Coarrays in the next Fortran Standard

John Reid, JKR Associates, UK

June 15, 2009

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 PL22.3 (formerly 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

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 almost all its normal optimization techniques as if only one image were present. In particular, code motion optimizations may be applied within a segment, provided calls of atomic subroutines (Section 15.3) are not involved. An example of an optimization that is not available is shown in at the end of Subsection 12.2.

At least in early implementations, it is expected that 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 statement

```
real, dimension(1000), codimension[*] :: 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(:) = 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 (with some restrictions) object-oriented features.

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). A portable compiling system for a subset of the extension has been implemented by Dotsenko, Coarfa, and Mellor-Crummey (2004). It is called `cafc` and performs source-to-source transformation of co-array code to Fortran 90 augmented with 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 `cafc` is related by Coarfa, Dotsenko, and Mellor-Crummey, Cantonnet, El-Ghazawi, Mohanti, Yao, and Chavarría-Miranda (2005). They found that on several platforms `cafc` 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).

From 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). It involves support for running images across networks through a central program, the 'G95 Coarray Console'.

---

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

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 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 as

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

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

3   Specifying data objects

Each image has its own set of data objects, all of which may be accessed in the normal Fortran way. Some objects are declared with **codimensions** in square brackets immediately following dimensions in parentheses or in place of them, 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 coarray on another image may be addressed by using subscripts in square brackets following any subscripts in parentheses, for example:

```
a(5)[3,7] = ib(5)[3]
d[3] = c
a(:)[2,3] = c[1]
```

We call any object whose designator includes square brackets a **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

```
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:

```
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 9.

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 11.

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. This ensures that the restrictions that apply to objects of the extended type also apply to objects of the original type.

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.

4 Accessing data objects

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

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

the value of the largest element of the 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

```
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

```
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 in Figure 1 employs the first image to read data and broadcast it to the other images.

Figure 1: 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
```

A coindexed object is permitted in intrinsic operations, intrinsic assignments, and input/output lists[2]. 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. To allow this, the dummy argument is not permitted to be `asynchronous` or `volatile`.

If the dummy argument is neither allocatable nor a pointer, and the actual argument is a coindexed object with an allocatable ultimate component (coarrays with allocatable or pointer components are discussed in Section 11), the dummy argument must have the intent `in` or the `value` attribute; this ensures that allocatables are not allocated remotely.

If the actual argument is a coindexed object and the dummy argument is allocatable or a pointer, the dummy argument must have intent `in`. Note that in the pointer case, the pointer association status of the dummy argument will be undefined unless the coindexed actual argument is on the executing image.

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, see Figure 2.

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

---

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

Figure 2: 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
```

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 3.

Figure 3: 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,:))
```

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 (new 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.

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

Automatic-array 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-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 12.8), 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 the specification part of a module or submodule, 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   Volatility

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 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 object.

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

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 (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. Similarly, if an allocatable coarray without the `save` attribute is declared within a `block` construct (new in Fortran 2008) and the coarray is still allocated when the block is left, there is an implicit deallocation.

10 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.

11 Coarrays with allocatable or pointer components

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 2006).

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

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

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

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

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

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

For this reason, an actual argument corresponding to a dummy pointer is not permitted to be a coindexed object and an actual argument that is a coindexed object is not permitted to have a pointer ultimate component. If an association with a target component on another image would otherwise be implied, the pointer becomes undefined. For example, this happens with the derived-type intrinsic assignments of Figure 4 are executed on an image with an index other than q.

Figure 4: 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
```

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:

```
type(something), allocatable ::  t[:]
  ...
allocate(t[*])        ! Allowed
allocate(t%ptr(n))    ! Allowed
allocate(t[q]%ptr(n)) ! 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 procedure reference through a procedure pointer component of a coindexed object,

for example,

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

is not permitted since the remote procedure target might be meaningless on the executing image. However, a reference though a type-bound procedure is allowed provided the type is not polymorphic; this ensures that the type and hence the procedure is the same on all images.

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

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.

12.1 `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 5: Read data on image 1 and broadcast it.

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

Figure 5 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.

12.2   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 12.8.

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 5, 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 5 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. Variables 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 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).

If 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)
   :
```

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)
```

because another image might define `x(8)` in a segment that is unordered with respect to this one.

