosubscripts. The behaviour of a program that generates an invalid cosubscript is not defined by the Standard.

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

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` construct is needed. For example,

```
if (this_image()==1) read(*,*) z
```

A coindexed object is permitted in intrinsic operations, intrinsic assignments, and input/output lists[2]. It is also permitted as an actual argument corresponding to a non-coarray dummy argument in a procedure invocation. 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 (unless it has intent `out`) and the result copied back on return (unless it has intent `in` or the `value` attribute). The rules for argument association have been carefully constructed so that such copying is always allowed.

Pointers are not allowed to have targets on remote images, because this would break the requirement for remote access to be obvious. Therefore, the target of a pointer is not permitted to be a coindexed object:

```
p => a(n)[p]   ! Not allowed (compile-time constraint)
```

[2] Apart from polymorphic subobjects, see Section 11.

A coindexed object is not permitted as the selector in an `associate` or `select type` statement because that would disguise a reference to a remote image (the associated name is without square brackets). However, a coarray is permitted as the selector, in which case the associated entity is also a coarray and its cobounds are those of the selector.

To avoid a requirement for a remote image to perform finalization, a coindexed object is not finalized if it occurs as the left-hand side of an intrinsic assignment or as an actual argument corresponding to an `intent(out)` dummy argument.

5 Coarrays in procedures

A dummy argument of a procedure is permitted to be a coarray. It may be of explicit shape, assumed size, assumed shape, or allocatable, see Figure 1.

Figure 1: Coarray dummy arguments.

```
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(:)[:,:] ! Allocatable
```

Figure 2: An interface to a procedure with coarray dummy arguments.

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

When the procedure is called, the corresponding actual argument must be a coarray. The association is with the coarray itself and not with a copy. Making a copy would require synchronization on entry and return to ensure that remote references within the procedure are not to a copy that does not exist yet or that no longer exists. 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. An example is shown in Figure 2.

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

- if the dummy argument is a coarray, the actual argument must be a coarray (see Section 3 for the rules on whether a subobject is a coarray);

- if the dummy argument is an array coarray that is not of assumed shape or is of assumed shape and has the `contiguous` attribute (new in Fortran 2008), the actual argument must be "simply contiguous" (a new term for an array that satisfies a set of conditions that have been chosen to allow the compiler to verify at compile time that the array will always be contiguous); and

- if a dummy argument is a coarray or has a component that is an allocatable coarray, it must not have the `value` attribute.

If a dummy argument is an allocatable coarray, the corresponding actual argument must be an allocatable coarray of the same rank and corank. Furthermore, it must be associated, perhaps through many levels of procedure call, with the same non-dummy-argument coarray on every image. This allows the coarray to be allocated or deallocated in the procedure.

If a dummy argument is an allocatable coarray or has a component that is an allocatable coarray, it must not have `intent(out)`. This avoids the possibility of an implicit synchronization associated with deallocation.

Automatic coarrays are not permitted. For example, the following code fragment is not permitted

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

Were automatic coarrays 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 their array sizes are the same in all images. Furthermore, it would mean that the procedure would need to be called on all images concurrently (see penultimate paragraph of this section). Rather than using automatic coarrays, the programmer should use allocatable coarrays (Section 9).

A function result is not permitted to be a coarray or to be of a type that has a coarray component at any level of component selection. A coarray function result is like an automatic coarray and is disallowed for the same reasons.

The rules for resolving generic procedure references have not been extended to allow overloading of array and coarray versions since the syntactic form of the actual argument would not distinguish between the two cases.

A pure or elemental procedure is not permitted to define a coindexed object or contain any image control statements (Section 12.8), since these involve side effects (defining a coindexed object is comparable with defining a variable from the host or a module). However, it may reference the value of a coindexed object.

An elemental procedure is not permitted to have a coarray dummy argument.

