

More Coarray Features

PRACE Autumn School, October 29th 2010

Parallel Programming with Coarray Fortran

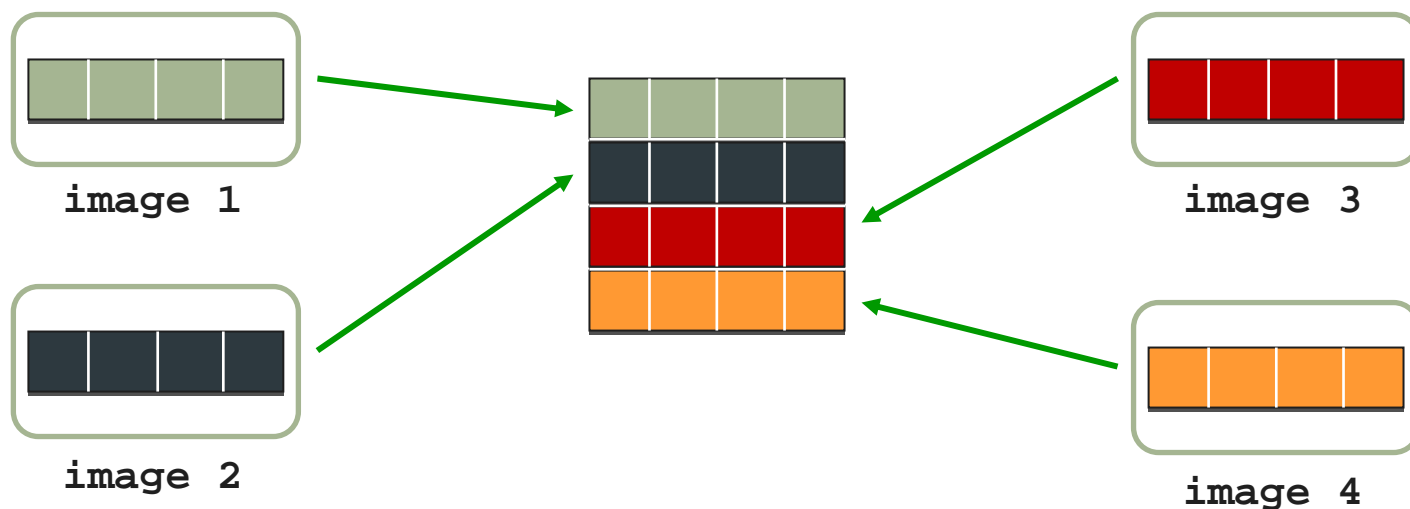
Overview

- Multiple Dimensions and Codimensions
- Allocatable Coarrays and Components of Coarray Structures
- Coarrays and Procedures

2D Data

- Corray Fortran has a “bottom-up” approach to global data
 - assemble rather than distribute
 - unlike HPF (“top-down”) or UPC shared distributed data
- Can assemble a 2D data structure from 1D arrays

