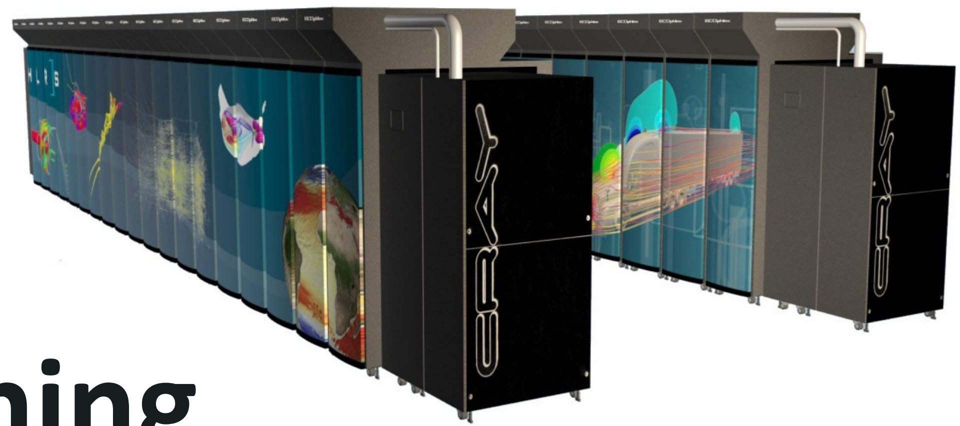


PGAS programming with Fortran coarrays

Jason J Beech-Brandt
Cray UK

CRAY
THE SUPERCOMPUTER COMPANY

H L R S

Overview

- PGAS in context with other programming models
- Fortran coarray features

Programming models and PGAS

- Parallel programming models allow us to build applications that can run efficiently on parallel architectures
- PGAS stands for Partitioned Global Address Space and is one of the programming models used in parallel programming
- We will introduce the PGAS approach in context with other traditional programming models
  - shared memory directives
  - message-passing

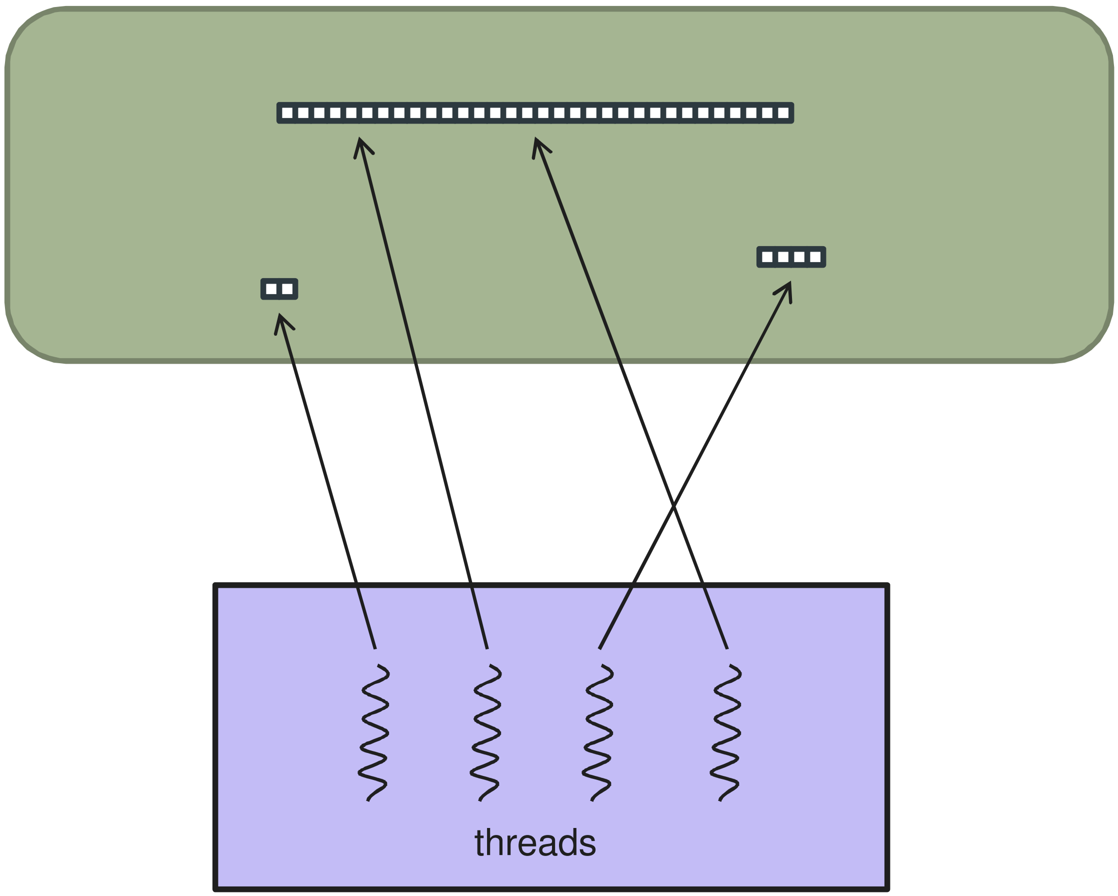
Shared Memory Directives

- Multiple threads share global memory
- Most common variant: OpenMP
- Program loop iterations distributed to threads, more recent task features
  - Each thread has a means to refer to private objects within a parallel context
- Terminology
  - Thread, thread team
- Implementation
  - Threads map to user threads running on one SMP node
  - Extensions to multiple servers not so successful

