Coarrays in the next Fortran Standard

John Reid, JKR Associates, UK

March 18, 2008

Abstract

The WG5 committee, at its meeting in Delft, May 2005, decided to include coarrays in the next Fortran Standard. A Fortran Forum article in August 2005 explained the feature, but because many of the details have been changed since then and the features have been split into a core set that is intended for Fortran 2008 and further features that will be defined in a Technical Report, it seems appropriate to describe it afresh. Only the core set is described here. This article is not an official document and has not been approved by J3 or WG5.

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, 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 and the index of the current image.
## Contents

1 Introduction 4

2 Referencing images 6

3 Specifying data objects 6

4 Accessing data objects 8

5 Coarrays in procedures 9

6 Interoperability 11

7 Storage association 11

8 Allocatable coarrays 11

9 Restrictions on intrinsic assignment for coarrays 12

10 Array pointers 12

11 Synchronization 14

   11.1 sync all statement .................................................. 14
   11.2 Execution segments ................................................ 15
   11.3 The sync images statement ....................................... 16
   11.4 The sync memory statement ....................................... 17
   11.5 stat= and errmsg= specifiers in synchronization statements .... 18
   11.6 Critical sections ................................................. 18
   11.7 The image control statements ................................... 18

12 Program termination 19

   12.1 Example of normal and error termination ....................... 20

13 Input/output 20

14 Intrinsic procedures 22
14.1 Inquiry functions .................................................. 22
14.2 Image index functions ............................................. 22

15 Acknowledgements .................................................. 22

16 References ............................................................ 23

17 Appendix: Changes from the Fortran Forum article of 2005 ...... 24
1 Introduction

Coarrays 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 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 all its normal optimization techniques as if only one image were present.

At least in early implementations, it is expected that each image will execute the same executable code (.o 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 statement

\[
\text{real, dimension(1000)[*]} :: x, y
\]

declares two objects \(x\) and \(y\), each as a coarray. A coarray always has the same shape on each image. In this example, each image has two real coarrays of size 1000. If an image executes the statement:

\[
x(:,:, q) = y(:,:, q)
\]

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. 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 coarray on an image and the corresponding coarrays 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 coarray may be stored from 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 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 and automatic type conversions in assignments, information about structure layout, and even object-oriented features.

The coarray feature adopted by WG5 was formerly known as CoArray Fortran, an informal extension to Fortran 95 by Numrich and Reid (1998). CoArray 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 CoArray 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 caf and performs source-to-source transformation of coarray code to Fortran 90 augmented with communication operations. One instantiation uses the Aggregate Remote Memory Copy Interface (ARMCI) library for one-sided communication (Nieplocha and Carpenter 1999) and another uses GASNet (Bonachea 2002). Experience with the use of caf is related by Coarfa, Dotsenko, and Mellor-Crummey, Cantonnet, El-Ghazawi, Mohanti, Yao, and Chavarría-Miranda (2005). They found that on several platforms caf gave performance comparable with MPI on the NAS MG, CG, SP, and BT parallel benchmarks (Bailey, Harris, Saphir, van der Wijngaart, Woo, and Yarrow 1995).

Reid (2005) proposed that coarrays 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 coarrays in May 2005, but made some changes and further changes have been made since then by J3, the Primary Development Body for Fortran. 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 a Technical Report. The rest of this article contains a complete description of the core set. For more extensive discussion and examples, see Numrich and Reid (1998).
2 Referencing images

Data on other images are normally referenced by cosubscripts enclosed in square brackets. Each 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 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 function this_image() with no arguments. The set of subscript indices that correspond to the current image for a coarray z are available as this_image(z). The image index that corresponds to a set of cosubscript indices sub for a coarray z is available as image_index(z, sub). For example, on image 5, for the array declared

\[
\text{real} :: z(10,20)[10,0:9,0:*]
\]

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 codimensions in square brackets immediately following dimensions in parentheses or in place of them, for example:

\[
\begin{align*}
\text{real, dimension}(20)[20,\ast] & : : a \quad \text{An array coarray} \\
\text{real} & : : c[\ast], d[\ast] \quad \text{Scalar coarrays} \\
\text{character} & : : b(20)[20,0:\ast] \\
\text{integer} & : : ib(10)[\ast] \\
\text{type(interval)} & : : s[20,\ast]
\end{align*}
\]

