Coarrays in Fortran 2008

John Reid, ISO Fortran Convener,
JKR Associates and
Rutherford Appleton Laboratory

The technical content of Fortran 2008 was decided at the meeting in November 2008. It is now in Final CD ballot.

Coarrays remain the major extension from Fortran 2003, but there have been considerable changes since the first PGAS conference in 2005.

The aim of this talk is provide a fresh overview of all the features.

PGAS 2009,
Washington,
7 October 2009.

Design objectives

Coarrays are the brain-child of Bob Numrich (Minnesota Supercomputing Institute, formerly Cray).

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

A subset has been implemented by Cray for some ten years.

Coarrays have recently been added to the g95 compiler.

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 in [ ] for access between images
- Images mostly execute asynchronously
- Synchronization: `sync all`, `sync images`, `sync memory`, `lock`, `unlock`, `allocate`, `deallocate`, `critical` construct
- Intrinsics: `this_image`, `num_images`, `image_index`, `atomic_define`, `atomic_ref`

Full summary: WG5 N1787 (Google WG5)

Examples of coarray syntax

```
real :: r[*], s[0:*] ! Scalar coarrays
real :: x(n)[*]      ! Array coarray
type(u) :: u2(m,n)[np,*]
! Coarrays always have assumed
! cosize (equal to number of images)

real :: t                  ! Local
integer p, q, index(n)     ! variables
     :
t = s[p]
x(:) = x(:)[p]
! Reference without [] is to local part
x(:)[p] = x(:)
u2(i,j)%b(:) = u2(i,j)[p,q]%b(:)
```

Implementation model

Usually, each image resides on one processor.

However, several images may share a processor (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).*

On a shared-memory machine, a coarray might be implemented as a single large array.

On any machine, a coarray may be implemented so that each image can 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)
```

Limit execution to one image at a time

```
critical
   :
end critical
```

Limit execution in a more flexible way

```
lock(lock_var[6])
   p[6] = p[6] + 1
unlock(lock_var[6])
```

These are known as **image control statements**.

The sync images statement

Ex 1: 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
```

Ex 2: impose the fixed order 1, 2, ... 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 )
```

Execution segments

On an image, the sequence of statements executed before the first image control statement or between two of them is known as a **segment**.

For example, this code reads data on image 1 and broadcasts them.

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

Execution segments (cont)

Here we show three segments.

On any image, these are executed in order, Segment 1, Segment 2, Segment 3.

The `sync all` statements ensure that Segment 1 on any image precedes Segment 2 on any other image and similarly for Segments 2 and 3.

However, two segments 1 on different images are unordered.

Overall, we have a partial ordering.

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

Atomic subroutines

```
call atomic_define(atom[p],value)
call atomic_ref(value,atom[p])
```

The effect of executing an atomic subroutine is as if the action occurs instantaneously, and thus does not overlap with other atomic actions that might occur asynchronously.

It acts on a scalar variable of type `integer(atomic_int_kind)` or `logical(atomic_logical_kind)`.

The kinds are defined in an intrinsic module.

The variable must be a coarray or a coindexed object.

Spin-wait loop

Atomics allow the spin-wait loop, e.g.

```
  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
```

Here, segment 1 on `p` and segment 2 on `q` are unordered but `locked` is `atomic` so it is OK. Image `q` will emerge from its spin when it sees that `locked` has become false.

Dynamic coarrays

Only dynamic form: the allocatable coarray.

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

All images synchronize at an `allocate` or `deallocate` statement so that they can all perform their allocations and deallocations in the same order. The bounds, cobounds, and length parameters must not vary between images.

An allocatable coarray may be a component of a structure provided the structure and all its ancestors are scalars that are neither pointers nor coarrays.

A coarray is not allowed to be a pointer.

Non-coarray dummy arguments

A coarray may be associated as an actual argument with a non-coarray dummy argument (nothing special about this).

A coindexed object (with square brackets) may be associated as an actual argument with a non-corray dummy argument. Copy-in copy-out is to be expected.

These properties are very important for using existing code.

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(:)[:,:]
```

Where the bounds or cobounds are declared, there is no requirement for consistency between images. The local values are used to interpret a remote reference. Different images may be working independently.

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

Coarrays and SAVE

Unless allocatable or a dummy argument, a coarray must be given the SAVE attribute.

This is to avoid the need for synchronization when coarrays go out of scope on return from a procedure.

Similarly, automatic-array coarrays

```
subroutine subr (n)
  integer :: n
  real :: w(n)[*]
```

and array-valued functions

```
function fun (n)
  integer :: n
  real :: fun(n)[*]
```

are not permitted, since they would require automatic synchronization, which would be awkward.

Structure components

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

Pointers must have targets in their own image:

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

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

Program termination

The aim is for an image that terminates normally (`stop` or `end program`), to remain active so that its data is available to other executing images, while an error condition leads to early termination of all images.

Termination is *normal* or *error* and occurs in three steps: *initiation*, *synchronization*, and *completion*. Each image completes the kind of termination that it initiates.

Data on an image terminating normally is available to others until they all reach the synchronization.

If an image hits an error condition or executes `all stop`, it and all other images that have not initiated termination initiate error termination.

Input/output

Default input (`*`) is available on image 1 only.

Default output (`*`) and error output are available on every image. The files are separate, but their records will be merged into a single stream or one for the output files and one for the error files.

To order the writes from different images, need synchronization and the `flush` statement.

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.

Optimization

Most of the time, the compiler can optimize as if the image is on its own, using its temporary storage such as cache, registers, etc.

There is no coherency requirement while unordered segments are executing. The programmer is required to follow the rules.

The compiler also has scope to optimize communication.

Recent changes

- Locks have been added.
- Atomics have been added.
- Concept of ‘segments’ has been added.
- Careful consideration given to termination.
- While a ‘core’ set remains in Fortran 2008, the following features have moved into a separate Technical Report on ‘Enhanced Parallel Computing Facilities’:

1. The collective intrinsic subroutines.
2. Teams and features that require teams.
3. The `notify` and `query` statements.
4. File connected on more than one image, unless default output or default error.

A comparison with MPI

A colleague (Ashby, 2008) recently converted most of a large code, SBLI, a finite-difference formulation of Direct Numerical Simulation (DNS) of turbulance, from MPI to coarrays using a small Cray X1E (64 processors).

Since MPI and coarrays can be mixed, he was able to do this gradually, and he left the solution writing and the restart facilites in MPI.

Most of the time was taken in halo exchanges and the code parallelizes well with this number of processors. The speeds were very similar.

The code clarity (and maintainability) was much improved. The code for halo exchanges, excluding comments, was reduced from 176 lines to 105 and the code to broadcast global parameters from 230 to 117.

Advantages of coarrays

- Easy to write code – the compiler looks after the communication
- 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.

References

Ashby, J.V. and Reid, J.K (2008). Migrating a scientific application from MPI to coarrays. CUG 2008 Proceedings. RAL-TR-2008-015, see `http://www.numerical.rl.ac.uk/reports/reports.shtml`

Reid, John (2008). *Coarrays in the next Fortran Standard.* ISO/IEC/JTC1/SC22/ WG5 N1787, see `ftp://ftp.nag.co.uk/sc22wg5/N1751-N1800`

WG5(2008). *Final CD revision of the Fortran Standard.* ISO/IEC/JTC1/SC22/ WG5 N1787, see `ftp://ftp.nag.co.uk/sc22wg5/N1751-N1800`