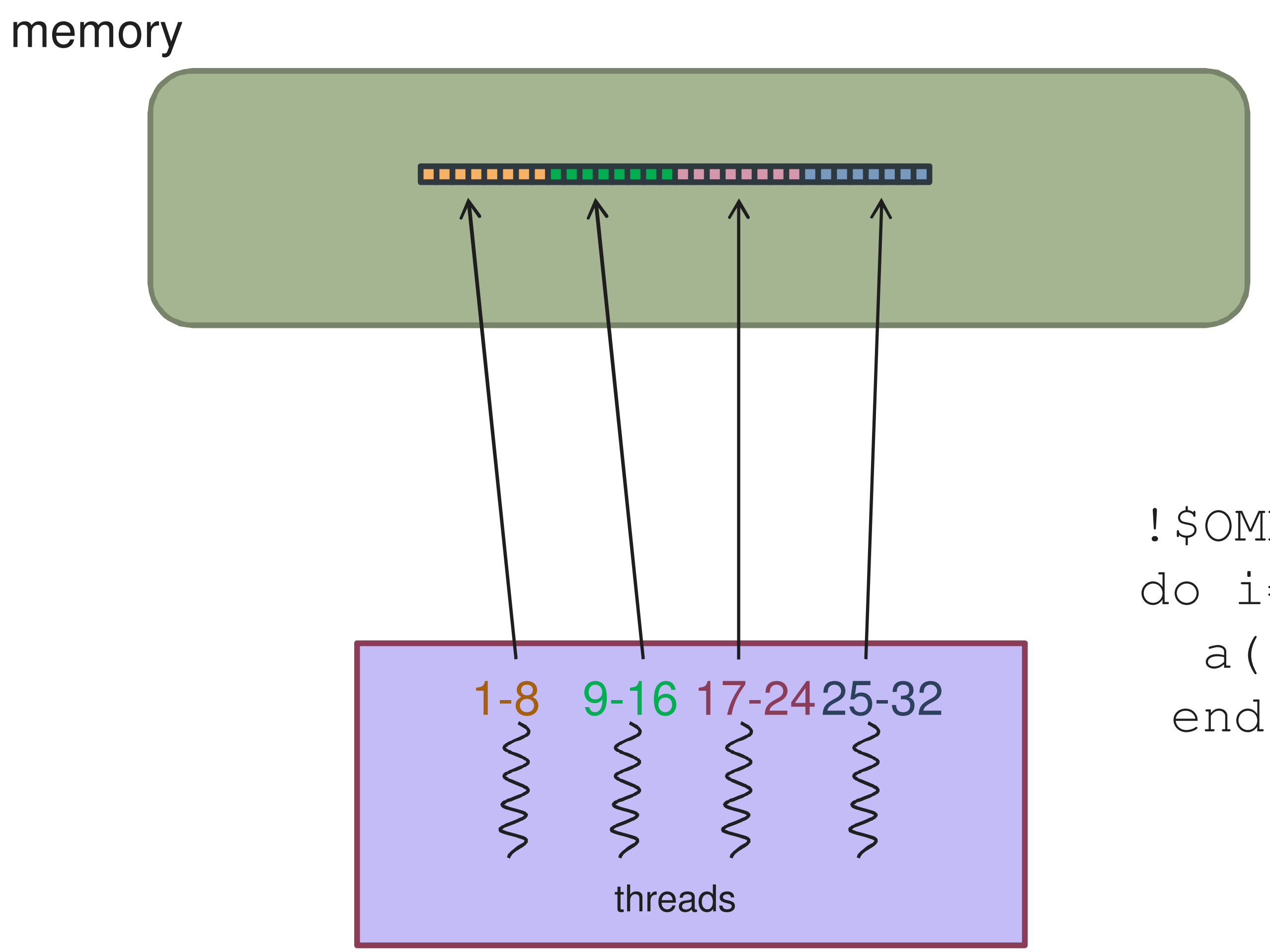
CRAY THE SUPERCOMPUTER COMPANY

H L R I S

OpenMP

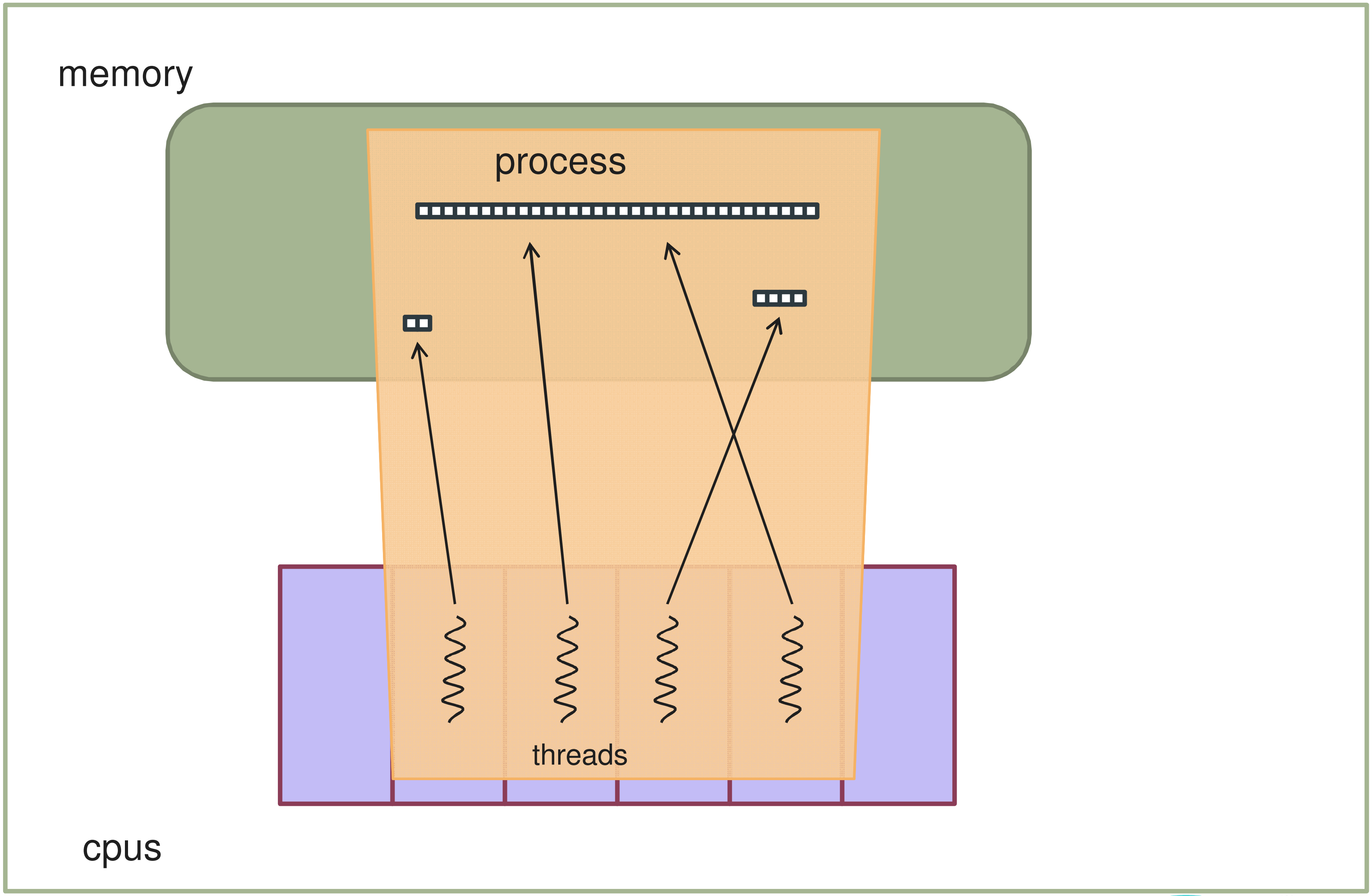


OpenMP: work distribution




```
!$OMP PARALLEL
do i=1,32
  a(i)=a(i)*2
 end do
```

