

Parallel Programming with Coarray Fortran

PRACE Autumn School, October 29th 2010

David Henty, Alan Simpson (EPCC)

Harvey Richardson, Bill Long, Nathan Wichmann (Cray)

Tutorial Overview

- The Fortran Programming Model
- Basic coarray features
- Practical Session 1
- Further coarray features
- Practical Session 2
- Advanced coarray features
- Practical Session 3
- Experiences with coarrays

The Fortran Programming Model

Motivation

- Fortran now supports parallelism as a full first-class feature of the language
- Changes are minimal
- Performance is maintained
- Flexibility in expressing communication patterns

Programming models for HPC

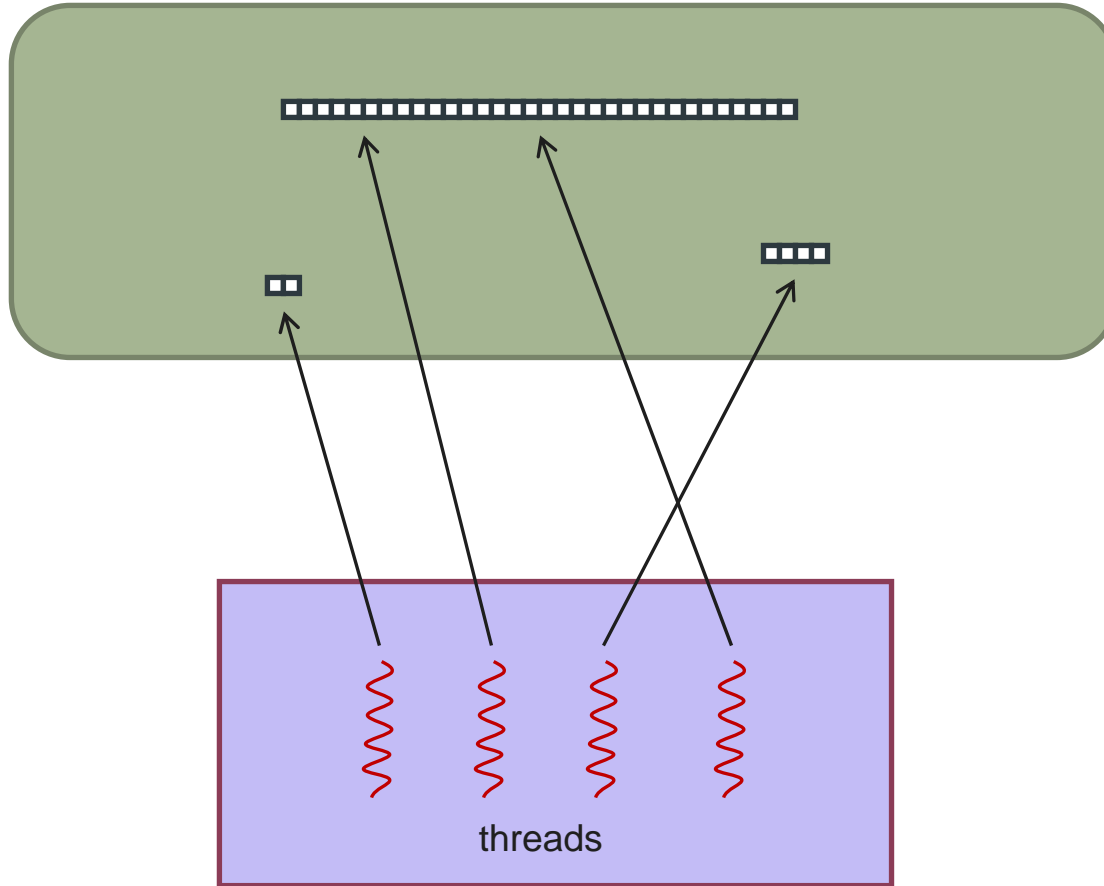
- The challenge is to efficiently map a problem to the architecture we have
 - Take advantage of all computational resources
 - Manage distributed memories etc.
 - Optimal use of any communication networks
- The HPC industry has long experience in parallel programming
 - Vector, threading, data-parallel, message-passing etc.
- We would like to have models or combinations that are
 - efficient
 - safe
 - easy to learn and use

Why consider new programming models?

- Next-generation architectures bring new challenges:
 - Very large numbers of processors with many cores
 - Complex memory hierarchy
 - even today (2010) we are at 200k cores
- Parallel programming is hard, need to make this simpler
- Some of the models we currently use are
 - bolt-ons to existing languages as APIs or directives
 - Hard to program for underlying architecture
 - unable to scale due to overheads
- So, is there an alternative to the models prevalent today?
 - Most popular are OpenMP and MPI ...

