

# Coarrays for Parallel Processing

See “Coarrays in the next Fortran standard”

John Reid, 21 April 2010

<ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1824.pdf>

Van Snyder

[van.snyder@jpl.nasa.gov](mailto:van.snyder@jpl.nasa.gov)

Jet Propulsion Laboratory  
California Institute of Technology



4 July 2011

Copyright © 2011 California Institute of Technology. Government sponsorship acknowledged.

## Summary

The design of the Coarray feature of Fortran 2008 was guided by answering the question “What is the smallest change required to convert Fortran to a robust and efficient parallel language.”

Two fundamental issues that any parallel programming model must address are work distribution and data distribution.

In order to coordinate work distribution and data distribution, methods for communication and synchronization must be provided.

Although originally designed for Fortran, the Coarray paradigm has stimulated development in other languages. X10, Chapel, UPC, Titanium, and class libraries being developed for C++ have the same conceptual framework.

## Work Distribution

Work distribution is addressed by a simple Single Program Multiple Data (SPMD) programming model.

When a program starts, several instances of it, called “images,” are initiated. The number of images, and their allocation to computing resources, is not addressed by the standard.

Images proceed independently, according to ordinary Fortran rules, until they synchronize.

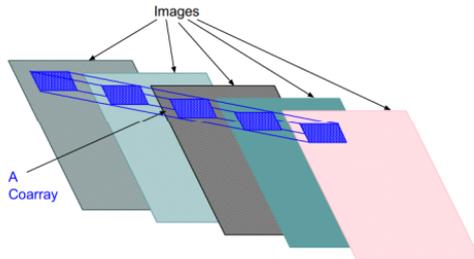
The intrinsic function `NUM_IMAGES()` returns the number of images initiated.

The images are numbered 1 : `NUM_IMAGES()`. The intrinsic function `THIS_IMAGE()` returns an image’s image number.

## Data Distribution

Data distribution is addressed by specifying the relationship of data objects and images using a syntax similar to normal Fortran array declarations.

The objects thus declared, called *Coarrays*, are declared similarly to any other data object, except that they have a sequence of bounds for *codimensions* declared within square brackets. The upper bound of the last codimension is required to be an asterisk, meaning “it’s specified when the program runs, not when it’s compiled.”



## Data Distribution (cont.)

The number of coelements of a coarray is the same as the number of images. A coelement of a coarray is either a scalar or an array. The rank + corank shall not exceed 15.

```
real :: A[-1:1,*] ! A coarray of scalars
real :: B(10)[*] ! A coarray of arrays
```

Images correspond to coelements in the same way that consecutive memory locations correspond to array elements: The first cosubscript varies most rapidly. Since the number of coelements is the same as the number of images, the largest value for the last cosubscript might depend upon the values of the other cosubscripts. For example, for the coarray scalar  $A[-1:1,*]$ , if there are ten images  $A[-1,4]$  is allowed (image 10) but  $A[0,4]$  is not (image 11).

## Data Distribution (cont.)

Coarrays can be of any type and kind, and can even be polymorphic, except that a coarray cannot have a coarray component. If polymorphic, every coelement has the same dynamic type.

Every subobject of a coarray is a coarray and has the same cobounds as the coarray. For example, if a coarray  $P$  is of a type that has a component  $X$ , then  $P[3]\%X$  is the  $X$  component of  $P$  on image 3.

The intrinsic function `IMAGE_INDEX` returns the image number corresponding to a coelement of a coarray.

The intrinsic function `THIS_IMAGE` returns the sequence of cosubscripts that references a coelement corresponding to a specified image.

## Coarrays and Procedures

Dummy arguments can be coarrays, in which case the associated actual argument shall also be a coarray. Restrictions on coarray arguments ensure that copy in / copy out argument association is never needed, to avoid a need for synchronization at every procedure reference.

If a dummy coarray is an array, it can be explicit shape, assumed shape, assumed size, or allocatable. If it is allocatable, it must ultimately be associated with the same non-dummy-argument coarray on every image.

Unless it is allocatable or a dummy argument, a coarray or an object with a coarray subobject is required to have the SAVE attribute. Automatic coarrays are not allowed. In Fortran 2008, all module variables implicitly have the SAVE attribute.

## Interoperability

Coarrays are not interoperable, since C does not have coarrays.

## Storage Association

Coarrays are not permitted in COMMON or EQUIVALENCE statements.

## Allocatable Coarrays

When a coarray is allocated, the dynamic type and the value of each bound, cobound, and length parameter shall be the same on every image.