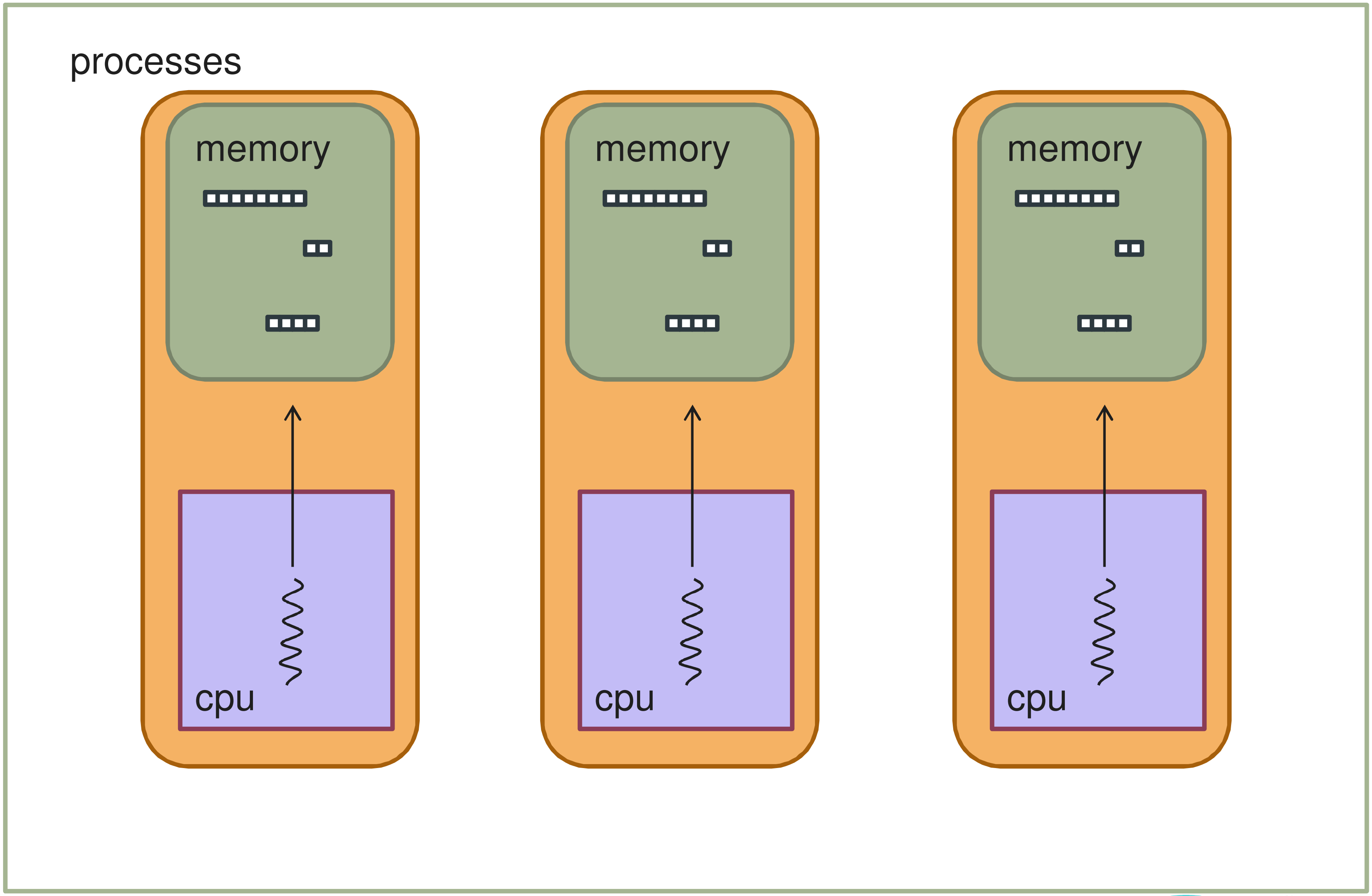
OpenMP implementation



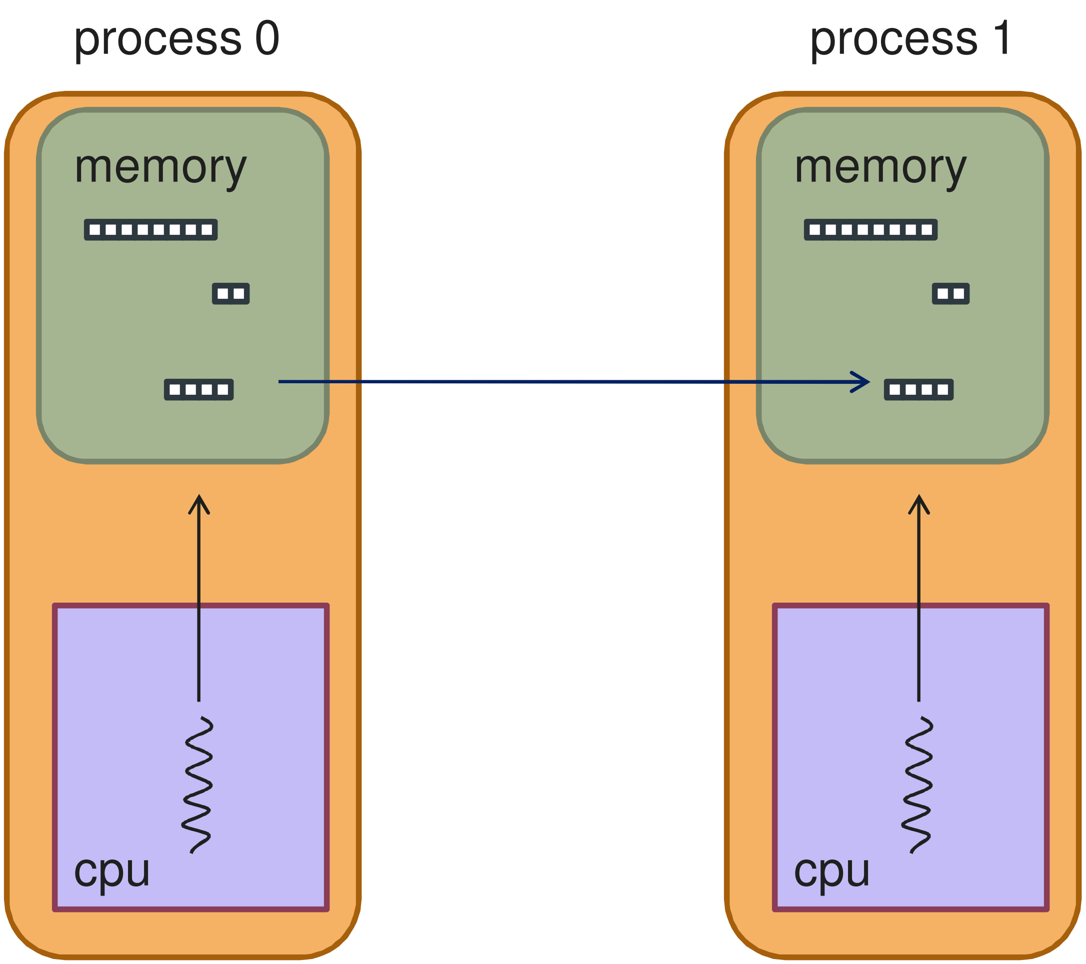
Message Passing

- Participating processes communicate using a message-passing API
- Remote data can only be communicated (sent or received) via the API
- MPI (the Message Passing Interface) is the standard
- Implementation:
  MPI processes map to processes within one SMP node or across multiple networked nodes
- API provides process numbering, point-to-point and collective messaging operations