Unless it is allocatable or a dummy argument, an object that is a coarray or has a coarray component is required to have the `save` attribute. Note that in Fortran 2008, variables declared

in the specification part of a module or submodule, as well as main-program variables, automatically have the `save` attribute. If a coarray 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 coarray is not required to have the `save` attribute because a recursive procedure may need separate allocatable coarrays at more than one level of recursion.

A procedure with a non-allocatable coarray dummy argument will usually be called simultaneously on all images with the same actual coarray, but this is not a requirement. For example, the images may be grouped into two teams and the images of one team may be calling the procedure with one coarray while the images of the other team are calling the procedure with another coarray or are executing different code.

Each image independently associates its non-allocatable coarray dummy argument with an actual coarray, perhaps through many levels of procedure call, and defines the corank and cobounds afresh. It uses these to interpret each reference to a coindexed object, taking no account of whether a remote image is executing the same procedure with the corresponding coarray.

6 Volatile and asynchronous attributes

The volatility of a dummy coarray is required to agree with the volatility of the corresponding actual argument. Without this restriction, the value of a non-volatile coarray might be altered via another image by means not specified by the program, that is, behave as volatile.

Similarly, agreement of volatility is required when accessing a coarray by use association, host association, or in a `block` construct (new in Fortran 2008) from the scope containing it. Here, the restriction is simple; since the volatility is the same by default, the volatile attribute must not be respecified for an accessed coarray.

For the same reason, volatility agreement is required for pointer association with any part of a coarray.

There are rules in the language to ensure that if an asynchronous or volatile actual argument corresponds to an asynchronous or volatile dummy argument, copy-in copy-out does not occur. This is because a new value might be given to the actual argument while the procedure is in execution and be overwritten on return from the procedure. For an actual argument that is a coindexed object, the requirement is for copy-in copy-out to be allowed (see Section 3), so the dummy argument is not permitted to have the asynchronous or volatile argument in this case.

7 Interoperability

Coarrays are not interoperable, since C does not have the concept of a data object like a coarray. Interoperability of coarrays with UPC might be considered in the future.

8 Storage association

Coarrays are not permitted in `common` and `equivalence` statements.

9 Allocatable coarrays

A coarray may be allocatable. The `allocate` statement is extended so that the cobounds can be specified, for example,

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

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

```
allocate( a(n) )     ! Not allowed for a coarray (no cobounds)
allocate( a(n)[p] )  ! Not allowed (cobound not *)
```

Also, the value of each bound, cobound, 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. Together, these restrictions ensure that the corrays exist on every image and are consistent.

There is implicit synchronization of all images in association with each `allocate` statement that involves one or more coarrays. Images do not commence executing subsequent statements until all images finish executing the same `allocate` statement (on the same line of the source code). 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 have 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 coarray and records how the corresponding coarrays on other images are to be addressed. The compiler is not required to enforce the rule that the bounds and cobounds are the same on all images, although it may do so in debug mode. Nor is the compiler responsible for detecting or resolving deadlock problems.

If an allocatable coarray is declared without the `save` attribute in a procedure or in a block construct (new in Fortran 2008), and the coarray is still allocated when the procedure or block construct completes execution, implicit deallocation of the coarray and therefore synchronization of all images occurs.

The allocation of a polymorphic coarray is not permitted to create a coarray that is of type `c_ptr`, `c_funptr`, or of a type with a coarray ultimate component.

Fortran 2003 allows the shapes or length parameters to disagree on the two sides of an intrinsic array assignment to an allocatable array; the system performs the appropriate reallocation. Such disagreement is not permitted for an allocatable coarray, since it would imply synchronization.

For the same reason, intrinsic assignment is not permitted to a polymorphic coarray.

10 Coarrays with allocatable or pointer components

A coarray is permitted to be of a derived type with allocatable or pointer components (but components of type `c_ptr` or `c_funptr` are not permitted).

