Coarrays in Fortran 2008 and Fortran 2018

John Reid, Former ISO Fortran Convener,

JKR Associates and

Rutherford Appleton Laboratory

Tokyo

2 August 2019

Abstract

I will summarize the coarray features of Fortran 2008, which provide a pleasant way to perform parallel programming.

I will explain main additions in Fortran 2018:

A. Teams
B. Collectives
C. Events
D. Continued execution with failed images

It was D that was the most difficult to design.

Design objectives

Coarrays were 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 implementers
- Retain optimization between synchronizations
- Make remote references apparent
- Provide scope for optimization of communication

Have been implemented by Cray for 20 years.

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`, `lock`, `unlock`, `critical` construct, `allocate`, `deallocate`
- Intrinsics: `this_image`, `num_images`, `image_index`

Examples of coarray syntax

```
real,save :: r[*], s[0:*] ! Scalar coarrays
real,save :: 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 node (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

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*)

Limit execution to one image at a time

```
      critical
        block
      end critical
```

These are known as **image control** statements

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.

```
  :                          ! 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.

Dynamic coarrays

Only dynamic form: the allocatable coarray.

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

The bounds, cobounds, and length parameters must not vary between images.

All images synchronize at an `allocate` or `deallocate` statement so that they can all perform their allocations and deallocations in the same order (for symmetric memory).

Coarray dummy arguments

A dummy argument may be a coarray. It may be of explicit shape, assumed size, assumed shape, or allocatable.

```
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(:)[:,:]
```

There are rules to ensure that copy-in copy-out of a coarray is never needed.

Structure components

A coarray may be of a derived type with allocatable or pointer components.

Provides a simple but powerful mechanism for cases where the size varies from image to image, avoiding loss of optimization.

Pointers must have targets in their own image:

```
q => z[i]%p        ! Not allowed
allocate(z[i]%p) ! Not allowed
```

Fortran 2008 support in compilers

Full support: Cray, Intel

Support coarrays: gfortran

Near full support of coarrays: Fujitsu

Significant support of other features (alphabetic order): gfortran, IBM, NAG, NEC

For details, see Fortran Forum, August 2019

Fortran 2018

The main additions in Fortran 2018 were:

A. Teams

B. Collectives

C. Events

D. Continued execution with failed images

Teams

Needed for independent computations on subsets of images.

Code that has been written and tested on whole machine should run on a team.

Therefore, image indices need to be relative to the team.

Collective activities, including syncs and allocations, need to be relative to the team.

team_type and form team

The intrinsic module `iso_fortran_env` contains a derived type `team_type`. A scalar object of this type identifies a team of images.

The same `form team` statement must be executed on all images of a team to form subteams

```
form team(number,new_team)
```

Images with the same value of number form a new team.

All images of the current team synchronize.

`change team` construct

```
change team (team,local[*]=>coarray)
  ! Block executed as a team
  if(team_number()==1) then ! New intrinsic
      : ! Code for team 1
  else
      :
end team
```

Associating `local` with `coarray` allows corank and cobounds to change. Other attributes are unchanged.

The new teams synchronize at `change team` and `end team`.

Changing teams is likely to be costly – avoid doing it often.

Accessing another team

```
real, save :: a(n)[*]
type(team_type) :: initial, block
initial = get_team() ! New intrinsic
i = ...
form team(i,block)
change team (block)
   :
   sync team(initial) ! New statement
   a(k) = a(1)[me+1,team=initial]
end team
```

Collectives

The collective subroutines are:

`co_broadcast, co_max, co_min,`
`co_sum, co_reduce.`

Invoked by the same statement on all images of the team and involve synchronization within them, but not necessarily at start and end.

The main argument is not required to be a coarray.

Events

Events are useful if one or more images need to do something before another image can continue.

For example, in the multifrontal method for factorizing a sparse matrix, work at a node of the assembly tree has to wait for all the work at its child nodes to be completed.

Event variable

An event variable is a scalar coarray of type `event_type`. It contains a count which increases by 1 each time the event is "posted".

```
use iso_fortran_env
type(event_type), save :: event[*]
  :
event post(event[i]) ! Atomic
  :
if(this_image()==i) then
  event wait(event)
  ! Waits until count >= 1, then atomically
  ! decreases it by 1 and continues
```

failed_images intrinsic function

`failed_images()`

Returns an integer array holding image indices of known failed images in the current team.

`failed_images(team)`

Returns an integer array holding image indices of known failed images in `team`.

Testing for failed images in image control statements

```
parent = get_team()
change team (team_a)
      :
   sync_all(parent,stat=st)
   if (st==stat_failed_image) exit
end team
sync all(stat=st)
if (st==stat_failed_image) then
   : Deal with failure
end if
```

Testing for failed image in a remote reference

```
use iso_fortran_env
   :
a = b[image,stat=st]
if (st==stat_failed_image) then
   : Deal with failure
end if
```

Advantages of coarrays

References to local data are obvious as such.

Easy to maintain code - more concise than MPI and easy to see what is happening

Integrated with Fortran - type checking, type conversion on assignment, ...

The compiler can optimize communication

Local optimizations still available

Does not make severe demands on the compiler, e.g. for coherency.