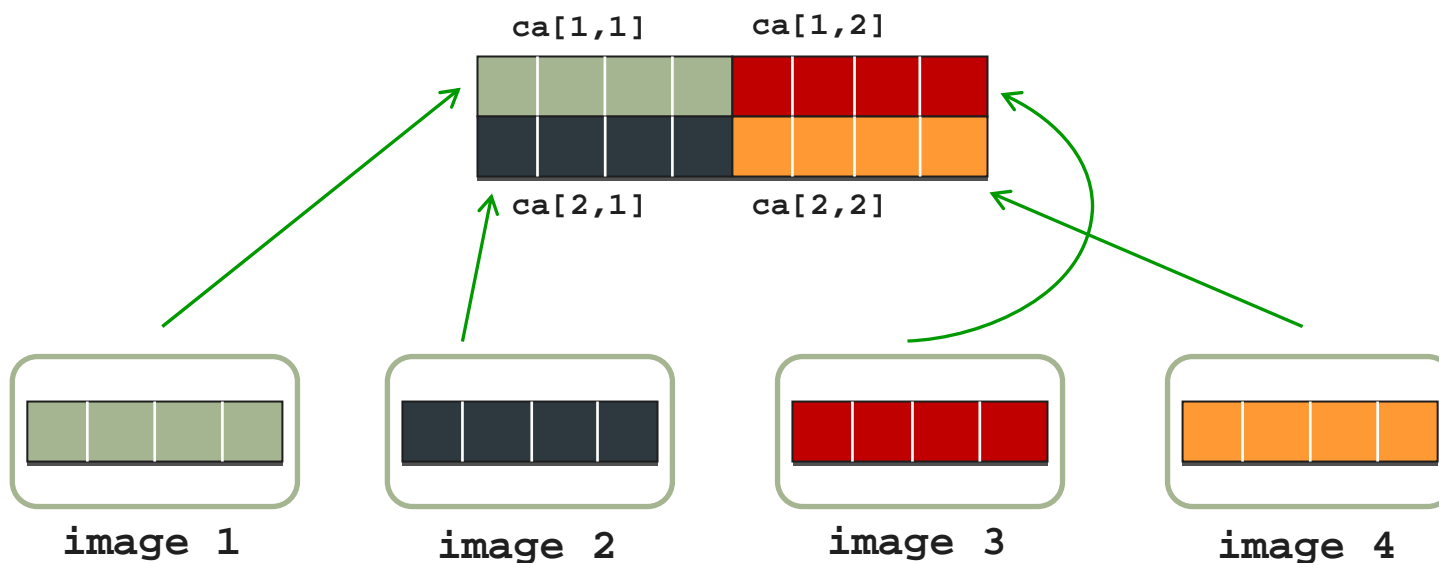
```
integer :: ca(4)[*]
```



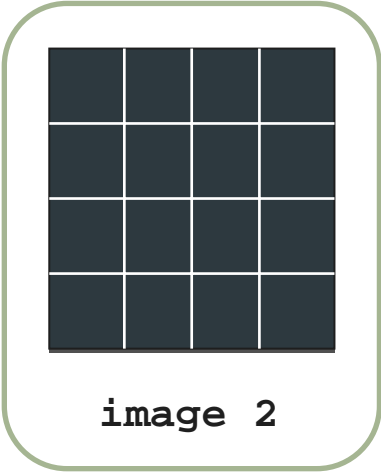
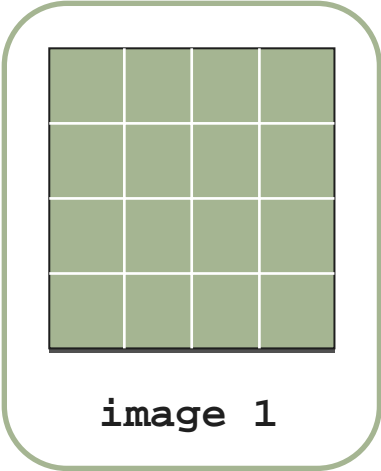
2D Data

- However, images are not restricted to a 1D arrangement
- For example, we can arrange images in 2x2 grid
 - coarrays with 2 codimensions