12.3   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. Figure 6 shows an example that imposes the fixed order 1, 2, ... on images. 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 6: 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 )
```

Figure 7: 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
```

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 7, 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.

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` and has private components. It 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 its value is by executing the `lock` or `unlock` statement. If a lock variable is locked, it can be unlocked only by the image that locked it.

The example in Figure 8 illustrates the use of `lock` and `unlock` statements. Each image has its own queue; any image can add a task to any queue. 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

Figure 8: Using `lock` and `unlock` to manage queues.

```
module queue_manager
    use, intrinsic :: iso_fortran_env, only: lock_type
    type task
        :
    end type
    type(lock_type), private :: queue_lock[*]
    type(task), private :: queue(100)[*]
    integer, private :: queue_size[*]
contains
    subroutine get_task(job)
        type(task),intent(out) :: job
        lock(queue_lock)
            job=queue(queue_size)
            queue_size=queue_size-1
        unlock(queue_lock)
    end subroutine get_task
    subroutine put_task(job,image)
        type(task),intent(in) :: job
        integer,intent(in) :: image
        lock(queue_lock[image])
            queue_size[image]=queue_size[image]+1
            queue(queue_size[image])[image] = job
        unlock(queue_lock[image])
    end subroutine put_task
end module queue_manager
```

image. The effect in this example is that **get_task** has to wait if another image is adding a task to the queue and **put_task** has to wait if **get_task** is getting a task from the queue or another image is executing **put_task** for the same queue.

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

```
logical :: success
lock(queue_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 on image M is immediately followed in this sequence by execution of
a `lock` statement on image T, the segment that precedes the execution of the `unlock` statement
on image M precedes the segment that follows the execution of the `lock` statement on image T.

12.5   The `sync memory` statement

The execution of `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 9, 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 9: 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.)
else if (iam == q) then
   val = .true.
   do while (val)
      call atomic_ref(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 cause the

program to hang while doing nothing on image `p`.

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 final 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 implemenatations.

12.6 `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 a stopped image, 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.

12.7 Critical sections

Exceptionally, it may be necessary to limit execution of a piece of code to one image at a time. Such code is called a **critical section**. There is a new construct to delimit a critical section:

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

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

12.8 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 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. 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`, `lock`, `unlock`, 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.

13.1   Example of normal and error termination

The code fragment in Figure 10 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.

14   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 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

Figure 10: `stop` and `all 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 (...) all 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
```

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. 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.

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

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

`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   Image index functions

`image_index(coarray,sub)` returns the index of the image corresponding to the set of cosubscripts for `coarray` as a default integer scalar.

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

`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

17   References

Bailey, D., Harris, T., Saphir, W., van der Wijngaart, Woo, A. and Yarrow, M. (1995). The NAS Parallel Benchmarks 2.0. Report NAS-95-020, December, 1995.

Bonachea, D. (2002). GASNet Specification, v 1.1. Report No. UCB/CSD-02-1207, Computer Science Division (EEES), University of California, Berkeley, California 94729, USA.

Coarfa, C., Dotsenko, Y., Mellor-Crummey, J., Cantonnet, F, El-Ghazawi, T., Mohanti, A., Yao, Y., and Chavarría-Miranda, D. (2005). An evaluation of global address space languages: co-array fortran and unified parallel C. In PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, 36-47.

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

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

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

Numrich, R.W. (2005). Parallel numerical algorithms based on tensor notation and co-array Fortran syntax. *Parallel Computing*, 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/N1601-N1650/N1723.pdf`

18   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. 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. An actual argument corresponding to a dummy pointer is not permitted to be a coindexed object.

8. A coindexed object is allowed to be of a type with pointer ultimate components, but is not permitted as an actual argument. A pointer component becomes undefined if its value is copied from an associated pointer on another image.

9. 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.

10. Significant changes re termination have been made. Section 13 has been added to this report to explain how it now works.

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

12. `lcobound` and `ucobound` have been added.

13. A procedure reference through a procedure pointer component of a coindexed object is not permitted.

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

15. 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.

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

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

18. The `lock` and `unlock` statements have been added.

19. The atomic subroutines `atomic_define` and `atomic_ref` have been added.