MPI



MPI

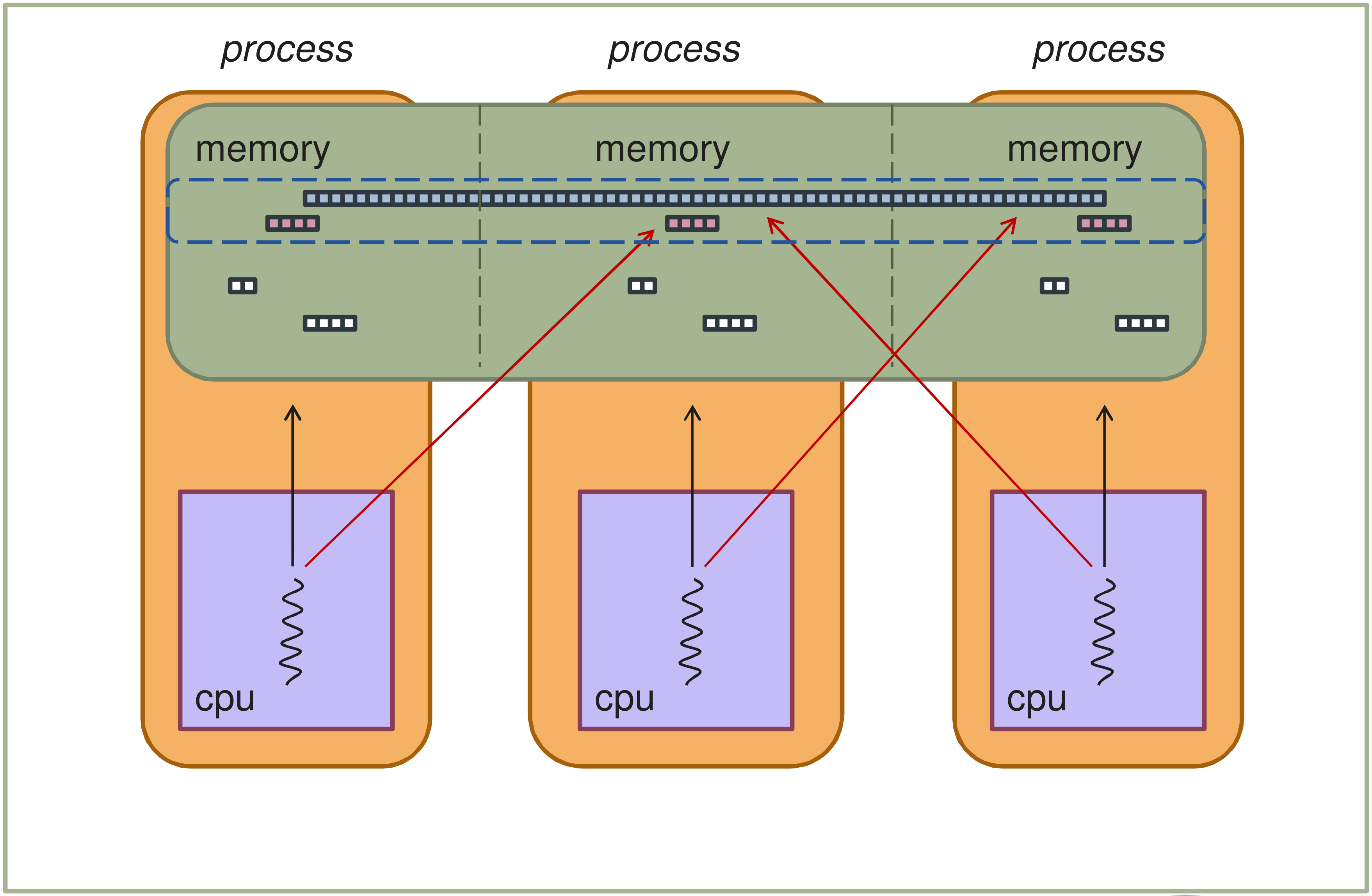


`MPI_Send(a,...,1,…)` `MPI_Recv(a,...,0,…)`

Partitioned Global Address Space Model

- Shortened to PGAS
- Participating processes/threads have access to **local memory** via standard program mechanisms
- Access to **remote memory** is directly supported by the PGAS language

PGAS

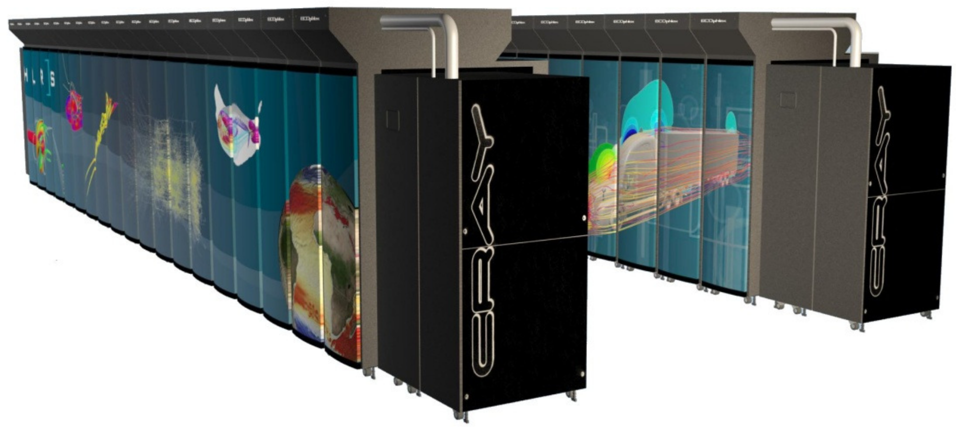


PGAS Languages

- Various Implementations including Fortran coarrays, UPC and Titanium
- Coarrays (Fortran)
  - Participating images
  - New codimension attribute for objects
  - New mechanism for remote access:
    `a(:)=b(:)[image]` ! Get b from remote image
- UPC
  - Participating “*threads*”
  - New *shared* data structures
  - Language constructs to divide up work on shared data

PGAS Advantages

- Remote access is a full feature of the language:
  - Type checking
  - Opportunity to optimize communication
- No performance penalty for local memory access
- Single-sided programming model more natural for some algorithms
  - and a good match for modern networks with RDMA



Fortran coarrays

An introduction

Coarrays

"Coarrays were designed to answer the question:

'What is the smallest change required to convert Fortran into a robust and efficient parallel language?'