Unless the coarray is allocatable (Section 8), 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 coarray on another image may be addressed by using subscripts in square brackets following any subscripts in parentheses, for example:

\[
\begin{align*}
a(5)[3,7] & = \text{ib}(5)[3] \\
d[3] & = c \\
a(:,:,2,3] & = c[1]
\end{align*}
\]

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 parent has nonzero rank. For example, a[2,3] is not permitted as a shorthand for a(:,:,2,3].
A subobject of a coarray is a coarray if it has no cosubscripts, no vector-valued subscripts, no allocatable component selection unless the component is a coarray, and no pointer component selection. These restrictions make it 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 **rank**, **bounds**, **extents**, **size**, and **shape** of a whole coarray are given by the data in parentheses in its declaration or allocation. Its **corank**, **cobounds**, and **coextents** are given by the data in square brackets in its declaration or allocation. Any subobject of it that is a coarray has the same corank, cobounds, and coextents. 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, which depend on the number of images. For example, when the number of images is 128, the coarray declared thus

```plaintext
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. Note that the set of cosubscripts [1,1,1] is valid, but the set [9,1,1] is not since it would correspond to image 129.

A coarray 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), cosubscripts are not permitted in **data** statements. For example:

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

A coarray may be allocatable as discussed in Section 8.

A coarray is not permitted to be a pointer, but a coarray may be of a derived type with pointer or allocatable components as discussed in Section 10.

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 logically equivalent to $z[q]%x$.

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

Because finalization may involve synchronization, the steps of finalization are required to be performed in the same order on all images.
4 Accessing data objects

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

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

the value of the largest element of the coarray `z` on the executing image is placed in the 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

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

then a reference to `z(:,1,4)` is valid, since it has cosubscript order value 16, but a reference to `z(:,2,4)` is invalid, since it has cosubscript order value 17. The programmer is responsible for generating valid cosubscripts. The behaviour of a program that generates an invalid cosubscript 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

```fortran
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

```fortran
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 coindexed 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 coindexed 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. Similarly, if the actual argument is a coindexed object and the dummy argument is allocatable or a pointer, the dummy argument must have intent in.

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.

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:

```fortran
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
end subroutine subr
```

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 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

```fortran
interface
   subroutine sub(x,y)
      real :: x(:)[*], y(:)[*]
   end subroutine sub
end interface
```

```fortran
call sub(a(:),b(1,:))
```

The restrictions that avoid copy-in and/or copy-out are:
1. the actual argument must be a coarray (see Section 3 for the rules on whether a subobject is a coarray); and

2. if the dummy argument is a coarray that has the contiguous attribute (to be added in Fortran 2008) or is not of assumed shape, 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).

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.

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

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

Were automatic-array 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 the sizes are the same in all images.

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 are based on the local properties and are therefore unchanged. The rules cannot be extended to allow overloading of array and coarray 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 coindexed object or contain any image control statements (Section 11.7), since these involve side effects. However, it may reference the value of a coindexed object.

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. 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 arrays at more than one level of recursion.

Note that in Fortran 2008, all variables declared in a module, including coarrays, automatically have the save attribute.

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 the remote image is executing the same procedure with the same coarray.

6 Interoperability

Coarrays are not interoperable, since C does not have the concept of a coarray.

7 Storage association

Coarrays are not permitted in common and equivalence statements.

8 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 (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.

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. 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 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.

For an allocatable coarray declared without the save attribute in a procedure, if the coarray 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.

9 Restrictions on intrinsic assignment for coarrays

Fortran 2003 allows the shapes or length parameters 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 coarray, since it would involve synchronization. Similarly, 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.

Restrictions are also needed for coindexed objects to avoid the possibility of requiring allocation to occur on another image. In an intrinsic assignment to a coindexed object that is an allocatable array, the shapes and length parameters are required to agree. Intrinsic assignment to a coindexed object of a type with an allocatable component at any level of component selection is not permitted.

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

10 Array pointers

A coarray 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 coarray is not permitted to be of either of these types.

However, a coarray 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 coarray syntax for data structures with different sizes on different images, we may declare a coarray 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 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 2005b).

If coarray \( z \) contains a pointer component \( \text{ptr} \), \( z[q]\%\text{ptr} \) is a reference to the target of component \( \text{ptr} \) of \( z \) on image \( q \). This target must reside on image \( q \) and must have been established by an \texttt{allocate} statement executed on image \( q \) or a pointer assignment executed on image \( q \), for example,

\[
z\%p \Rightarrow r \quad \text{Local association}
\]

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

\[
r \Rightarrow z\%p \quad \text{Local association}
\]

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

\[
r \Rightarrow z[q]\%p \quad \text{Not allowed (compile-time constraint)}
\]

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

\[
z[q] = z \quad \text{The pointer component of} \ z[q] \ \text{becomes undefined}
\]

\[
z = z[q] \quad \text{The pointer component of} \ z \ \text{becomes undefined}
\]

when executed on an image with an index other than \( q \).

Similarly, for a coarray of a derived type that has a pointer or allocatable component, it is illegal to allocate one of those components on another image:

\[
\text{type(something), allocatable :: } t[:]
\]

\[
\ldots
\]

\[
\text{allocate}(t[*]) \quad \text{Allowed}
\]

\[
\text{allocate}(t\%\text{ptr}(n)) \quad \text{Allowed}
\]

\[
\text{allocate}(t[q]\%\text{ptr}(n)) \quad \text{Not allowed (compile-time constraint)}
\]

A coarray 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

\[
\text{call } a[p]\%\text{proc}(x)
\]

finds the target procedure on image \( p \) and calls this procedure on the executing image.
11 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 variable, no other image might still need the old value and that, whenever an image accesses the contents of a variable, 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.

11.1 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 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.