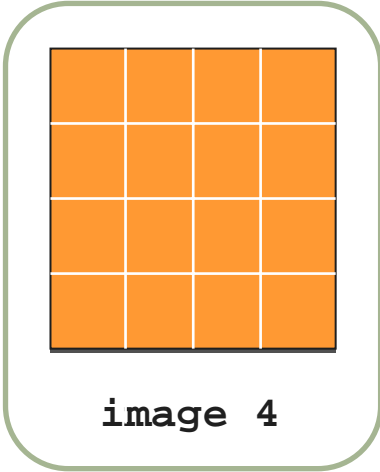
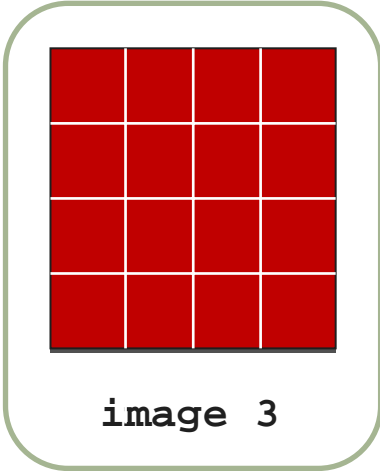
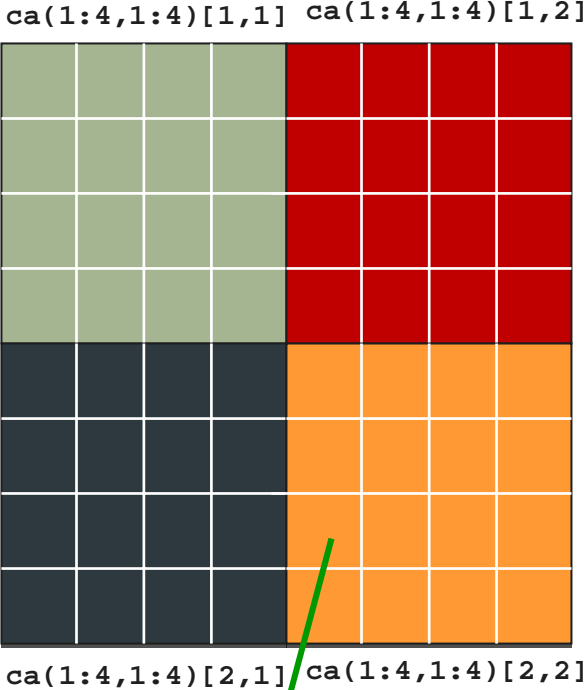
```
integer :: ca(4)[2,*]
```



2D Local Array on 2D Grid



A



global access: `ca(3,1)[2,2]`
local access: `ca(3,1)`

Coarray Subscripts

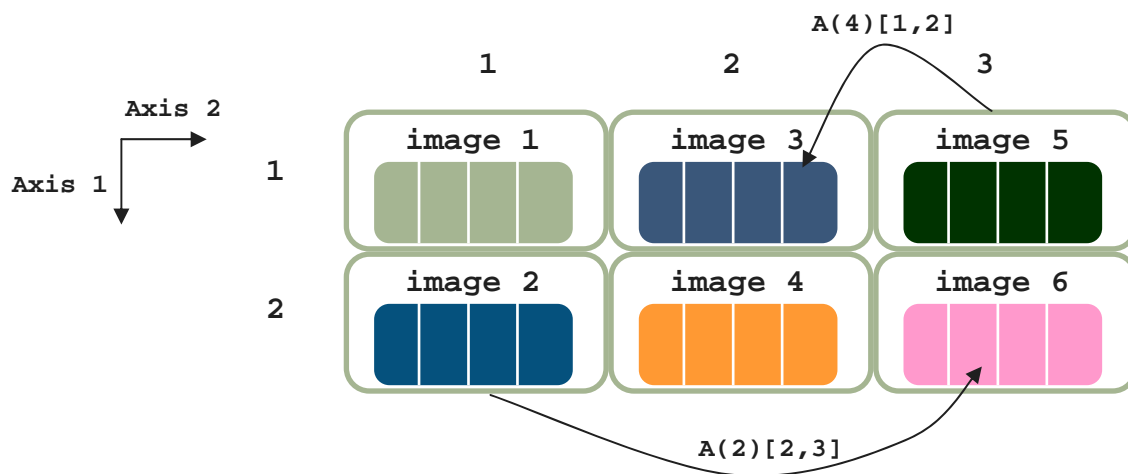
- Fortran arrays defined by rank, bounds and shape
`integer, dimension(10,4) :: array`
 - rank 2
 - lower bounds 1, 1; upper bounds 10, 4
 - shape [10, 4]
- Coarray Fortran adds corank, cobounds and coshape
`integer :: array(10,4)[3,*]`
 - corank 2
 - lower cobounds 1, 1; upper cobounds 3, m
 - coshape [3, m]
m would `ceiling(num_images() / 3)`