## Communication

Data transfer between images is accomplished by referencing or assigning values to coelements that correspond to different images, e.g.

$$A[-1,1] = A[1,2]$$

transfers data from the coelement on image 6 to the one on image 1. Any image can do this; it isn't restricted to image 1 or 6. This also works for components:

$$P[1]\%X = P[6]\%X$$

If no cosubscripts appear, the referenced or defined coelement is the one on the same image as the one executing the reference or definition. E.g., on image 3,  $B = A$  is the same as  $B[3] = A[1,1]$ . Since  $B$  is an array the value of  $A[1,1]$  is broadcast to every element of  $B[3]$ .

## Communication (cont.)

Each statement that is executed is executed by the image on which the sequence of execution reaches that statement. The appearance of a cosubscript does not affect which image executes a statement. For example

```
B[1] = 1
```

is executed by every image on which the sequence of execution arrives at that statement. To cause it to be executed on only one image, e.g.

```
if ( this_image() == 1 ) B[1] = 1
```

causes the statement to be executed only on image 1.

## Coarrays and Pointers

Coarrays cannot be pointers, but they can have allocatable or pointer components. The targets of pointer components of coarrays are always on the same image as the coelement. If they are arrays, they are not required to have the same bounds on every image. The pointer target can be accessed on a different image. For example

$$x = z[q]\%p$$

means “Go to image  $q$  and get the address of  $z\%p$  on that image; then transfer the data to  $x$  on my image.” This is difficult to express using other parallel programming models.

## Coarrays and Pointers (cont.)

The association of a pointer component of a coarray with a target is always established, either by allocation or pointer assignment, on the same image as the coelement of which it is a component. A pointer on one image cannot be associated with a target on another image. The following are prohibited:

```
allocate ( z[q]%p )
z[q]%p => r
r => z[q]%p
```

but the following are allowed because they apply to a single image:

```
allocate ( z%p )
z%p => r
r => z%p
```

## Coarrays and Pointers (cont.)

If cross-image association would be implied, pointer components become undefined:

```
z = z[q] ! Assuming z has  
z[q] = z ! pointer  
z[q] = z[r] ! components
```

A coarray can have a procedure pointer component or a type-bound procedure. Invoking a procedure using a coelement on a different image does not imply remote procedure reference:

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

means “Go to image q and determine the target procedure;  
invoke it on the current image with arguments a[q] and x.”

## Synchronization

Most of the time, each image executes independently without regard to execution of other images. It is the program's responsibility to ensure that when it changes a coarray, no other image needs the old value, and when it references a coarray it accesses its current value. Image control statements are

- ▶ SYNC ALL, SYNC IMAGES and SYNC MEMORY
- ▶ ALLOCATE or DEALLOCATE involving a coarray, or END, END BLOCK or RETURN that results in automatic deallocation of a coarray
- ▶ CRITICAL and END CRITICAL
- ▶ STOP or END PROGRAM

## Execution Segments

On each image the sequence of statements executed before the first image control statement, or between the execution of two image control statements, is a *segment*.

Segments on a single image are always ordered according to the usual Fortran rules. Execution of corresponding image control statements on different images can ensure that segments on different images are ordered. Thus the set of all segments on all images is partially ordered.

Restrictions on what is permitted in segments that are not ordered with respect to each other give compilers scope for optimization.

## Restrictions on Segments

If a coarray is defined in a segment P, it must not be referenced, defined or become undefined in a segment Q that is not ordered with respect to P.

If the allocation status or pointer association status of a coarray subobject is changed in a segment P, that subobject must not be referenced in a segment Q that is not ordered with respect to P.

If a procedure invocation defines a noncoarray dummy argument, the associated actual argument shall not be referenced or defined in another segment unless that segment precedes the first one or succeeds the last one in that invocation. This allows (but does not require) copy-in / copy-out argument passing, without hidden synchronization.

## Synchronization statements

If an image executes a `SYNC ALL` statement, all other images must execute the same statement before that image can proceed. This orders the segments before the statement with respect to the segments after it.

If an image executes a `SYNC IMAGES` statement, all other images specified in that statement must execute the same statement before that image can proceed. This orders the segments before the statement with respect to the segments after it.

A `SYNC MEMORY` statement divides a segment on an image. It has no synchronization effect with respect to other images. An optimizer should not move statements across `SYNC MEMORY`, and should flush values from registers to memory when it is executed.