10.1 Non-coarray components

To use coarray syntax for data structures with different sizes, length parameter values, or types on different images, we may declare a coarray of a derived type with a non-coarray component that is an allocatable or pointer array. On each image, the component is allocated locally or is pointer assigned to a local target, so that it has the desired properties for that image (or is 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 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 coarray syntax gives the programmer power and flexibility for writing parallel code. Numrich used this technique to build an object-based parallel library for CoArray Fortran (Numrich 2006).

If coarray `z` contains a data pointer component `ptr`, `z[q]%ptr` in a context that refers to its target 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%ptr => r ! Local association
```

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

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

but may not be associated with a target component on another image,

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

If an association with a target component on another image would otherwise be implied, the pointer component becomes undefined. For example, this happens with the derived-type intrinsic assignments of Figure 3 are executed on an image with an index other than `q`. It can also happen in a procedure invocation if `z[q]` is an actual argument or `z[q]%ptr` is associated with a pointer

Figure 3: Code executed on an image with an index other than q.

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

dummy argument.

Similarly, for a coarray of a derived type that has a pointer or allocatable component, allocating one of those components on another image is not allowed:

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

To avoid the possibility of a remote allocation, the shapes and length parameters are required to agree in an intrinsic assignment to a coindexed object that is an allocatable array and intrinsic assignment to a coindexed object with an allocatable ultimate component is not allowed. Intrinsic assignment is not permitted to a polymorphic coindexed object. Furthermore, if an actual argument is a coindexed object with an allocatable ultimate component and the corresponding dummy argument is neither allocatable nor a pointer, the dummy argument must have intent `in` or the `value` attribute.

10.2 Coarray components

A derived type is permitted to have a coarray component provided the component is allocatable. If an object has a coarray component at any level of component selection, each ancestor of the coarray component must be a non-allocatable, non-pointer, non-coarray scalar. Were we to allow a coarray of a type with coarray 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 equivalent to `z[q]%x`.

If an object with an allocatable coarray component is declared without the `save` attribute in a procedure and the coarray is still allocated on return, there is an implicit deallocation and associated synchronization. Similarly, if such an object is declared within a `block` construct and the coarray is still allocated when the block completes execution, there is an implicit deallocation and associated synchronization.

To avoid the possibility of implicit reallocation in an intrinsic assignment for a scalar of a derived type with an allocatable coarray component, no disagreement of allocation status or shape is permitted for the coarray component. This, of course, can be checked by the system on the executing image.

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

10.3 Procedure pointer components

A coarray is permitted to be of a type that has a procedure pointer component or a type bound procedure. A procedure reference through a procedure pointer component of a coindexed object, for example,

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

is not permitted since the remote procedure target might be meaningless on the executing image. However, a reference through a type-bound procedure (see, for example, Section 16.6 of Metcalf, Reid, and Cohen 2004) is allowed provided the type is not polymorphic; this ensures that the type and hence the procedure is the same on all images.

11 References to polymorphic subobjects

So that the implementation does not need to access the dynamic type of an object on another image, no references are permitted to a polymorphic subobject of a coindexed object or to a coindexed object that has a polymorphic allocatable subcomponent.

12 Synchronization

Each image executes on its own without regard to the execution of other images except when it encounters special statements called **image control statements**. The programmer inserts image control statements to ensure that, whenever one image alters the value of a coarray variable or a variable with the `target` attribute, no other image still needs the old value, and that whenever an image accesses the value of a variable, it receives the wanted value – either the old value, not updated by another image, or the new value that has been updated by another image. The following sections describe each image control statement in detail. A complete list is found in Section 12.8.

12.1 `sync all` statement

The `sync all` statement provides a barrier where all images synchronize before executing further statements. All statements executed before the barrier on image P execute before any statement executes after the barrier on image Q. If the value of a variable is changed by image P before the barrier, the new value is available to all other images after the barrier. If an image references the value of a variable before the barrier, it obtains the value before crossing the barrier.

Figure 4 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 `z` by another image. The second `sync all` ensures that another image does not access `z` before the new value has been set by image 1.

Figure 4: Read on image 1 and broadcast to the others.