Multiple Codimensions

- Coarrays with multiple Codimensions:
 - `character :: a(4)[2, *] !2D grid of images`
→ for 4 images, grid is 2x2; for 16 images, grid is 2x8
 - `real :: b(8,8,8)[10,5,*] !3D grid of images`
→ 8x8x8 local array; with 150 images, grid is 10x5x3
 - `integer :: c(6,5)[0:9,0:*] !2D grid of images`
→ lower cobounds [0, 0]; upper cobounds [9,n]
→ useful if you want to interface with MPI or want C like coding
- Sum of rank and corank should not exceed 15
- Flexibility with cobounds
 - can set all but final upper cobound as required

Codimensions: What They Mean

- Images are organised into a logical 2D, 3D, grid
 - for that coarray only
- A map so an image can find the coarray on any other image
 - access the coarray using its grid coordinates
- e.g. `character a(4)[2, *]` on 6 images
 - gives a 2 x 3 image grid
 - usual Fortran subscript order to determine image index



Codimensions and Array-Element Order

- Storage order for multi-dimensional Fortran arrays

```
real p(2,3,8)
```

Location	Element
1	p(1,1,1)
2	p(2,1,1)
3	p(1,2,1)
4	p(2,2,1)
5	p(1,3,1)
6	p(2,3,1)
7	p(1,1,2)
8	p(2,1,2)
...	
48	p(2,3,8)

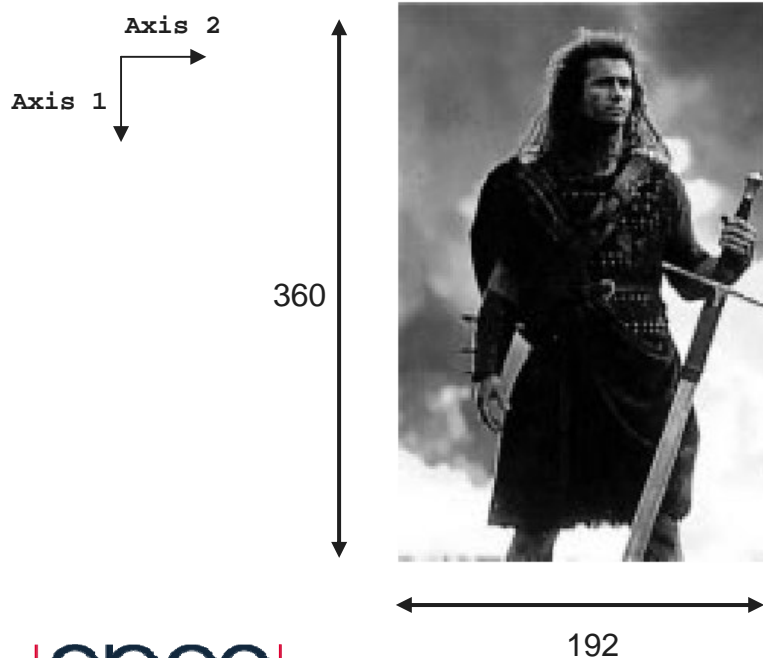
- Ordering of images in multi-dimensional cgrids

```
real q(4)[2,3,*]
```

Image	Elements
1	q(1:4)[1,1,1]
2	q(1:4)[2,1,1]
3	q(1:4)[1,2,1]
4	q(1:4)[2,2,1]
5	q(1:4)[1,3,1]
6	q(1:4)[2,3,1]
7	q(1:4)[1,1,2]
8	q(1:4)(2,1,2)
...	
48	q(1:4)[2,3,8]
...	

