Introduction to Fortran Coarrays

John Reid, ISO Fortran Convener,

JKR Associates and

Rutherford Appleton Laboratory

Abstract

This talk will

- explain the objectives of coarrays,
- give a quick summary of their history,
- introduce the coarray features in Fortran 2008, and
- say something about the further coarray features that we are planning.

I will assume some knowledge of Fortran but not that you know it all.

I will remind you about features as you meet them for the first time.

Design objectives

Coarrays are the brain-child of Bob Numrich (Minnesota Supercomputing Institute, formerly Cray) and date from his work in the mid 1990s.

The original design objectives were for

- A simple extension to Fortran
- Small demands on the implementors
- Retain optimization between synchronizations
- Make remote references apparent
- Provide scope for optimization of communication

Coarrays has been implemented by Cray for more than ten years. Intel now includes them and they are being added to gfort.

Summary of coarray model

- SPMD - Single Program, Multiple Data
- Replicated to a number of **images** (probably as executables)
- Number of images fixed during execution
- Each image has its own set of variables
- Coarrays are like ordinary variables but have second set of subscripts [ ] for access between images
- Images mostly execute asynchronously
- Synchronization: **`sync all`**, **`sync images`**, **`stop`**, **`lock`**, **`unlock`**, **`allocate`**, **`deallocate`**, **`move_alloc`**, **`critical`** construct
- Intrinsics: **`this_image`**, **`num_images`**, **`image_index`**

Examples of coarray syntax

```
real,save :: r[*], s[0:*] ! Scalar coarrays
real,save,codimension[*] :: x(n) ! Array coarray
type(u),save :: u2(m,n)[np,*]
  ! Coarrays always have assumed cosize
  ! (equal to number of images)
real :: t                    ! Local variable
integer p, q, index(n)  ! Local variables
:
t = s[p]
x(:) = x(:)[p]
! Reference without [] is to local object
x(:)[p] = x(:)
u2(i,j)%b(:) = u2(i,j)[p,q]%b(:)
```

Implementation model

- Usually, each image resides on one core.
- However, several images may share a core (e.g. for debugging) and one image may execute on a cluster (e.g. with OpenMP).
- A coarray has the same set of bounds on all images, so the compiler may arrange that it occupies the same set of addresses within each image (known as *symmetric memory*).
- This allows each image to calculate the memory address of an element on another image.

Synchronization

With a few exceptions, the images execute asynchronously.

If syncs are needed, the user supplies them explicitly.

Barrier on all images **`sync all`**

Wait for others **`sync images`**(*image-set*)

Need not be same statement on the images involved.

Limit execution to one image at a time

```
   critical
      block
   end critical
```

Limit execution in a more flexible way

```
 use :: iso_fortran_env
type(lock_type),save :: lock_var[*]
lock(lock_var[6])
p[6] = p[6] + 1
unlock(lock_var[6])
```

The synchronization statements are known as **image control** statements

The sync images statement

Example: make other images to wait for image 1:

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

Correction: lines reordered in else block.

Execution segments

On an image, the statements executed up to the first image control statement or after one and up to the next is known as a **segment**.

For example, this code reads a value on image 1 and broadcasts it.

```
real,save :: p[*]
  :                        ! Segment 1
sync all                   ! Segment 1
if(this_image()==1)then    ! Segment 2
   read (*,*) p            !   :
   do i = 2, num_images()!   :
      p[i] = p             !   :
   end do                  !   :
end if                     !   :
sync all                   ! Segment 2
  :                        ! Segment 3
```

Execution segments (cont)

The normal rules of statement execution on a single image and the synchronization statements together ensure a partial ordering of all the segments.

Important rule: if a variable is defined in a segment, it must not be referenced, defined, or become undefined in a another segment unless the segments are ordered.

It is up to the programmer to ensure this.