## Critical Sections

A critical section begins with a CRITICAL statement and ends with an END CRITICAL statement. If an image enters a critical section, no other image can enter the same critical section until the first one completes execution of it. This is used, for example, for “atomic updates:”

```
CRITICAL
    A[1] = A[1] + 1
END CRITICAL
```

This prevents another image from slipping its update between the load, add, and store, which would result in adding 1 to A[1] instead of 2.

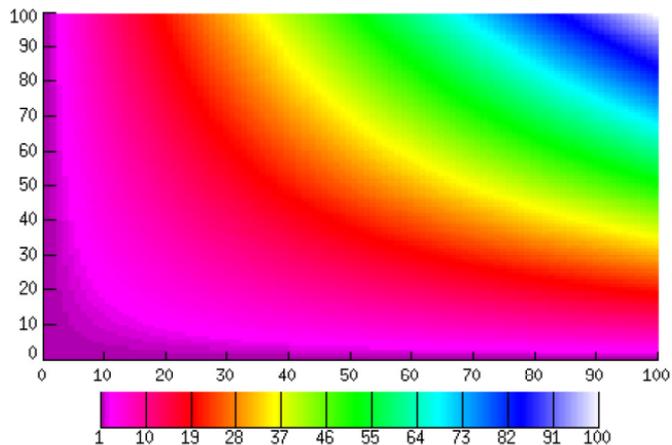
## Locks

A type LOCK\_TYPE is provided, along with LOCK and UNLOCK statements, for use in situations where critical sections impose too rigid a structure.

## Example – Jacobi relaxation

```
real :: A[m,*], Diff[m,*], NewA
integer :: I, J
i = mod(this_image()-1,m) + 1 ! my row num
j = (this_image()-1)/m + 1 ! my col num
if ( i == 1 ) then ; a = j ! bottom row
else if ( j == 1 ) then ; a = i ! left col
else ! i > 1 and j > 1 here
  a = 0.0
  do
    newA = ( a + a[i-1,j] + a[i+1,j] + &
             & a[i,j-1] + a[i,j+1] ) / 5.0
    diff = abs(newA-a)
    if ( co_maxval(diff) < 0.01 ) exit
    a = newA
  end do
end if
```

## Example – Jacobi relaxation



## Input/Output

Standard input is connected only on image 1.

Each other preconnected unit is connected only on the executing image, and the connected file is different from any preconnected file on any other image. A processor is permitted but not required to interleave the standard output and standard error streams from several images.

An OPEN statement opens a connection only on the executing image. Whether a named file on one image is the same as a file of the same name on a different image is processor dependent. A named file shall not be connected on more than one image.

## Program Termination

A STOP statement or an END PROGRAM statement is a synchronization statement. The program doesn't finish execution until all images execute a STOP or END PROGRAM statement, or any image executes an ALL STOP statement or initiates error termination.

An ALL STOP statement terminates execution of the image that executes it, and terminates execution of all other images as soon as possible. It is intended for error or emergency termination.

The program can also be terminated immediately on all images if an error condition arises during the execution of one of them, for example, by deallocating a deallocated allocatable variable without a STAT= specifier in the DEALLOCATE statement.

## Why not just use MPI or OpenMP?

MPI is far more verbose, especially for structures in Fortran. This is because of the inherent rigidity of procedure references, which don't have access to the full richness of the syntax without excessive complication.

MPI is inefficient on shared-memory systems; with Coarrays, your compiler + runtime should choose the most efficient transport for each transaction.

OpenMP doesn't scale to large systems.

## Implementations

Cray has provided Coarrays for over a decade.

Intel 12.0 provides Coarrays.

Gnu Fortran 95 (g95) provides incomplete support for Coarrays, and hopes for complete support soon. `gfortran` supports Coarray syntax, but provides only single-image execution.

Coarrays are in the 2008 Fortran standard, so most other major Fortran compiler vendors (IBM, Oracle/Sun, NAG, ...) will be supporting them.

Rice University is working on a translator to Fortran 90 + ARMCI (Aggregate Remote Memory Copy Interface).

## More information

### **Fortran 2008 standard**

<http://j3-fortran.org/doc/standing/archive/007/10-007r1.pdf>

### **Coarrays in the next Fortran standard**

John Reid, 21 April 2010

<ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1824.pdf>

### **Using MPI (2nd ed.): Portable Parallel Programming with the Message-Passing Interface**

William Gropp, Ewing Lusk, Anthony Skjellum, MIT Press  
(1999) ISBN 0-262-57132-3

<http://www.co-array.org> (a bit out of date).