The answer: a simple syntactic extension.

It looks and feels like Fortran and requires Fortran programmers to learn only a few new rules."

John Reid,
ISO Fortran Convener

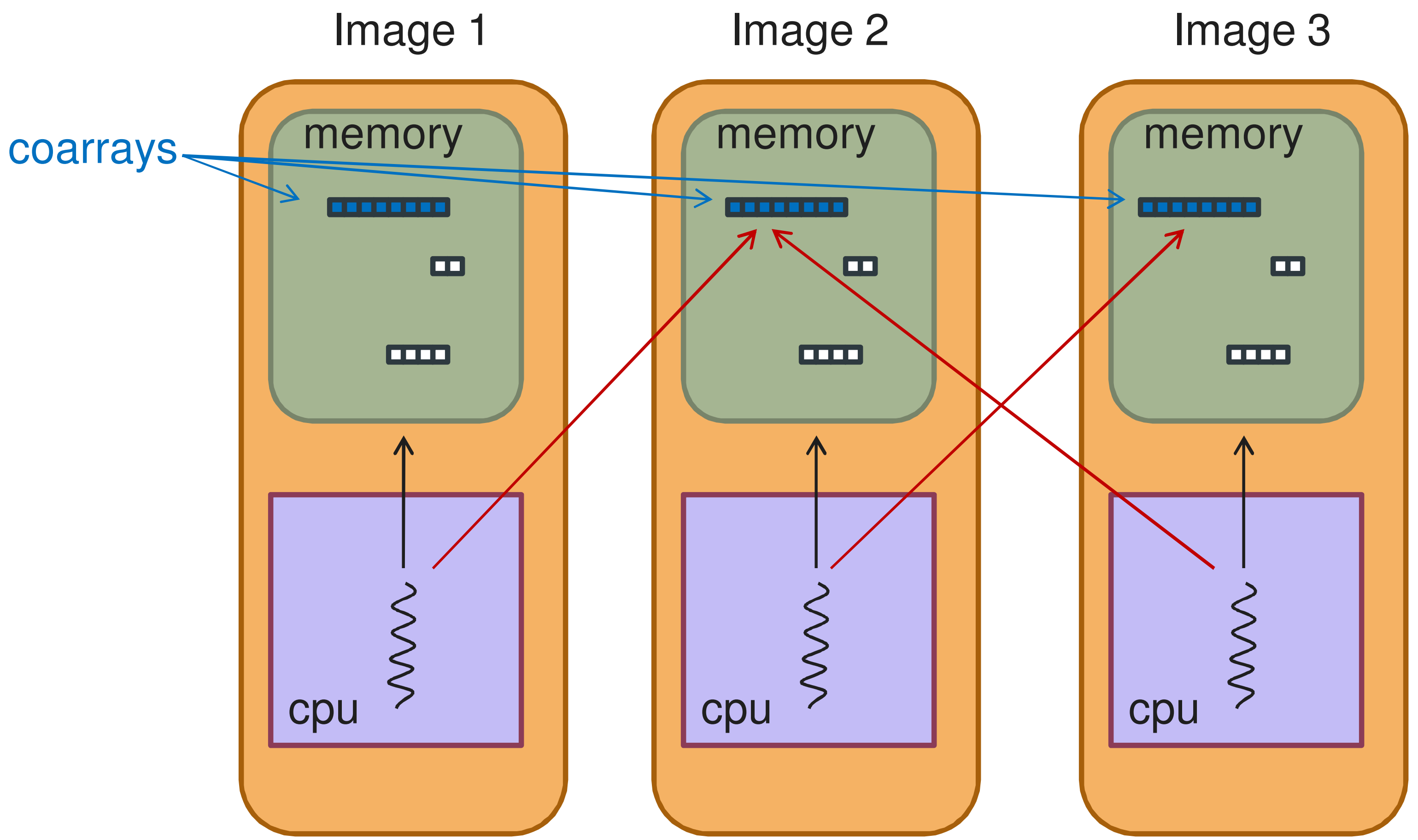
Coarrays in Fortran

- Introduced in current form by Numrich and Reid in 1998 as a simple extension to Fortran 95 for parallel processing
- Implemented on various Cray hardware platforms
- A set of core features are now part of the Fortran standard: ISO/IEC 1539-1:2010
- Additional features are expected to be published in a Technical Report in due course.
- Various vendor and GNU projects (Intel, g95, gfortran) underway

Basic execution model and features

- Program executes as if replicated to multiple copies with each copy executing asynchronously (SPMD)
- Each copy (called an **image**) executes as a normal Fortran application
- New object indexing with [] can be used to access objects on other images.
- New features to inquire about image index, number of images and to synchronize

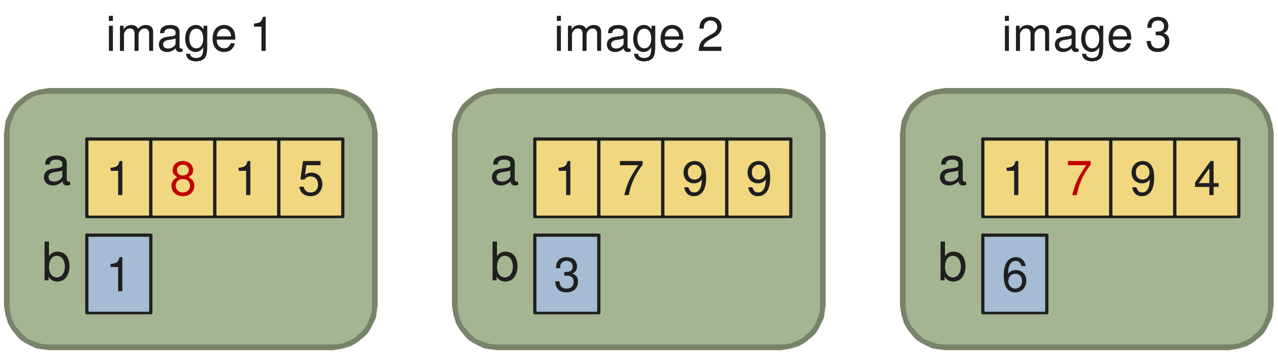
Coarray execution model



Remote access with square bracket indexing: a(:)[2]