```
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
   ...
```

Although usually the synchronization will be initiated by 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.

The behaviour at program initiation is as if there were a `sync all` as the first executable statement of the main program. The code can rely on initializations of coarray variables on other images.

12.2 Execution segments

On each image, the sequence of statements executed before the first execution of an image control statement or between the execution of two image control statements 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 4, 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 P_{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 4 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 on what is permitted in a segment that is unordered with respect to another segment. These provide the compiler with scope for optimization. A coarray may be defined and referenced during the execution of unordered segments by calls to atomic subroutines (Section 15.3). Apart from this,

- if a variable is defined in a segment on an image, 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 coarray or the pointer association of a pointer subobject of a coarray is changed in a segment on an image, 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-coarray 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).

It follows that for code in a segment, the compiler is free to use almost all its normal optimization techniques as if only one image were present. In particular, code motion optimizations may be applied, provided calls of atomic subroutines are not involved. For an example of an optimization that is not available, consider the code

```
integer (kind=short) x(8)[*]
   :
!  Computation that references and alters x(1:7)
```

Because another image might define `x(8)` in a segment that is unordered with respect to this one, the compiler must not effectively make this replacement:

```
integer (kind=short) x(8)[*], temp(8)
   :
temp(1:8) = x(1:8) ! Faster than temp(1:7) = x(1:7)
!  Computation that references and alters temp(1:7)
x(1:8) = temp(1:8)
```

12.3 The `sync images` statement

For greater flexibility, the `sync images` statement

```
sync images(image-set)
```

where *image-set* is either an integer array of rank one holding distinct image indices or an asterisk indicating all images, performs a synchronization of the image that executes it with each of the other images in its image set. 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. Figure 5 shows an example that imposes the fixed order 1, 2, ... on images.

Figure 5: Using `sync images` to impose an order on images.

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

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 example in Figure 6, 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 for each other.

Figure 6: Using `sync images` to make other images to wait for image 1.

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

12.4 The `lock` and `unlock` statements

Locks provide a mechanism for controlling access to data that is referenced or defined by more than one image.

A lock is a scalar variable of the derived type `lock_type` that is defined in the intrinsic module `iso_fortran_env`. The type has private components that are not pointers and are not allocatable. It does not have the `bind(C)` attribute or type parameters, and is not a sequence type.

All components have default initialization. A lock must be a coarray or a subobject of a coarray. It has one of two states: **locked** and **unlocked**. The unlocked state is represented by a single value and this is the initial value. All other values are locked. The only way to change the value of a lock is by executing the `lock` or `unlock` statement. For example, if a lock is a dummy argument or a subobject of a dummy argument, the dummy argument must not have intent `out`. If a lock variable is locked, it can be unlocked only by the image that locked it.

Figure 7: Using `lock` and `unlock` to manage stacks.

```
module stack_manager
   use, intrinsic :: iso_fortran_env, only: lock_type
   type task
      :
   end type
   type(lock_type), private :: stack_lock[*]
   type(task), private :: stack(100)[*]
   integer, private :: stack_size[*]
contains
   subroutine get_task(job)
   ! Get a task from my stack
      type(task),intent(out) :: job
      lock(stack_lock)
         job=stack(stack_size)
         stack_size=stack_size-1
      unlock(stack_lock)
   end subroutine get_task
   subroutine put_task(job,image)
   ! Put a task on the stack of image
      type(task),intent(in) :: job
      integer,intent(in) :: image
      lock(stack_lock[image])
         stack_size[image]=stack_size[image]+1
         stack(stack_size[image])[image] = job
      unlock(stack_lock[image])
   end subroutine put_task
end module stack_manager
```

Figure 7 illustrates the use of `lock` and `unlock` statements to manage stacks. Each image has its own stack; any image can add a task to any stack. If a `lock` statement is executed for a lock variable that is locked by another image, the image waits for the lock to be unlocked by that image. The effect in this example is that `get_task` has to wait if another image is adding a task to the stack and `put_task` has to wait if `get_task` is getting a task from the stack or another image is executing `put_task` for the same stack.

There is a form of the `lock` statement that avoids a wait when the lock variable is locked:

