Chapter 34
Coarray Fortran

Science is a wonderful thing if one does not have to earn one's living at it.

Einstein

Aim
The aims of this chapter is to provide a short introduction to coarray programming in Fortran.

34.1 Introduction

Coarrays were the major component of the Fortran 2008 standard. As stated earlier they are based on a single program multiple data model. Coarrays are a simple parallel programming extension to Fortran. They are effectively variables that can be shared across multiple instances of the same program or images in Fortran terminology.

Coarray variables look like conventional Fortran arrays, except that they use [] brackets instead of () brackets. In the simple declaration below

```
character(len=20) :: name[*]='*****'
```

We declare name to be a `coarray` and the * in the [] brackets means that the bounds of the coarray are calculated at run time, rather than compile time.

```
read  *, name
```

is a reference to the coarray on the current image.

We can then use the following statement

```
  name[i] = name
```


I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*,
https://doi.org/10.1007/978-3-319-75502-1_34

to broadcast the value read in to each of the other images.

Note the Fortran coarray syntax here. We use the [] brackets to reference the coarray variable on other images and the omission of the [] brackets is a reference to the coarray variable on the current image.

34.2 Some Basic Coarray Terminology

The following is taken from the Fortran 2018 standard and covers some of the basic coarray terminology.

- `codimension` attribute - The `codimension` attribute species that an entity is a coarray. The coarray-spec specifies its corank or corank and cobounds.
- Allocatable coarray - A coarray with the allocatable attribute has a specified corank, but its cobounds are determined by allocation or argument association.
- Explicit-coshape coarray - An explicit-coshape coarray is a named coarray that has its corank and cobounds declared by an explicit-coshape-spec.
- Coindexed named objects - A coindexed-named-object is a named scalar coarray variable followed by an image selector.
- Image selectors - An image selector determines the image index for a coindexed object.
- Image execution control and image control statements - The execution sequence on each image is specified in 5.3.5 of the standard.
- Execution of an image control statement divides the execution sequence on an image into segments. Each of the following is an image control statement:
  - `sync all` statement;
  - `sync images` statement;
  - `sync memory` statement;
  - `allocate` or `deallocate` statement that has a coarray allocate-object;
  - `critical` or `end critical`;
  - `lock` or `unlock` statement;
  - Any statement that completes execution of a block or procedure and which results in the implicit deallocation of a coarray;
  - `stop` statement;
  - `end` statement of a main program.
- Coarray - A coarray is a data entity that has nonzero corank; it can be directly referenced or defined by any image. It may be a scalar or an array.
- Coarray dummy variables - If the dummy argument is a coarray, the corresponding actual argument shall be a coarray and shall have the `volatile` attribute if and only if the dummy argument has the `volatile` attribute.
- Some coarray intrinsics
  - `image_index` - convert a cosubscript to an image index

- `lcobound` - cobounds of a coarray
- `num_images` - the number of images
- `this_image` - image index or cosubscripts
- `ucobound` - cobounds of a coarray

Let us look now at some simple examples.

34.3 Example 1: Hello World

The first is the classic Hello world.

```
program ch3401

  implicit none

  print *, ' Hello world from image ', &
    this_image()

end program ch3401
```

Here is the output from the Intel compiler.

```
  Hello world from image            5
  Hello world from image            3
  Hello world from image            4
  Hello world from image            8
  Hello world from image            1
  Hello world from image            6
  Hello world from image            2
  Hello world from image            7
```

Here is sample output from the Cray Archer service.

```
  Hello world from image  16
  Hello world from image  6
  Hello world from image  13
  Hello world from image  25
  Hello world from image  34

lines deleted

  Hello world from image  38
  Hello world from image  44
```

```
 Hello world from image  35
 Hello world from image  28
 Hello world from image  33
 Hello world from image  32
 Hello world from image  30
 Hello world from image  29
```

The output is obviously very similar to the corresponding MPI and OpenMP versions.