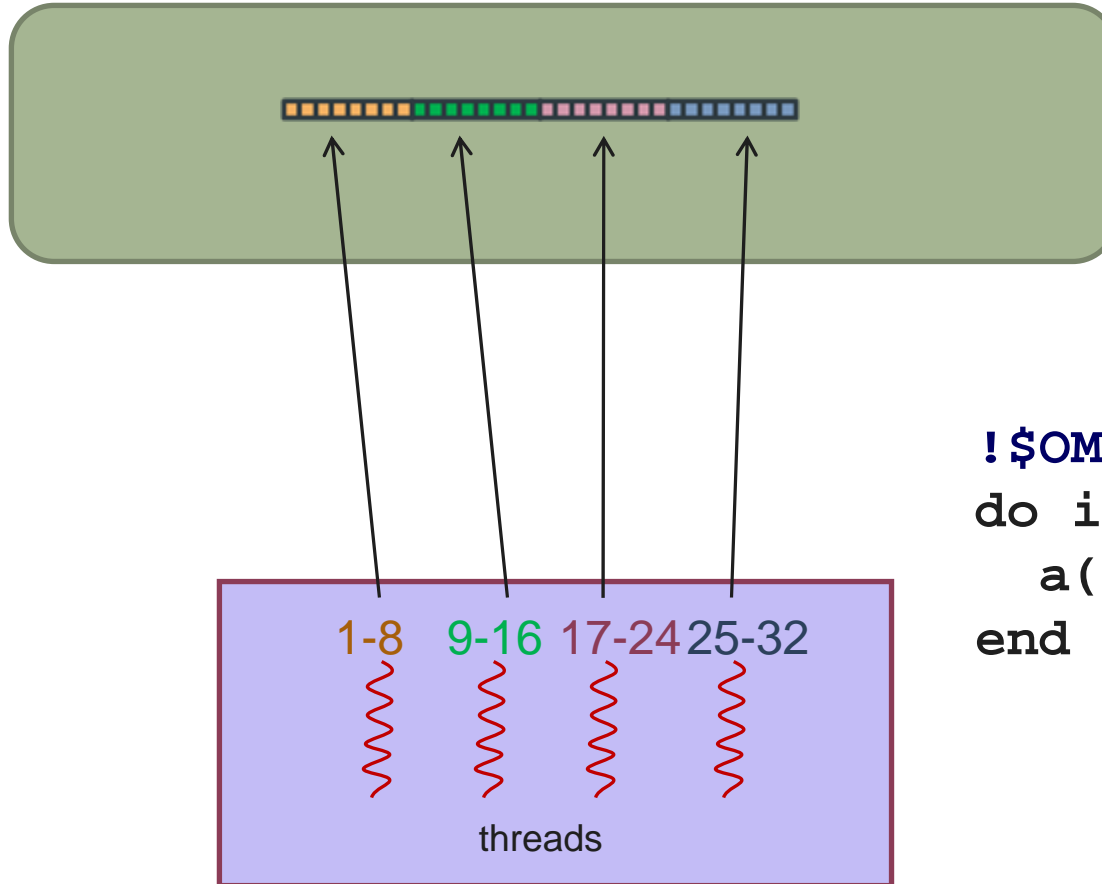
Shared-memory directives and OpenMP

memory



OpenMP: work distribution

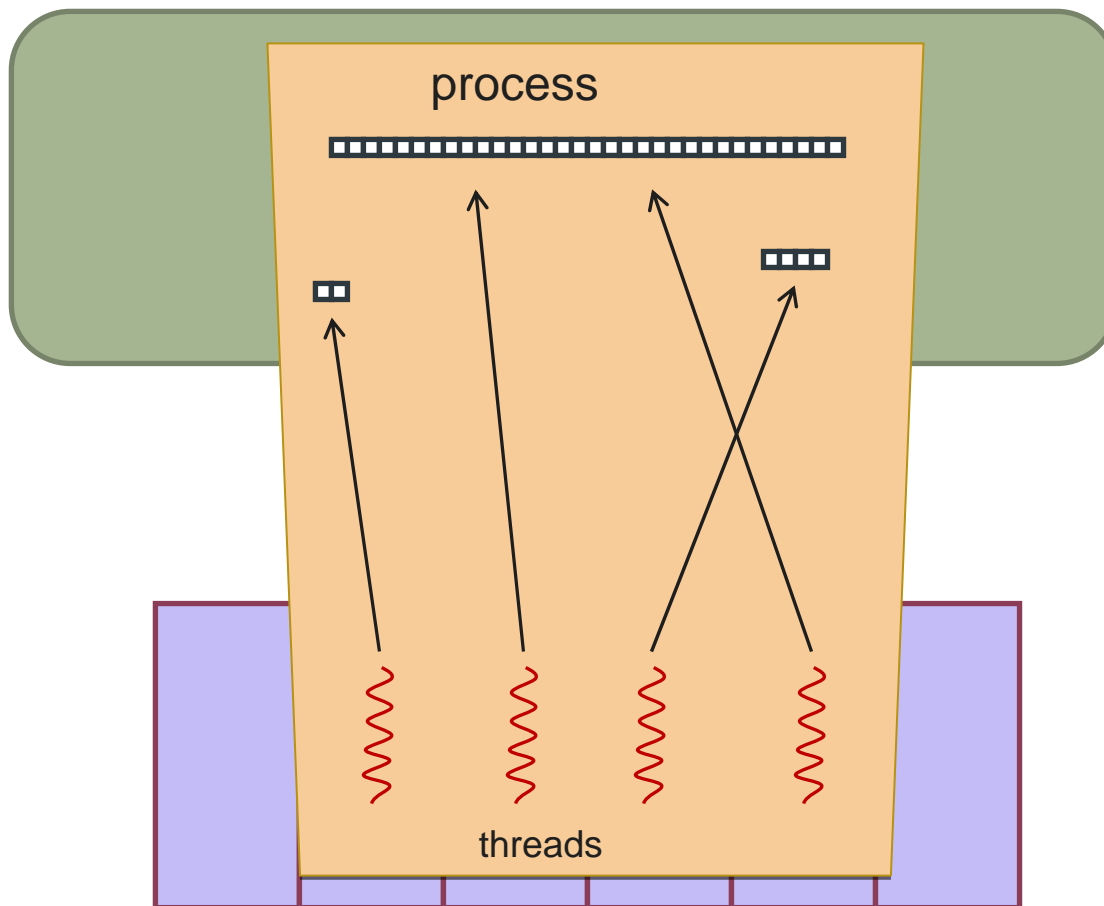
memory



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


OpenMP implementation

memory

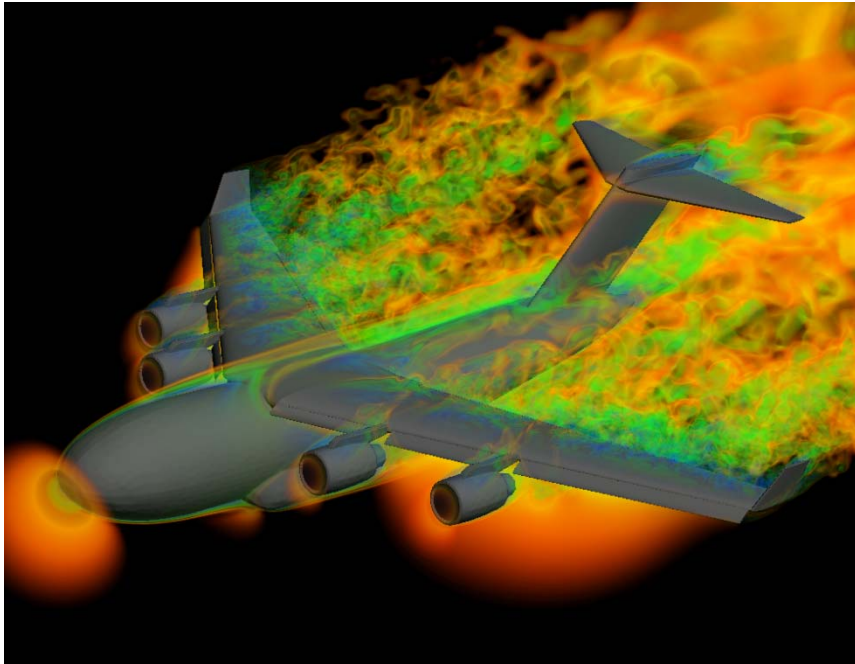


cpus

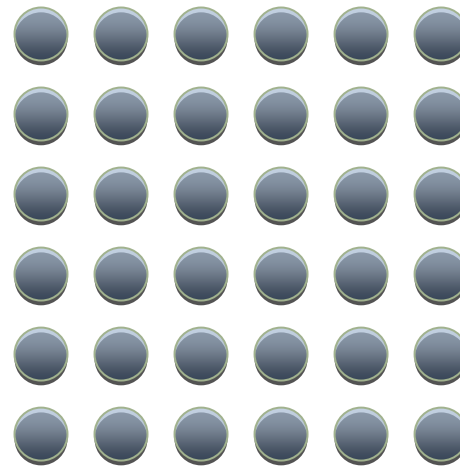
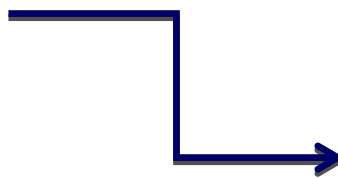
Shared Memory Directives

- 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 distributed memory not so successful
- OpenMP is a good model to use within a node