```
logical :: success
lock(stack_lock,acquired_lock=success)
```

If the variable is unlocked, it is locked and the value of `success` is set to true; otherwise, `success` is set to false and there is no wait.

An error condition occurs for a `lock` statement if the lock variable is already locked by the executing image and for an `unlock` statement if the lock variable is not already locked by the executing image.

Any particular lock variable is successively locked and unlocked by a sequence of `lock` and `unlock` statements, each of which separates two segments on the executing image. If execution of such an `unlock` statement M_u on image M is immediately followed in this sequence by execution of a `lock` statement T_l on image T, the segment that precedes the execution of M_u on image M precedes the segment that follows the execution of T_l on image T.

For a sourced allocation of a coarray (using `source=` to take its value from another variable or expression), the source is not permitted to have type `lock_type` or have a subcomponent of this type because this might create a new lock that is not unlocked initially.

12.5 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 must be a single segment. Branching into or out of a critical section is not permitted.

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.

12.6 The `sync memory` statement

The execution of a `sync memory` statement defines a boundary on an image between 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 8, which is executed on images `p` and `q` and calls atomic subroutines (see Section 15.3). The do loop is known as a spin-wait loop. Once image

Figure 8: Spin-wait loop

```
use, intrinsic :: iso_fortran_env
logical(atomic_logical_kind) :: locked[*] = .true.
logical :: val
integer :: iam, p, q
   :
iam = this_image()
if (iam == p) then
   sync memory
   call atomic_define(locked[q],.false.)
   ! Has the effect of locked[q]=.false.
else if (iam == q) then
   val = .true.
! Spin until val is false
   do while (val)
      call atomic_ref(val,locked)
      ! Has the effect of val=locked
   end do
   sync memory
end if
```

`q` starts executing it, it will continue until it finds the value `.false.` for `val`. The `atomic_ref` call ensures that the value is refreshed on each loop execution. The effect is that the segment on image `p` ahead of the first `sync memory` statement precedes the segment on image `q` that follows the second `sync memory` statement. The normative text of the Standard does not specify how resources should be distributed between images, but a note expects that the sharing should be equitable. It is therefore just possible that a conforming implementation might give all its resources to the spin loop while doing nothing on image `p`, causing the program to hang.

Note that the segment in which `locked[q]` is altered is unordered with respect to the segment in which it is referenced. This is permissible by the rules in the penultimate paragraph of Section 12.2.

Given the atomic subroutines and the `sync memory` statement, customized synchronizations can be programmed in Fortran as procedures, but it may be difficult for the programmer to ensure that they will work correctly on all implementations.

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

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

If any of these statements, including `allocate` and `deallocate`, encounter an image that has executed a `stop` or `end program` statement and have a `stat=` specifier, the `stat=` variable is given the value of the constant `stat_stopped_image` in the `iso_fortran_env` intrinsic module, and the effect of executing the statement is otherwise the same as that of executing the `sync memory` statement. Without a `stat=` specifier, the execution of such a statement initiates error termination (Section 13).

12.8 The image control statements

The full list of image control statements is

- `sync all` statement;
- `sync images` statement;
- `lock` or `unlock` statement;
- `sync memory` statement;
- `allocate` or `deallocate` statement involving a coarray;
- `critical` or `end critical` statement;
- `end` or `return` statement that involves an implicit deallocation of a coarray;
- a statement that completes the execution of a `block` (new in Fortran 2008) and results in an implicit deallocation of a coarray;
- `stop` or `end program` statement.

All of the image control statements except `critical`, `end critical`, `lock`, and `unlock` include the effect of executing a `sync memory` statement.

13 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. This is called **normal termination**. 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. This is called **error termination**.

Normal termination occurs in three steps: initiation, synchronization, and completion. An image initiates normal termination if it executes a `stop` or `end program` statement. All images

synchronize execution at the second step so that no image starts the completion step until all images have finished the initiation step. The synchronization step allows its data to remain accessible to the other images until they all reach the synchronization step. Normal termination may also be initiated during execution of a procedure defined by a C companion processor.

An image initiates error termination if it executes a statement that would cause the termination of a single-image program but is not a `stop` or `end program` statement. This causes all other images that have not already initiated error 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