34.4 Example 2: Broadcasting Data

Here is a simple program that broadcasts data from one image to the rest. This is a common requirement in parallel programming.

```
program ch3402

  implicit none

  integer :: i
  character (len=20) :: name [ * ] = '*****'

  print 100, name, this_image()

  if (this_image()==1) then
    print *, ' Type in your name'
    read *, name
    do i = 2, num_images()
      name [ i ] = name
    end do
  end if

  sync all

  print 100, name, this_image()
100 format (1x, ' Hello ', a20, ' from image ', &
    i3)

end program ch3402
```

Here is the output from the Intel compiler.

```
 Hello *****               from image   1
```

```
  Hello *****                from image   3
  Hello *****                from image   5
  Hello *****                from image   7
  Hello *****                from image   2
  Hello *****                from image   4
  Hello *****                from image   8
  Type in your name
  Hello *****                from image   6
Jane
  Hello Jane                 from image   4
  Hello Jane                 from image   8
  Hello Jane                 from image   2
  Hello Jane                 from image   6
  Hello Jane                 from image   7
  Hello Jane                 from image   3
  Hello Jane                 from image   5
  Hello Jane                 from image   1
```

Again no particular ordering of the image numbers.

34.5 Example 3: Parallel Solution for pi Calculation

```
include 'precision_module.f90'

include 'timing_module.f90'

program ch3403

  use precision_module
  use timing_module

  implicit none

  real (dp) :: fortran_internal_pi
  real (dp) :: partial_pi
  real (dp) :: coarray_pi
  real (dp) :: width
  real (dp) :: total_sum
  real (dp) :: x
  real (dp), codimension [ * ] :: partial_sum

  integer :: n_intervals
```

```
  integer :: i
  integer :: j
  integer :: current_image
  integer :: n_images

  fortran_internal_pi = 4.0_dp*atan(1.0_dp)
  n_images = num_images()
  current_image = this_image()

  if (current_image==1) then
    print *, ' Number of images = ', n_images
  end if

  n_intervals = 100000

  do j = 1, 5
    if (current_image==1) then
      call start_timing()
    end if
    width = 1.0_dp/real(n_intervals, dp)
    total_sum = 0.0_dp
    partial_sum = 0.0_dp
    do i = current_image, n_intervals, n_images
      x = (real(i,dp)-0.5_dp)*width
      partial_sum = partial_sum + f(x)
    end do
    partial_sum = partial_sum*width
    sync all
    if (current_image==1) then
      do i = 1, n_images
        total_sum = total_sum + partial_sum [ i &
          ]
      end do
      coarray_pi = total_sum
      print 100, n_intervals, time_difference()
      print 110, coarray_pi, abs(coarray_pi- &
        fortran_internal_pi)
    end if
    n_intervals = n_intervals*10
    sync all
  end do

100 format (' n intervals = ', i12, ' time =', &
    f8.3)
110 format (' pi = ', f20.16, /, &
```

```
      ' difference = ', f20.16)

contains

  real (dp) function f(x)
    implicit none
    real (dp), intent (in) :: x

    f = 4.0_dp/(1.0_dp+x*x)

  end function f

end program ch3403
```

Here is the output from the Intel compiler.

```
Number of images =            8
2011/ 6/10 13:40:48 479
n intervals =       100000 time =   0.004
pi =   3.1415926535981260
difference =   0.0000000000083329
2011/ 6/10 13:40:48 486
n intervals =      1000000 time =   0.004
pi =   3.1415926535898802
difference =   0.0000000000000870
2011/ 6/10 13:40:48 490
n intervals =     10000000 time =   0.012
pi =   3.1415926535897936
difference =   0.0000000000000004
2011/ 6/10 13:40:48 500
n intervals =    100000000 time =   0.105
pi =   3.1415926535897749
difference =   0.0000000000000182
2011/ 6/10 13:40:48 605
n intervals =   1000000000 time =   0.992
pi =   3.1415926535898455
difference =   0.0000000000000524
```

Here is the output from the Cray compiler.

```
 Number of images =  48
2015/ 3/21  1:11:50 130
n intervals =       100000 time =   0.005
pi =   3.1415926535981265
```

```
 difference =   0.0000000000083333
 2015/ 3/21  1:11:50 135
 n intervals =      1000000 time =   0.000
 pi =   3.1415926535898762
 difference =   0.0000000000000830
 2015/ 3/21  1:11:50 135
 n intervals =     10000000 time =   0.001
 pi =   3.1415926535897953
 difference =   0.0000000000000022
 2015/ 3/21  1:11:50 136
 n intervals =    100000000 time =   0.006
 pi =   3.1415926535897905
 difference =   0.0000000000000027
 2015/ 3/21  1:11:50 142
 n intervals =   1000000000 time =   0.054
 pi =   3.1415926535897949
 difference =   0.0000000000000018
```

We get the time improvement we have seen with both the MPI and OpenMP solutions.