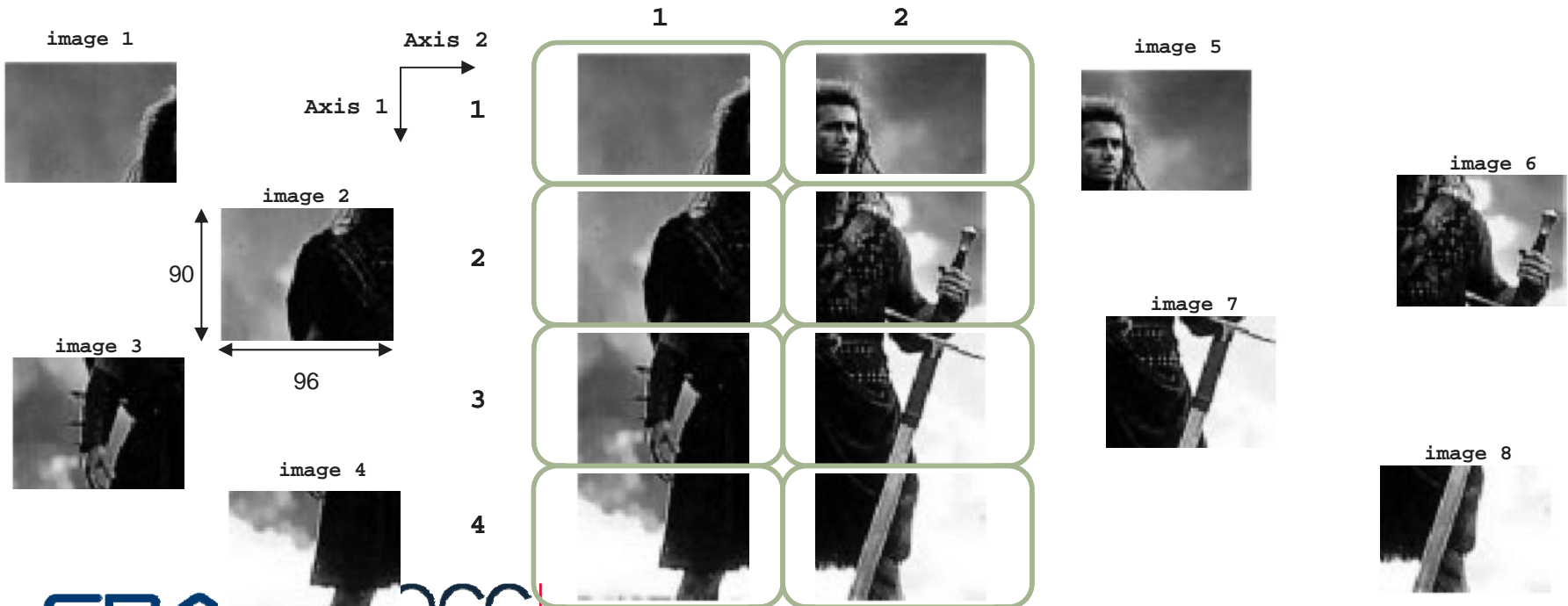
Multi Codimensions: An Example

- Domain Decomposition
 - `()` gives local domain size
 - `[]` provides image grid and easy access to other images
- 2D domain decomposition of Braveheart
- Global data is 360 x 192
- Domain decomposition on 8 images with 4 x 2 grid
 - local array size: $(360 / 4) \times (192 / 2) = 90 \times 96$
 - declaration = `real :: localMel(90,96)[4,*]`



Multi Codimensions: An Example

- Domain Decomposition
 - `()` gives local domain size
 - `[]` provides image grid and easy access to other images
- 2D domain decomposition of Braveheart
- Global data is 360 x 192
- Domain decomposition on 8 images with 4 x 2 grid
 - local array size: $(360 / 4) \times (192 / 2) = 90 \times 96$
 - declaration = `real :: localMel(90,96)[4,*]`