```
error 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 error termination to initiate error termination. It thus causes the whole calculation to stop as soon as is practicable. Just like `stop`, it must not be executed during I/O processing.

13.1 Example of normal and error termination

The code fragment in Figure 9 illustrates the use of `stop` and `error stop` in a climate model that uses two teams, one for the ocean and one for the atmosphere.

If something goes badly wrong in the atmosphere calculation, the whole model is invalid and a restart is impossible, so all images stop as soon as possible without trying to preserve any data.

If something goes slightly wrong with the atmosphere calculation, the images in the atmosphere team write their data to files and stop, but their data remain available to the ocean images which complete execution of the ocean subroutine. On return from the computation routines, if something went slightly wrong with the atmosphere calculation, the ocean images write data to files and stop, ready for a restart in a later run.

14 Input/output

Just as each image has its own variables, so it has its own input/output units. Whenever an input/output statement uses an integer expression to index a unit, it refers to the unit on the executing image.

The default unit for input (`*` in a `read` statement or `input_unit` in the intrinsic module `iso_fortran_env`) is preconnected on image one only.

The default unit for output (`*` in a `write` statement or `output_unit` in the intrinsic module `iso_fortran_env`) and the unit that is identified by `error_unit` in the intrinsic module

Figure 9: `stop` and `error stop` in a climate model.

```
   use,intrinsic :: iso_fortran_env, only: stat_stopped_image
   integer, allocatable :: ocean_team(:), atmosphere_team(:)
   integer :: i, sync_stat
   :
! Form two teams
   ocean_team =[(i,i=1,num_images()/2)]
   atmosphere_team = [(i,i=1+num_images()/2,num_images())]
   :
! Perform independent calculations
   if (this_image() > num_images()/2) then
      call atmosphere (atmosphere_team)
   else
      call ocean (ocean_team)
   end if
! Wait for both teams to finish
   sync all (stat=sync_stat)
   if (sync_stat == stat_stopped_image) then
       : ! Preserve data on file
      stop
   end if
   call exchange_data ! Exchange data between teams
   :
contains
   subroutine atmosphere (team)
      integer :: team(:)
      : ! Perform atmosphere calculation.
      if (...) then ! Something has gone slightly wrong
         : ! Preserve data on file
        stop
      end if
   :
      if (...) error stop  ! Something has gone very badly wrong
      :
      sync images (team, stat=sync_stat))
      if (sync_stat == stat_stopped_image) then
         : ! Remaining atmosphere images preserve data in a file
        stop
      end if
   end subroutine atmosphere
```

`iso_fortran_env` are preconnected on each image. The files to which these are connected are regarded as separate, but it is expected that the processor will merge their records into a single stream or a stream for all `output_unit` files and a stream for all `error_unit` files. If the order of writes from images is important, synchronization and the `flush` statement are required, since the image is permitted to hold the data in a buffer and delay the transfers until it executes a `flush` statement for the file or the file is closed.

Any other preconnected unit is connected on the executing image only and the file is completely separate from any preconnected file on another image.

The `open` statement connects a file to a unit on the executing image only. It is implementation specific as to whether a file connected by a given name differs according to the image that opens it.

The effect of connecting a named file on more than one image is outside the scope of the Fortran 2008 Standard but is expected to be defined in the forthcoming Technical Report.