34.6 Example 4: Work Sharing

This example looks at one way of splitting work up between images. We use the image number to determine which image does which work. It is a coarray version of the MPI work sharing example.

```
program ch3404
  implicit none
  integer :: n, i, j
  integer :: me, nim, start, end
  integer, parameter :: factor = 5
  integer, dimension (1:factor), &
    codimension [ * ] :: x

  nim = num_images()
  me = this_image()
  n = nim*factor
  x = 0
  start = factor*(me-1) + 1
  end = factor*me
  j = 1
  do i = start, end
```

```
      x(j) = i*factor
      print *, 'on image ', me, 'j = ', j, &
        ' x(j) = ', x(j)
      j = j + 1
    end do
    sync all
    if (me==1) then
      print *, 'coarray x on image ', me, ' is: ', &
        x
      do i = 2, nim
        print *, 'coarray x on image ', i, &
          ' is: ', x(:) [ i ]
      end do
    end if
  end program ch3404
```

The following statements define the start and end points for the array processing for each image:

```
start = factor*(me-1) + 1
end   = factor*me
```

and partitions the work between the images. Each image will have its own start and end values. The following do loop does the work:

```
do i=start,end
   x(j) = i*factor
   print*,'on image ',me, 'j = ',j,' x(j) = ',x(j)
   j    = j + 1
end do
```

We need the

```
sync all
```

to ensure that each image has completed before further processing, and we then print out the data from each image on image 1.

Here is a subset of the output from the Intel compiler. This example runs on 8 images.

```
on image       2 j =         1  x(j) =       30
on image       7 j =         1  x(j) =      155
on image       8 j =         1  x(j) =      180
on image       8 j =         2  x(j) =      185
on image       8 j =         3  x(j) =      190
```

```
 on image      8 j =        4  x(j) =     195
 on image      8 j =        5  x(j) =     200
 on image      6 j =        1  x(j) =     130
 on image      6 j =        2  x(j) =     135
 on image      6 j =        3  x(j) =     140
...
...
...
 coarray x on image            1  is:
             5          10
15
          20          25
 on image      4 j =        1  x(j) =      80
 on image      4 j =        2  x(j) =      85
 on image      4 j =        3  x(j) =      90
 on image      4 j =        4  x(j) =      95
 on image      4 j =        5  x(j) =     100
 coarray x on image            2  is:
            30          35
40
          45          50
 coarray x on image            3  is:
            55          60
65
          70          75
 coarray x on image            4  is:
            80          85
90
          95         100
 coarray x on image            5  is:
           105         110
115dir
         120         125
 coarray x on image            6  is:
           130         135
140
         145         150
 coarray x on image            7  is:
           155         160
165
         170         175
 coarray x on image            8  is:
           180         185
190
         195         200
```

Here is a sample of the output from the Cray compiler on the Archer service. This example runs on 48 images.

```
 on image  1 j =  1  x(j) =  5
 on image  1 j =  2  x(j) =  10
 on image  3 j =  1  x(j) =  55
 on image  3 j =  2  x(j) =  60

stuff deleted

 on image  22 j =  5  x(j) =  550
 coarray x on image  1  is:  5,  10,  15,  20,  25
 on image  21 j =  1  x(j) =  505

stuff deleted

 on image  20 j =  3  x(j) =  490
 on image  6 j =  3  x(j) =  140
 on image  13 j =  2  x(j) =  310
 on image  6 j =  4  x(j) =  145

stuff deleted

 on image  7 j =  1  x(j) =  155
 on image  10 j =  2  x(j) =  235

stuff deleted

 on image  27 j =  2  x(j) =  660
 on image  41 j =  4  x(j) =  1020
 on image  28 j =  2  x(j) =  685

stuff deleted

 on image  33 j =  5  x(j) =  825
 on image  36 j =  5  x(j) =  900
 on image  40 j =  1  x(j) =  980

stuff deleted

 on image  40 j =  2  x(j) =  985
 on image  40 j =  3  x(j) =  990
 on image  40 j =  4  x(j) =  995
 on image  40 j =  5  x(j) =  1000
```

```
 on image  45 j =  4  x(j) =  1120
 on image  46 j =  5  x(j) =  1150
 on image  45 j =  5  x(j) =  1125
Application 13271719 resources: utime ~7s,
stime ~52s, Rss ~4288,
inblocks ~22292, outblocks ~39436
```

34.7 Summary

This chapter has looked briefly at some of the simple syntax of coarrays using a small set of examples. We have also seen the timing benefits that coarray programming can offer in the solution of the same problem.

34.8 Problem

34.1 Compile and run the examples in this chapter with your compiler.