Cooperating Processes Models



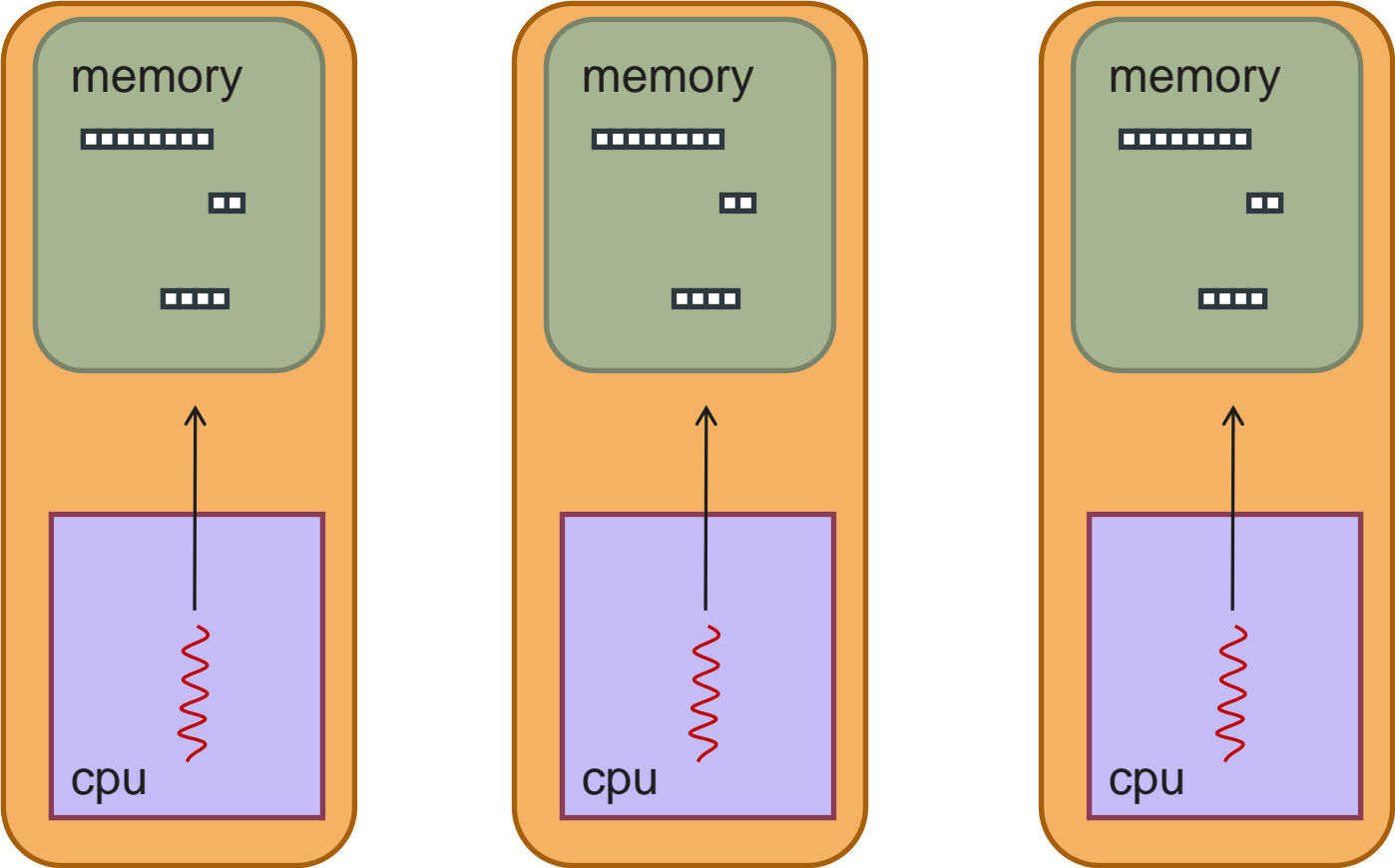
PROBLEM



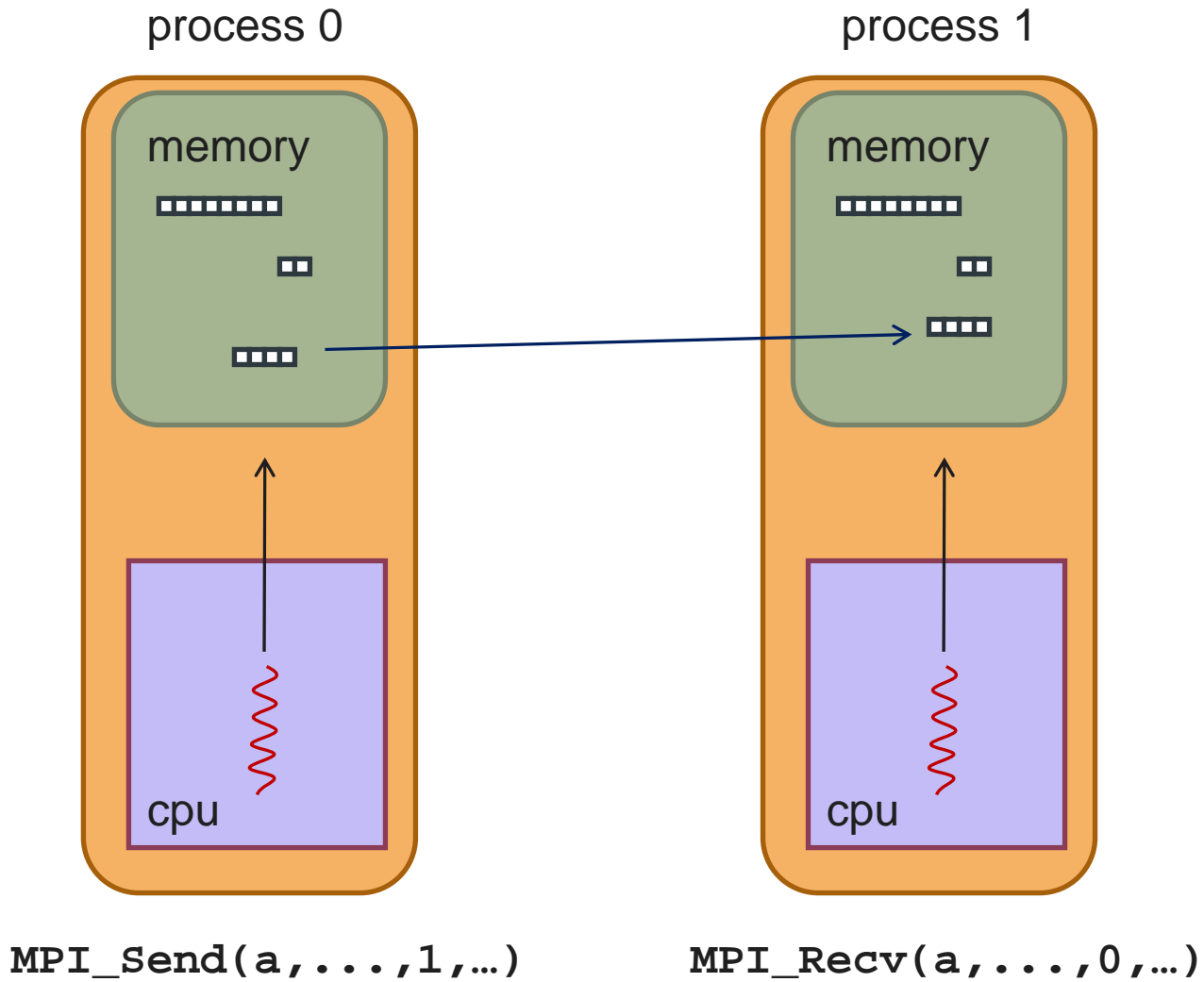
processes

Message Passing, MPI

process