Basic coarray declaration and usage

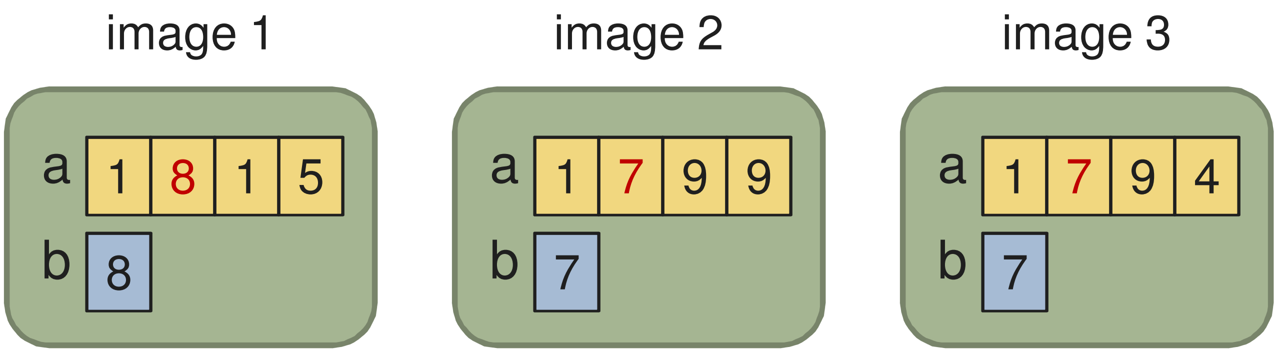
```
integer :: b
integer :: a(4)[*] !coarray
```



- Coarray has to be the same size on each image

Basic coarray declaration and usage

```
integer :: b
integer :: a(4)[*] !coarray
```

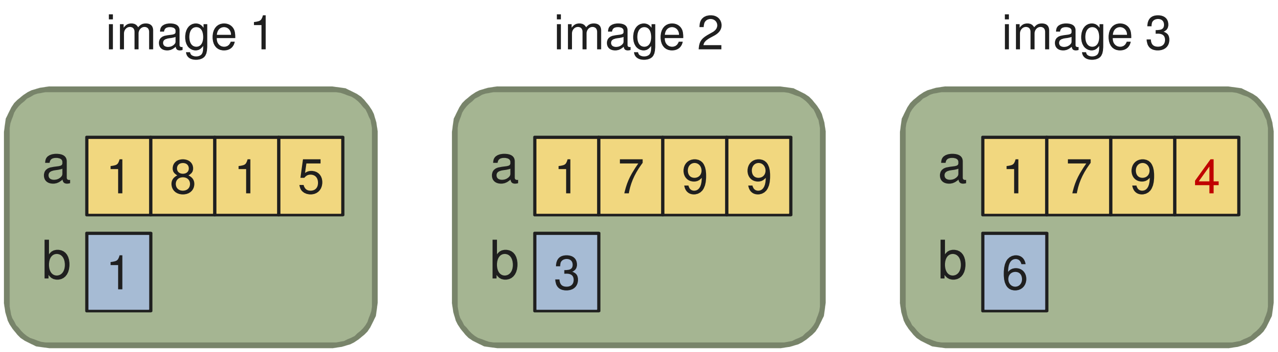



```
b=a(2)
```

- References without [] are local
- **b** is set to second element of **a** on each image

Basic coarray declaration and usage

```
integer :: b
integer :: a(4)[*] !coarray
```

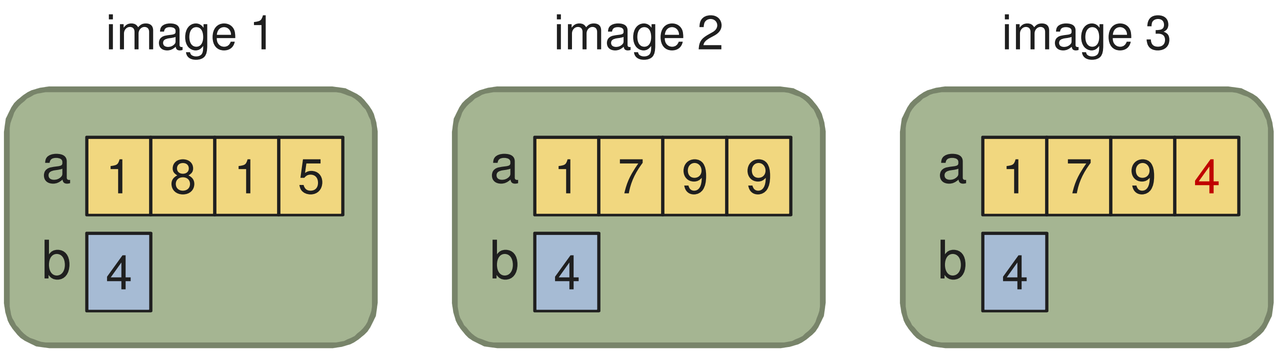



```
b=a(4)[3]
```

- [] indicates access to remote coarray data
- Each `b` is set to fourth element of array `a` on image 3

Basic coarray declaration and usage

```
integer :: b
integer :: a(4)[*] !coarray
```




```
b=a(4)[3]
```

- [] indicates access to remote coarray data
- Each **b** is set to fourth element of array **a** on image 3

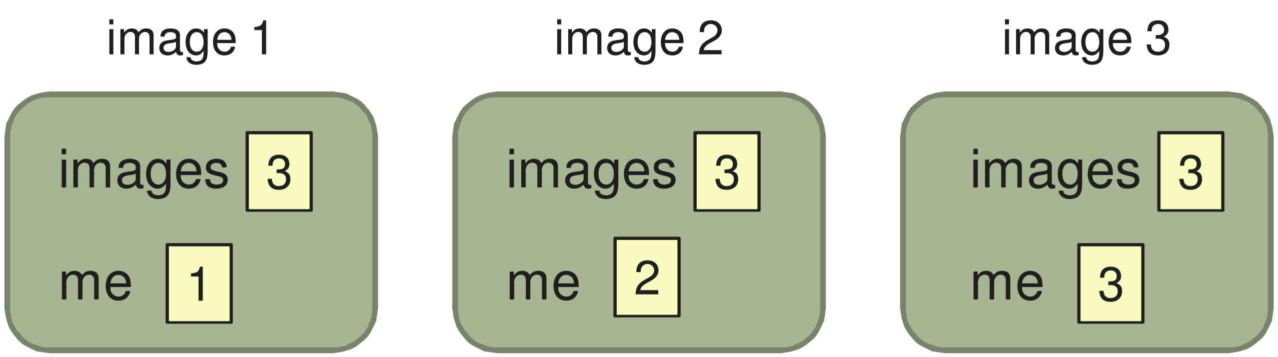
More Coarray Declarations

```
real :: residual[*]   ! Scalar coarray
real, dimension(100), codimension[*] :: x,y
integer, dimension(m) :: offsets[0:*]

type (color) map(512,512)[*]

character(len=80), allocatable ::search_space(:)[:]

allocate( search_space(2000)[*] )
```

Image execution

Functions provided to return number of images and index of executing image

```
images = num_images()
me = this_image()
```



- Used to allow images to organise problem distribution and to operate independently

Example: Read array from file

```
double precision, dimension(n) :: a
double precision, dimension(n) :: temp[*]
!...
if (this_image() == 1) then
  do i=1, num_images()
    read *,a
    temp(:)[i] = a
   end do
 end if

temp = temp + 273d0  !!! THIS IS NOT SAFE
```

- Read **n** elements at a time and distribute

Basic Synchronization: sync all

```
!...
if (this_image() == 1) then
  do i=1, num_images()
    read *,a
    temp(:)[i] = a
   end do
 end if

sync all
temp = temp + 273d0
```

- images only continue when all images have reached the statement and remote references are completed

Recap of coarray basics

- multiple images execute asynchronously
- we can declare a coarray which is accessible from multiple images
- indexing with [] is used to access remote data
- we can find out which image we are
  - `num_images()`
  - `this_image()`
- we can synchronize to make sure variables are up to date
  - `sync all`

Now consider a program example...