`this_image()` & `image_index()`

`this_image()`

- returns the image index, i.e., number between 1 and `num_images()`

`this_image(z)`

- returns the rank-one integer array of cosubscripts corresponding to the coarray `z`
- `this_image(z, dim)` returns cosubscript of codimension `dim` of `z`

`image_index(z, sub)`

- returns image index with cosubscripts `sub` for coarray `z`
- `sub` is a rank-one integer array

Example 1

```
PROGRAM CAF_Intrinsics
```

```
real :: b(90,96)[4,*]
```

```
write(*,*) "this_image() =", &  
  this_image()
```

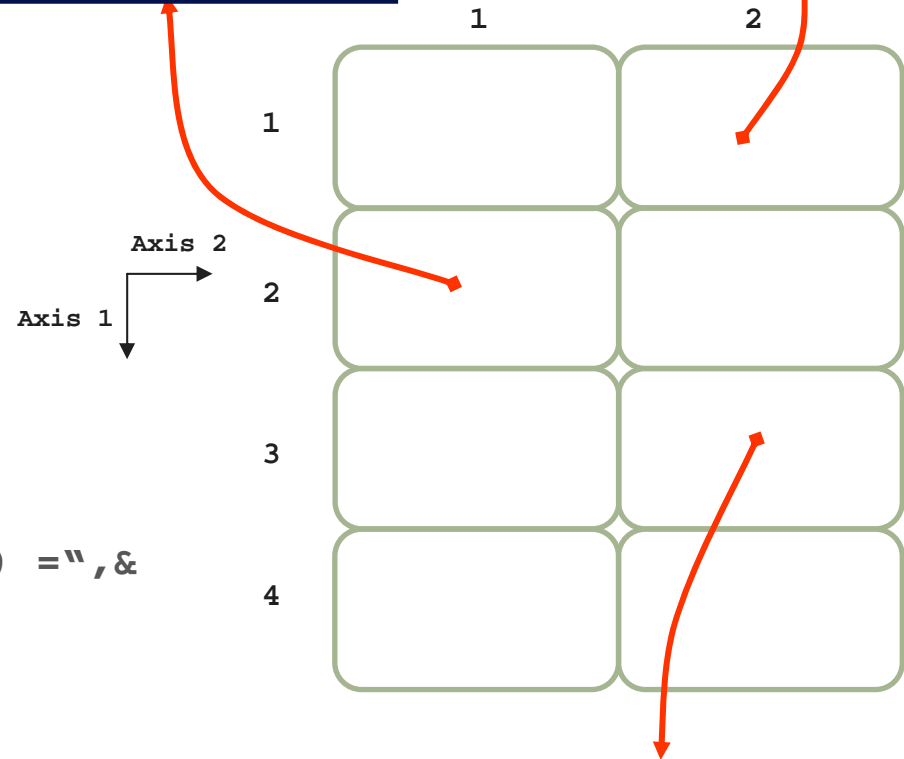
```
write(*,*) "this_image(b) =", &  
  this_image(b)
```

```
write(*,*) "image_index(b,[3,2]) =", &  
  image_index(b,[3,2])
```

```
END PROGRAM CAF_Intrinsics
```

```
this_image() = 2  
this_image(b) = [2, 1]  
image_index(b,[3,2]) = 7
```

```
this_image() = 5  
this_image(b) = [1, 2]  
image_index(b,[3,2]) = 7
```



```
this_image() = 7  
this_image(b) = [3, 2]  
image_index(b,[3,2]) = 7
```