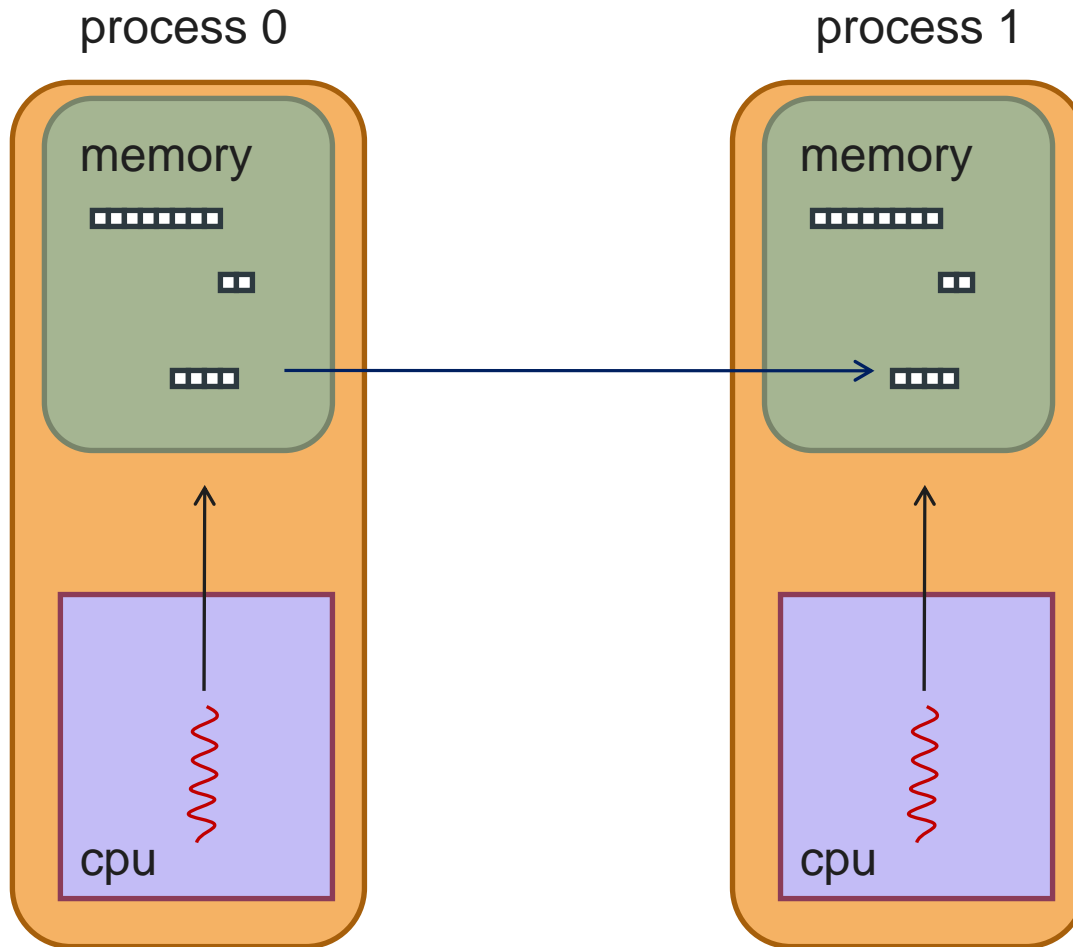
MPI



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
- Mostly used in two-sided way, each endpoint coordinates in sending and receiving