Example2: Calculate density of primes

```
program pdensity
 implicit none
 integer, parameter :: n=8000000, nimages=8
 integer start,end,i
 integer, dimension(nimages) :: nprimes[*]
 real density

 start = (this_image()-1) * n/num_images() + 1
 end = start + n/num_images() - 1

 nprimes(this_image())[1] = num_primes(start,end)

 sync all
```

Example2: Calculate density of primes

```
…
if (this_image()==1) then

  nprimes(1)=sum(nprimes)
  density=real(nprimes(1))/n
  print *,"Calculating prime density on", &
&         num_images(),"images"
  print *,nprimes(1),'primes in',n,'numbers'
  write(*,'(" density is ",2P,f5.2,"%")')density
  write(*,'(" asymptotic theory gives ", &
&           2P,f5.2,"%")')1.0/(log(real(n))-1.0)

end if
```

Example2: Calculate density of primes

- `Calculating prime density on 2 images`
- `539778 primes in 8000000 numbers`
- `density is  6.75%`
- `asymptotic theory gives  6.71%`

Program Launch

- The Fortran standard does not specify how a program is launched
- The number of images may be set at compile, link or run-time
- A compiler could optimize for a single image
- Examples on Linux
  - Cray XE
    aprun –n 16000 solver
  - g95
    ./solver --g95 -images=2

Multi-codimensional coarrays

- More general declarations

```
complex :: b[0:*]
complex :: p(32,32)[2,3,*]
```

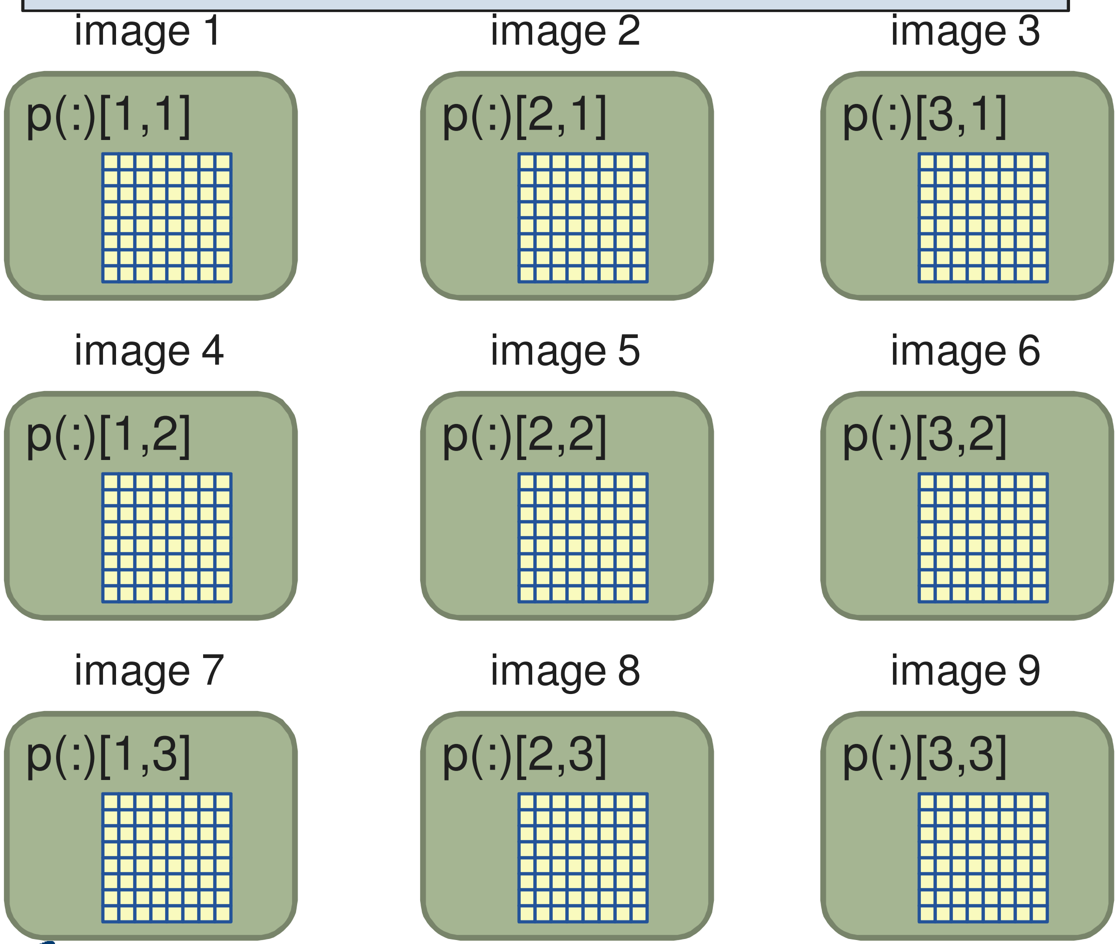
- Cosubscripts map to images in array-element order

image	b(:)[i]	p(:)[i,j,k]
1	b(:)[0]	p(:)[1,1,1]
2	b(:)[1]	p(:)[2,1,1]
3	b(:)[2]	p(:)[1,2,1]
4	b(:)[3]	p(:)[2,2,1]
5	b(:)[4]	p(:)[1,3,1]
6	b(:)[5]	p(:)[2,3,1]
7	b(:)[6]	p(:)[1,1,2]

Multi-codimensional coarrays…

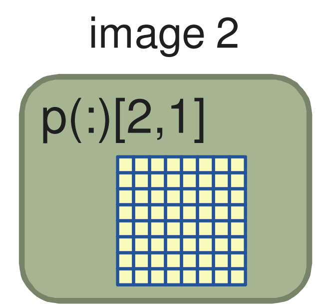
- Example: pixel data distributed on 9 images

```
type(pixel) :: p(8,8)[3,*]
```