Example 2

```
PROGRAM CAF_Intrinsics
```

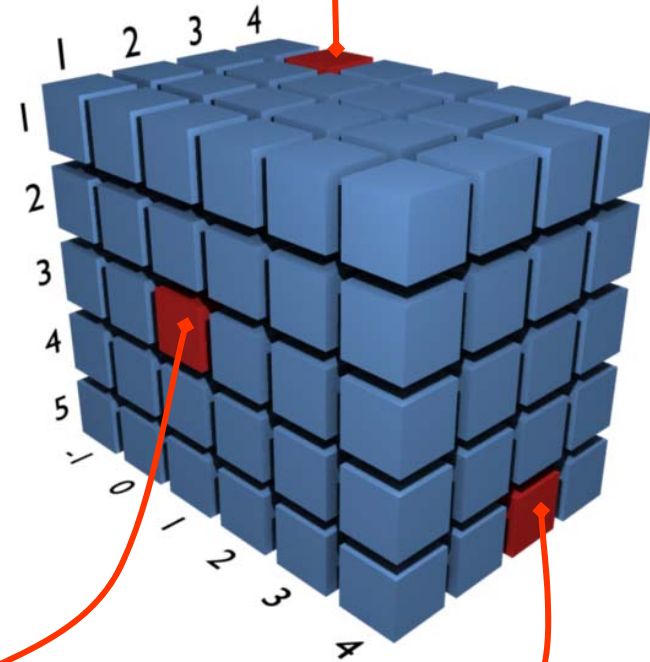
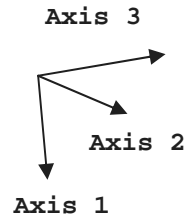
```
real :: c(4,4,4)[5,-1:4,*]
```

```
write(*,*) "this_image() =", &  
  this_image()
```

```
write(*,*) "this_image(c) =", &  
  this_image(c)
```

```
write(*,*) "image_index(c,[1,0,4]) =", &  
  image_index(c,[1,0,4])
```

```
END PROGRAM CAF_Intrinsics
```



```
this_image() = 96  
this_image(c) = [1, 0, 4]  
image_index(c,[1,0,4]) = 96
```

```
this_image() = 13  
this_image(c) = [3, 1, 1]  
image_index(c,[1,0,4]) = 96
```

```
this_image() = 90  
this_image(c) = [5, 4, 3]  
image_index(c,[1,0,4]) = 96
```

Boundary Swapping

```
PROGRAM CAF_HaloSwap
integer, parameter :: nximages = 4, nyimages = 2
integer, parameter :: nxlocal = 90, nylocal = 96
real :: pic(0:nxlocal+1, 0:nylocal+1)[nximages,*] ! Declare coarray with halos
integer :: myimage(2) ! Array for my row & column coordinates
```

```
myimage = this_image(pic) ! Find my row & column coordinates
```

Find cosubscripts

```
... ! Initialise pic on each image
```

```
sync all
```

Ensures pic initialised before accessed by other images

```
! Halo swap
```

```
if (myimage(1) > 1) &
    pic(0,1:nylocal) = pic(nxlocal,1:nylocal)[myimage(1)-1,myimage(2)]
```

```
if (myimage(1) < nximages) &
    pic(nxlocal+1,1:nylocal) = pic(1,1:nylocal)[myimage(1)+1,myimage(2)]
```

```
if (myimage(2) > 1) &
    pic(1:nxlocal,0) = pic(1:nxlocal,nylocal)[myimage(1),myimage(2)-1]
```

```
if (myimage(2) < nyimages)
    pic(1:nxlocal,nylocal+1) = pic(1:nxlocal,1)[myimage(1),myimage(2)+1]
```

```
sync all
```

Ensures all images have got old values before pic is updated

```
... ! Update pic on each image
```

```
END PROGRAM CAF_HaloSwap
```

Allocatable Coarrays

- Can have allocatable Coarrays

```
real, allocatable :: x(:)[:], s[:,:]  
n = num_images()  
allocate(x(n)[*], s[4,*])
```