15 Intrinsic procedures

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

15.1 Inquiry functions

`image_index(coarray,sub)` returns a default integer scalar. If `sub` holds a valid sequence of cosubscripts for `coarray`, the result is the corresponding image index. Otherwise, the result is zero.

`coarray` is a coarray of any type.
`sub` is a rank-one integer array of size equal to the corank of `coarray`.

`lcobound(coarray [,dim] [,kind])` returns the lower cobounds of a coarray in just the same way as `lbound` returns the lower bounds of an array.

`ucobound(coarray [,dim] [,kind])` returns the upper cobounds of a coarray in just the same way as `ubound` returns the upper bounds of an array.

15.2 Transformational functions

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

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

`this_image(coarray[,dim])` returns the set of cosubscripts of `coarray` that denotes data on the invoking image.

`coarray` is a coarray of any type.

`dim` is scalar integer whose value is in the range $1 \leq$ `dim` $\leq n$ where n is the corank of `coarray`.

If `dim` is absent, the result is a default integer array of rank one and size equal to the corank of `coarray`; it holds the set of cosubscripts of `coarray` for data on the invoking image. If `dim` is present, the result is a default integer scalar holding cosubscript `dim` of `coarray` for data on the invoking image.

15.3 Atomic subroutines

An **atomic subroutine** is a new class of intrinsic subroutine. It acts on a scalar variable `atom` of type `integer(atomic_int_kind)` or `logical(atomic_logical_kind)`, whose kind value is defined in the intrinsic module `iso_fortran_env`. The variable `atom` must be a coarray or a coindexed object. The effect of executing an atomic subroutine is as if the action on the argument `atom` occurs instantaneously, and thus does not overlap with other atomic actions that might occur asynchronously. To avoid performance loss, the ordering of interleaved actions on different atomic variables in different images is not defined by the Standard.

call `atomic_define(atom,value)` defines `atom` atomically with the value `value`.

`atom` is a scalar coarray or coindexed object of type `integer(atomic_int_kind)` or `logical(atomic_logical_kind)` and `intent(out)`. If its kind is the same as that of `value` or its type is `logical`, it is given the value of `value`. Otherwise, it is given the value `int(value,atomic_int_kind)`.

`value` is a scalar of the same type as `atom` and `intent(in)`.

call `atomic_ref(value,atom)` defines `value` atomically with the value of `atom`.

`value` is a scalar of the same type as `atom` and `intent(out)`. If its kind is the same as that of `atom` or its type is `logical`, it is given the value of `atom`. Otherwise, it is given the value `int(atom,kind(value))`.

`atom` is a scalar coarray or coindexed object of type `integer(atomic_int_kind)` or `logical(atomic_logical_kind)` and `intent(in)`.

16 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 coarrays in the US Fortran Committee PL22.3; and I would like to thank Bill Long, Aleks Donev, Van Snyder, Bob Numrich, and Reinhold Bader for carefully reading drafts of this document and suggesting corrections and improvements.

17 References

Metcalf, Michael, Reid, John, and Cohen, Malcolm (2004). Fortran 95/2003 explained. Oxford University Press.

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*, 31, pp. 588-607.

Numrich, R.W. (2006). 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. and Reid, J. K. (2007). Co-arrays in the next Fortran Standard. Scientific Programming (2006), 14, 1-18.

Numrich, R. W., Reid, J. K., and Kim, K. (1998). Writing a multigrid solver using Co-array Fortran. Proceeding of the fourth International Workshop on Applied Parallel Computing (PARA98), pp. 390-399, Springer Lecture Notes in Computer Science, LNCS 1541.

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`

WG5 (2008). Draft revision of the Fortran Standard. ISO/IEC/JTC1/SC22/WG5-N1723, see

`ftp://ftp.nag.co.uk/sc22wg5/N1701-N1750/N1723.pdf`

WG5 (2010). DIS revision of the Fortran Standard. ISO/IEC/JTC1/SC22/WG5-N1826, see

`ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1826.pdf`