Multi-codimensional coarrays…

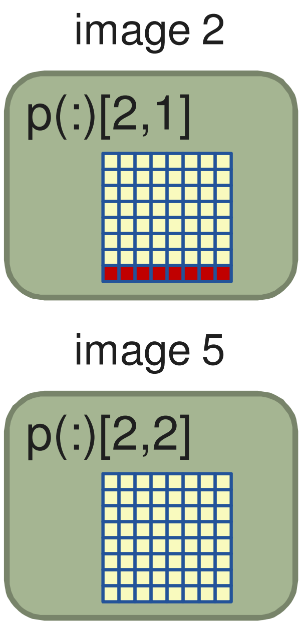
- There is a way to find out which part of the coarray is mapped to an image
  - `this_image(coarray)` yields codimensions
  - `this_image(coarray,dim)` yields one codimension
- So for the previous example, on image 2
  - `this_image(p)` is `[ 2, 1 ]`
- Can get image index from coarray:
  - `image_index(p,[2,1])` is 2
  - `image_index(p,[3,4])` is 0



Multi-codimensional coarrays…

- Example: copy the bottom row from the image 'above' me in the grid…

```
type(pixel) :: p(32,32)[3,*],&
&              copy(32)

px = this_image(p,1)
py = this_image(p,2)

copy = p(:,32)[px, py-1]
```



Allocatable coarrays

```
integer n,ni
real, allocatable :: pmax(:)[:]
real, allocatable :: p(:,:)[:,:]
!...

ni = num_images()
allocate( pmax(ni)[*], p(n,n)[4,*] )
```

- Require same shape and coshape on every image
- allocate and deallocate with coarray arguments cause a synchronization

Allocatable components of coarrays

Can have allocatable or pointer components of derived types

```
type treetype
  type(nodetype), allocatable :: node(:)
end type treetype

type(treetype) :: tree[*]

allocate( tree%node(nnodes) )
```

- The size can vary on each image

pointer components of coarrays

```
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) :: vects[*]
! …
vects%x => u ; vects%y => v ; vects%z => w
sync all

firstx = vects[1]%x(1)
```

- Pointers can point to non-coarray data
- Useful technique for adding coarray features into existing applications

Coarrays and procedures

- Can pass coarrays to procedures
  - to explicit, asuumed-shape, assumed-size or allocatable dummy arguments
- There must be an explicit interface for the call
- Actual argument can be a contiguous section of a coarray
- Function result can not be a coarray
- Rules designed to avoid copy and synchronization
- automatic coarrays are not allowed

  (local arrays sized based on dummy arguments)

coarrays and procedures…

```
subroutine bill(a, b, c, t, n)
 integer :: n
 real :: a(n,n) [n,*] ! explicit shape
 real :: b(:)    [*]    ! assumed shape
 real :: c(n,*) [*]    ! assumed size

 real, allocatable :: t(:,:) [:] ! allocatable
 real, save :: bill_totals(8)[*] ! saved coarray

! complex :: q(n)[*] ! automatic – not allowed
```

- All coarrays have to be dummy arguments, saved or allocatable
- A variable that is saved maintains its state on exit

coarrays and procedures…

We can remap the codimension to rank 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
! …
```

More on Synchronization

- can synchronize on a subset of images
- Example: accumulate partial sums along images

```
real :: a(n), psum[*]

me = this_image()
psum = sum(a)
if (me > 1) then
  sync images(me-1)
  psum = psum + psum[me-1]
end if
if (me < num_images()) sync_images(me+1)
```

- Argument to **`sync images`** can be rank-1 array or *

More on Synchronization

We have to be careful with one-sided updates

- If we get remote data was it valid?
- Could another process send us data and overwrite something we have not yet used?
- How do we know when remote data has arrived?

- The standard introduces **execution segments** to deal with this, segments are bounded by image control
- 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

Execution Segments

segment

```
double precision :: a(n)
double precision :: temp(n)[*]
!...
if (this_image() == 1) then
  do i=1, num_images()
    read *,a
    temp(:)[i] = a
  end do
 end if

sync all
temp = temp + 273d0
```

```
double precision :: a(n)
double precision :: temp(n)[*]
!...
if (this_image() == 1) then
  do i=1, num_images()
    read *,a
    temp(:)[i] = a
  end do
 end if

sync all
temp = temp + 273d0
```

image synchronization points

I/O

- Each image has its own set of input/output units
- units are independent on each image
- Default input unit is preconnected on image 1 only
  - `read *,…  ,read(*,…)…`
- Default output unit is avaiable on all images
  - `print *,…  ,write(*,…)…`
  - It is expected that the implementation will merge records from each image into one stream

Program Termination

- STOP or END PROGRAM statements initiate *normal termination* which includes a synchronisation step
- An image's data is still available after it has initiated normal termination
- Other images can test for this using STAT= specifier to synchronisation calls or allocate/deallocate
  - test for STAT_STOPPED_IMAGE (defined in ISO_FORTRAN_ENV module)
- The ERROR STOP statement initiates error termination and it is expected all images will be terminated.

CRAY THE SUPERCOMPUTER COMPANY HLRS

Other features we will not cover

- Memory synchronization (`sync memory`)
  - completion of remote operations but not segment ordering
- critical section (`critical`,…,`end critical`)
  - only one image executes the section at a time
- locks
  - control access to data held by one image
- status and error conditions for image control
- atomic subroutines (useful for flag variables)

CRAY THE SUPERCOMPUTER COMPANY

H L R S

Future for coarrays in Fortran

- Additional coarray features may be described in a Technical Report (TR)
- Work in progress and the areas of discussion are:
  - **image teams**
  - **collective intrinsics for coarrays**
  - file operations by more than one image
  - new atomics
  - coarray pointers and non-symmetric allocation
  - coscalars

Implementation Status

- History of coarrays dates back to Cray implementations
- Expect support from vendors as part of Fortran 2008
- G95 had multi-image support in 2010
- gfortran
  - work progressing (4.6 trunk) for single-image support
- Intel: multi-process coarray support in Intel Composer XE 2011 (based on Fortran 2008 draft)
- Runtimes are SMP, GASNet and compiler/vendor runtimes
  - GASNet has support for multiple environments (IB, Myrinet, MPI, UDP and Cray/IBM systems) so could be an option for new implementations

Implementation Status (Cray)

- Cray has supported coarrays and UPC on various architectures over the last decade (from T3E)
- Full PGAS support on the Cray XT/XE
  - Cray Compiling Environment 7.0 – Dec 2008
  - Cray Compiler Environment 7.3 – Dec 2010
  - Full Fortran 2008 coarray support
  - Full Fortran 2003 with some Fortran 2008 features
- Fully integrated with the Cray software stack
  - Same compiler drivers, job launch tools, libraries
  - Integrated with Craypat – Cray performance tools
  - Can mix MPI and coarrays

References

- ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1824.pdf
  "Coarrays in the next Fortran Standard", *John Reid, April 2010*
- http://lacsi.rice.edu/software/caf/downloads/documentation/nrRAL98060.pdf- Co-array Fortran for parallel programming, Numrich and Reid, 1998
- ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1824.pdf
  "Coarrays in the next Fortran Standard", John Reid, April 2010
- 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

THE SUPERCOMPUTER COMPANY