SHMEM

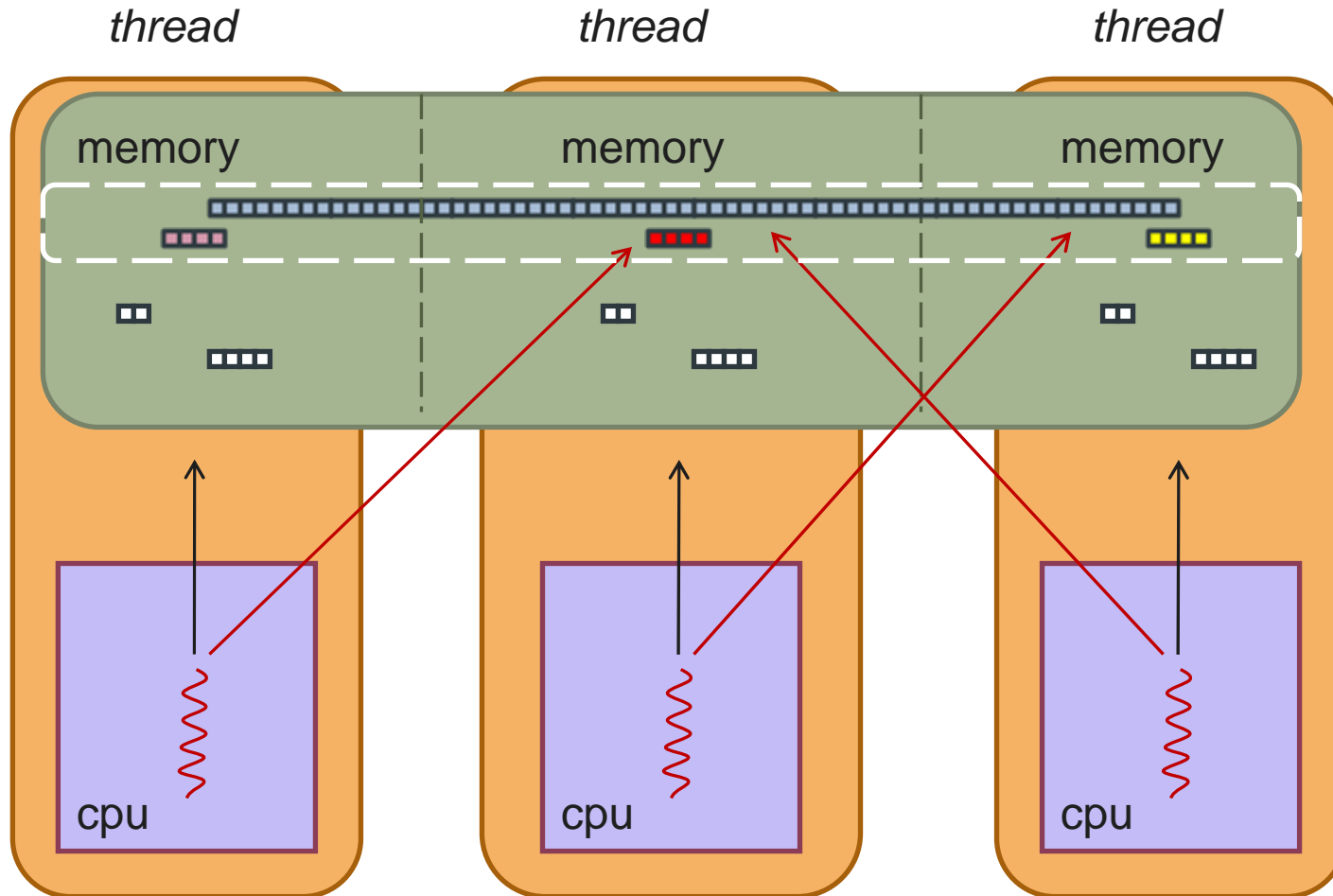


`shmem_put(a, b, ...)`

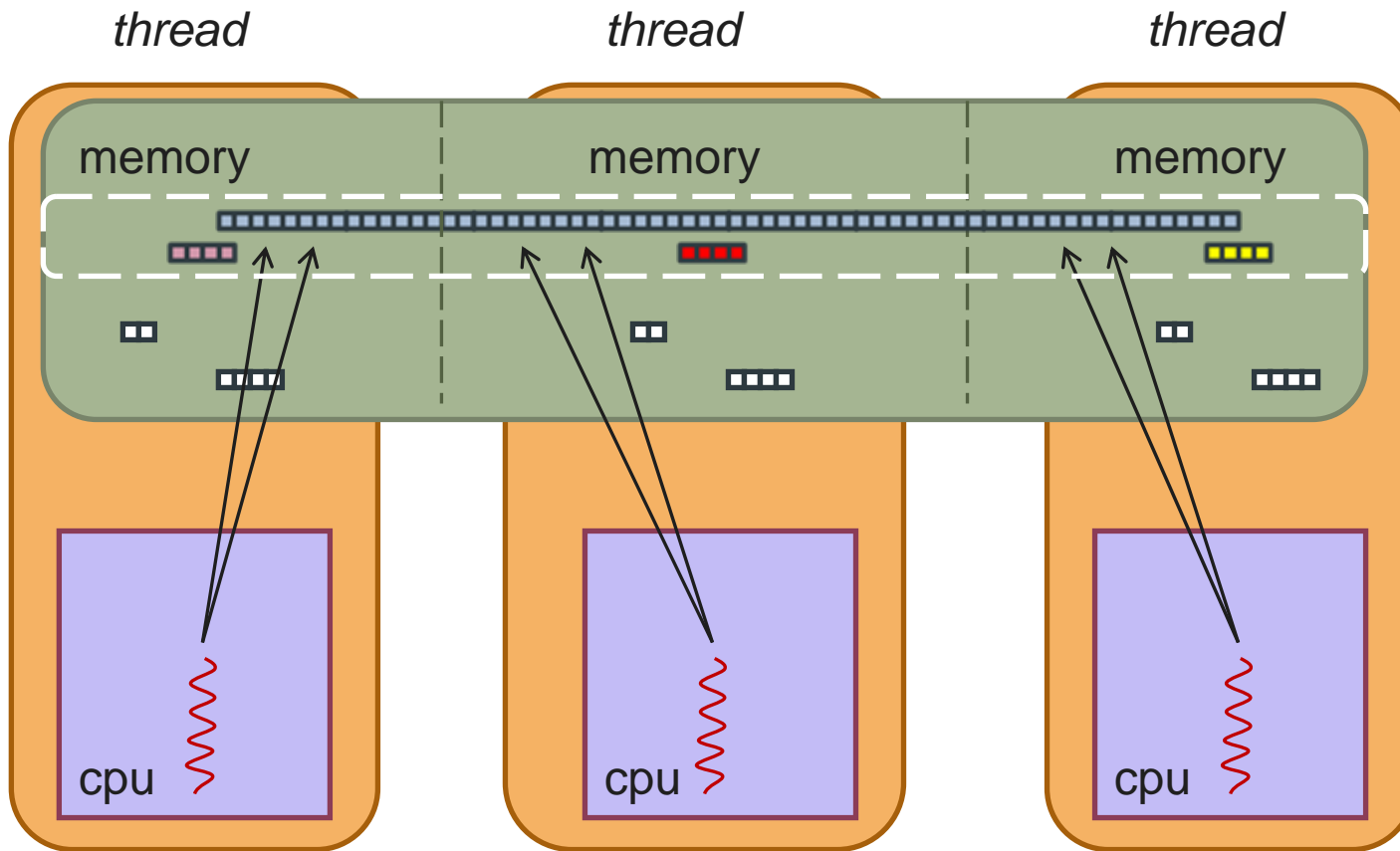
SHMEM

- Participating processes communicate using an API
- Fundamental operations are based on one-sided PUT and GET
- Need to use symmetric memory locations
- Remote side of communication does not participate
- Can test for completion
- Barriers and collectives
- Popular on Cray and SGI hardware, also Blue Gene version
- To make sense needs hardware support for low-latency RDMA-type operations