```fortran
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.

The behaviour at program initiation is as if there were a sync all as the first executable statement of the main program. The code may rely on initializations of coarray variables on other images.
11.2 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 11.7.

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 $P_i$, the statement execution order determines the segment order, $P_i, i=1, 2, ...$. Between images, the execution of corresponding image control statements on images $P_i$ and $Q_j$ 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 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 coarray variable that is of type default integer, default logical, or default real, 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 coarray variable is defined. Otherwise,

- if a 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 coarray or the pointer association of a pointer subobject of a coarray 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-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).

The restriction on a volatile variable that is permitted to be used in this way to one of three 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.

### 11.3 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:

```plaintext
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.

**Figure 2:** Using `sync images` to make other images to wait for image 1.

```plaintext
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
```

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 2, 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.

11.4 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 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 the 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 11.2 since its type is default logical. The reason for this is discussed in the final paragraph of Section 11.2.
Given the fundamental \texttt{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 \texttt{sync memory} statement to express customized synchronization operations in Fortran.

11.5 \texttt{stat=} and \texttt{errmsg=} specifiers in synchronization statements

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

If any of these statements, including \texttt{allocate} and \texttt{deallocate}, encounter a stopped image, the \texttt{stat=} variable is given the value of the constant \texttt{stat.stopped.image} in the \texttt{iso_fortran_env} intrinsic module, and the effect of executing the statement is otherwise the same as that of executing the \texttt{sync memory} statement.

11.6 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 \textbf{critical section}. There is a new construct to delimit a critical section:

\begin{verbatim}
critical
   :   ! code that is executed on one image at a time
end critical
\end{verbatim}

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 $T$.

11.7 The image control statements

The full list of image control statements is

- \texttt{sync all} statement;
- \texttt{sync images} statement;
- \texttt{sync memory} statement;
- \texttt{allocate} or \texttt{deallocate} statement involving a coarray;
- \texttt{critical} or \texttt{end critical} statement;
- \texttt{end}, \texttt{end block} (to be added in Fortran 2008), or \texttt{return} statement that involves an implicit deallocation of a coarray;
• stop or end program statement.

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

12 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 of execution of an image occurs in three steps: initiation, synchronization, and completion. All images synchronize execution at the second step so that no image starts the completion step until all images have finished the initiation step. Termination of execution of an image is either ‘normal termination’ or ‘error termination’. An image that initiates normal termination also completes normal termination. An image that initiates error termination also completes error termination. The synchronization step is executed by all images. Termination of execution of the program occurs when all images have terminated execution.

An image initiates normal termination if it executes a stop or end program statement. 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 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. Just like stop, it must not be executed during I/O processing.

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 images, or sync memory 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 condition occurs by means outside Fortran.

12.1 Example of normal and error termination

The code fragment in Figure 4 illustrates the use of stop and all 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.

13 Input/output

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 identified by error_unit in the intrinsic module iso_fortran_env are preconnected on all images. The processor ensures that once an image commences transferring the data of a record, no other image transfers data until the whole record has been transferred. Thus, each record arises from a single image. 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 is executed or the file is closed.

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

The open statement connects a file to a unit on the executing image only. Whether a named file on one image is the same as a file with the same name on another image is processor dependent. A named file must not be connected on more than one image.
Figure 4: stop and all stop in a climate model.