- Must specify cobounds in **allocate** statement
- The size & value of each bound and cobound must be same on all images.
 - **allocate(x(this_image())[*])** ! Not allowed
- Implicit synchronisation of all images...
 - ...after each **allocate** statement involving coarrays
 - ...before **deallocate** statements involving coarrays

Differently Sized Coarray Components

- A coarray structure component can vary in size per image
- Declare a coarray of derived type with a component that is allocatable (or pointer)...

```
!Define data type with allocatable component
type diffSize
    real, allocatable :: data(:)
end type diffSize
!Declare coarray of type diffSize
type(diffSize) :: x[*]

! Allocate x%data to a different size on each image
allocate(x%data(this_image()))
```

Pointer Coarray Structure Components

- Pointer components are also useful to provide access to non-coarray data on an image
- Pointer points to data *local* to an image

```
subroutine calc(u,v,w)
  real, intent(in), target, dimension(100) :: u,v,w
  type coords
    real, pointer, dimension(:) :: x,y,z
  end type coords
  type(coords), save :: vects[*]
  ! ...
  vects%x => u ; vects%y => v ; vects%z => w
  sync all
  firstx = vects[1]%x(1)
```

Coarrays and Procedures

- An explicit interface is required if a dummy argument is a coarray
- Dummy argument associated with coarray, not a copy
 - avoids synchronisation on entry and return
- Other restrictions on passing coarrays are:
 - the actual argument should be contiguous
 - `a(:, 2)` is OK, but `a(2, :)` is not contiguous
 - or the dummy argument should be assumed shape
... to avoid copying
- Function results cannot be coarrays

Coarrays as Dummy Arguments

- As with standard Fortran arrays, the coarray dummy arguments in procedures can be:
 - Explicit shape: each dimension of a coarray declared with explicit value
 - Assumed shape: extents and bounds determined by actual array
 - Assumed size: only size determined from actual array
 - Allocatable: the size and shape can be determined at run-time

```
subroutine s(n, a, b, c, d)
integer :: n
real :: a(n) [n,*] ! explicit shape - permitted
real :: b(:,*) [*] ! assumed shape - permitted
real :: c(n,*) [*] ! assumed size - permitted
real, allocatable :: d(:) [,:,:] ! allocatable - permitted
```

Assumed Size Coarrays

- Allow the coshape to be remapped to corank 1

```
program cmax
real, codimension[8,*] :: a(100), amax

    a = [ (i, i=1,100) ] * this_image() / 100.0
    amax = maxval( a )
    sync all
    amax = AllReduce_max(amax)
...
contains
    real function AllReduce_max(r) result(rmax)
    real :: r[*]

    rmax = r
    do i=1,num_images()
        rmax = max( rmax, r[i] )
    end do
end function AllReduce_max
```

Coarrays Local to a Procedure

- Coarrays declared in procedures must have **save** attribute
 - unless they are dummy arguments or allocatable
 - save attribute: retains value between procedure calls
 - avoids synchronisation on entry and return
- Automatic coarrays are not permitted
 - Automatic array: local array whose size depends on dummy arguments
 - would require synchronisation for memory allocation and deallocation
 - would need to ensure coarrays have same size on all images

```
subroutine t(n)
integer :: n
real :: temp(n)[*] ! automatic - not permitted
integer, save :: x(4)[*] ! coarray with save attribute
integer :: y(4)[*] ! not saved - not permitted
```

Summary

- Coarrays with multiple codimensions used to create a grid of images
 - `()` gives local domain information
 - `[]` gives an image grid with easy access to other images
- Can be used in various ways to assemble a multi-dimensional data set
- **`this_image()`** and **`image_index()`**
 - are intrinsic functions that give information about the images in an multi-codimension grid
- Flexibility from non-coarray allocatable and pointer components of coarray structures
- Coarrays can be allocatable, can be passed as arguments to procedures, and can be dummy arguments