```fortran
use, intrinsic :: iso_fortran_env
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 == sync_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 (...) all stop ! Something has gone very badly wrong
  sync images (team, stat=sync_stat)
  if (sync_stat == sync_stopped_image) then
    ! Remaining atmosphere images preserve data in a file
    stop
  end if
end subroutine atmosphere
```
14 Intrinsic procedures

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

14.1 Inquiry functions

\texttt{num\_images()} returns the number of images as a default integer scalar.

\texttt{co\_lboun~d(coarray[,dim,kind])} returns the lower cobounds of a coarray in just the same way as \texttt{lbound} returns the lower bounds of an array.

\texttt{co\_uboun~d(coarray[,dim,kind])} returns the upper cobounds of a coarray in just the same way as \texttt{ubound} returns the upper bounds of an array.

14.2 Image index functions

\texttt{image\_index(coarray,sub)} returns the index of the image corresponding to the set of cosubscripts for \texttt{coarray} as a default integer scalar.

\texttt{co} is a coarray of any type.

\texttt{sub} is a rank-one integer array of size equal to the corank of \texttt{coarray}.

\texttt{this\_image()} returns the index of the invoking image as a default integer scalar.

\texttt{this\_image(coarray[,dim])} returns the set of cosubscripts of \texttt{coarray} that denotes data on the invoking image.

\texttt{co} is a coarray of any type.

\texttt{dim} is scalar integer whose value is in the range \(1 \leq \text{dim} \leq n\) where \(n\) is the corank of \texttt{coarray}.

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

15 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 J3; and I would like to thank Bill Long, Aleks Donev, Van Snyder, and Bob Numrich for carefully reading drafts of this document and suggesting corrections and improvements.
16 References


ftp://ftp.numerical.rl.ac.uk/pub/reports/nrRAL98060.pdf


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

17 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 hyphen has been removed from the terms ‘co-array’, ‘co-rank’, ‘co-bound’, etc., for consistency with the use of the terms ‘cosine’, ‘cotangent’, etc.

2. The term ‘coarray’ now refers to the object on the executing image, rather than the collection of objects on all the images. Furthermore, it includes those subjects that are permitted to be associated as actual arguments with dummy coarrays. The Standard was not consistent and this interpretation simplifies the text.

3. The following features have been moved to a future Technical Report and are not part of the core set: the collective subroutines, the type `image_team` and the features that use it, `notify` and `query`, file connection on more than one image, except for the files preconnected for default and error output (`output_unit` and or `error_unit` in the intrinsic module `iso_fortran_env`). The features that have been moved are not described in this article.

4. 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`.

5. The default input file is connected on image 1 only. The default output and error files are preconnected on all images.

6. If a dummy argument that is allocatable or has an allocatable ultimate component is associated with a coindexed actual argument, the dummy argument must have the intent `in` or `value` attribute. Otherwise, an explicit interface is no longer needed just because a coindexed object is an actual argument.

7. A coindexed object is allowed to be of a type with pointer components. A pointer component becomes undefined if its value is copied from an 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 coindexed 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 with the rule for arrays.

9. Significant changes on termination have been made. Section 12 has been added to this report to explain how it now works.

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

11. co_lboud and co_ubound have been added.

12. A remote procedure reference is regarded as local.

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

14. Rather than being undefined, the final upper cobound of a coarray is now defined as the largest value that the final cobound can take in a valid set of cobounds for the coarray.

15. The save attribute is no longer required for coarrays declared in a module because it is now implicit for all module variables.

16. No coarray is interoperable. Previously, coarrays were interoperable, but a procedure with a coarray dummy argument was not.