UPC



UPC



```
upc_forall(i=0;i<32;i++;affinity)  
  a[i]=a[i]*2  
end do
```

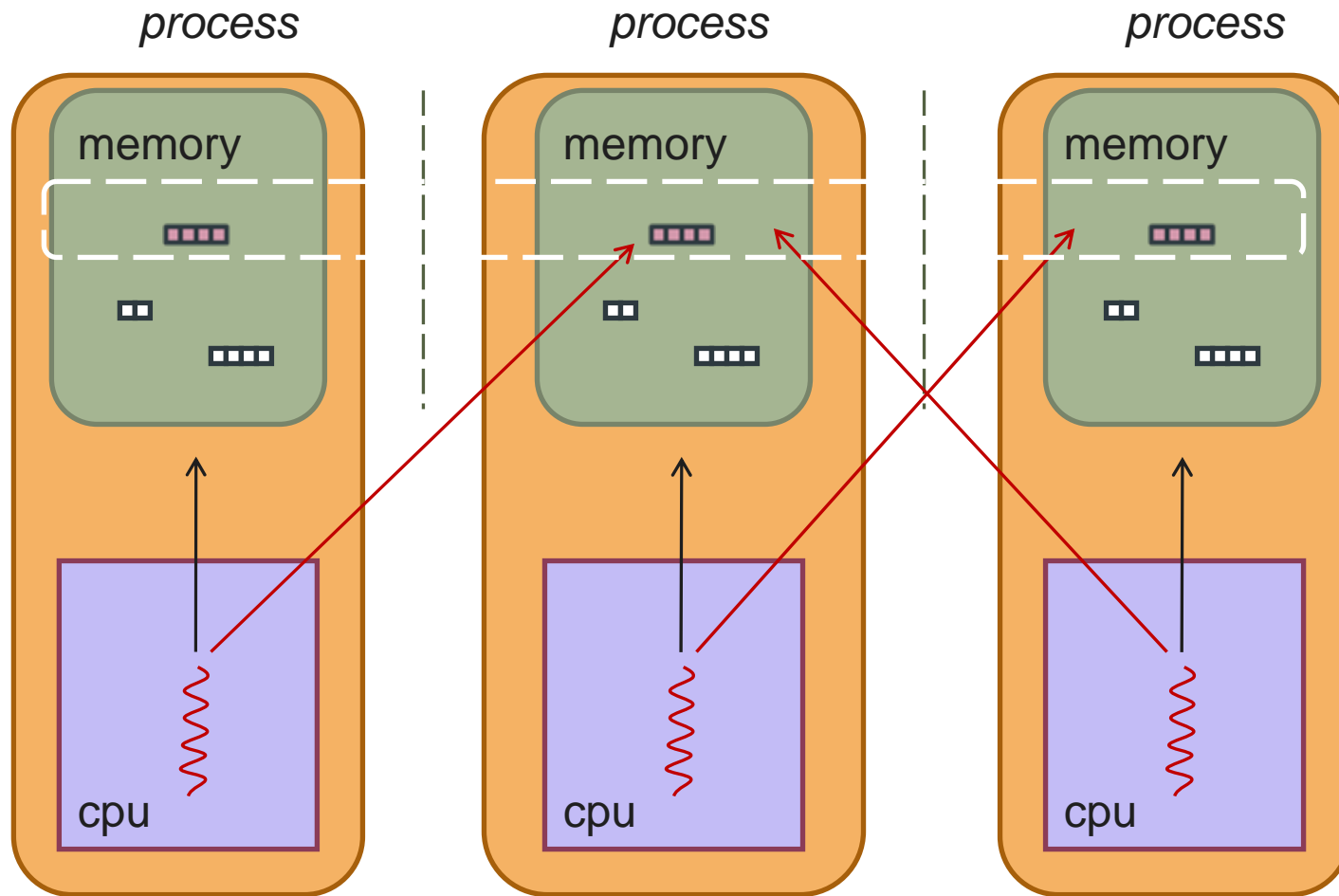
UPC

- Extension to ISO C99
- Participating “*threads*”
- New *shared* data structures
 - shared pointers to distributed data (block or cyclic)
 - pointers to shared data local to a thread
 - Synchronization
- Language constructs to divide up work on shared data
 - `upc_forall()` to distribute iterations of `for()` loop
- Extensions for collectives
- Both commercial and open source compilers available
 - Cray, HP, IBM
 - Berkeley UPC (from LBL), GCC UPC

Fortran 2008 coarray model

- Example of a Partitioned Global Address Space (PGAS) model
- Set of participating processes like MPI
- Participating processes have access to **local memory** via standard program mechanisms
- Access to **remote memory** is directly supported by the language

Fortran coarray model



Fortran coarrays

- Remote access is a full feature of the language:
 - Type checking
 - Opportunity to optimize communication
- No 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

Basic Features



Coarray Fortran

"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

Some History

- Introduced in current form by Numrich and Reid in 1998 as a simple extension to Fortran 95 for parallel processing
- Many years of experience, mainly on Cray hardware
- A set of core features are expected to form part of the Fortran 2008 standard
- Additional features are expected to be published in a Technical Report in due course.

How Does It Work?

- SPMD - Single Program, Multiple Data
 - single program replicated a fixed number of times
- Each replication is called an *image*
- Images are executed asynchronously
 - execution path may differ from image to image
 - some situations cause images to synchronize
- Images access remote data using *coarrays*
- Normal rules of Fortran apply

What are coarrays?

- Arrays or scalars that can be accessed remotely
 - images can access data objects on any other image
- Additional Fortran syntax for coarrays
 - Specifying a codimension declares a coarray

```
real, dimension(10), codimension[*]:: x  
real :: x(10)[*]
```

- these are equivalent declarations of a array x of size 10 on each image
- x is now remotely accessible
- coarrays have the same size on each image!

Accessing coarrays

```
integer :: a(4)[*], b(4)[*] !declare coarrays  
b(:) = a(:)[n]           ! copy
```

- integer arrays a and b declared to be size 4 on all images
- copy array a from remote image n into local array b
- () for local access [] for remote access
- e.g. for two images and n = 2:

image 1

a	1	2	3	4
b	8	9	3	8

image 2

a	2	9	3	7
b	10	19	32	13

Synchronisation

- Be careful when updating coarrays:
 - 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 that data sent to us has arrived?
- Fortran provides intrinsic synchronisation statements
- For example, barrier for synchronisation of all images:

```
sync all
```

- do not make assumptions about execution timing on images
 - unless executed after synchronisation
 - Note there is implicit synchronisation at program start

Retrieving information about images

- Two intrinsics provide index of this image and number of images
 - `this_image()` (image indexes start at 1)
 - `num_images()`

```
real :: x[*]
if(this_image() == 1) then
  read *,x
  do image = 2,num_images()
    x[image] = x
  end do
end if
sync all
```

Making remote references

- We used a loop over images

```
do image = 2,num_images()  
  x[image] = x  
end do
```

- Note that array indexing within the coindex is not allowed so we can not write

```
x[2:num_images()] = x      ! illegal
```

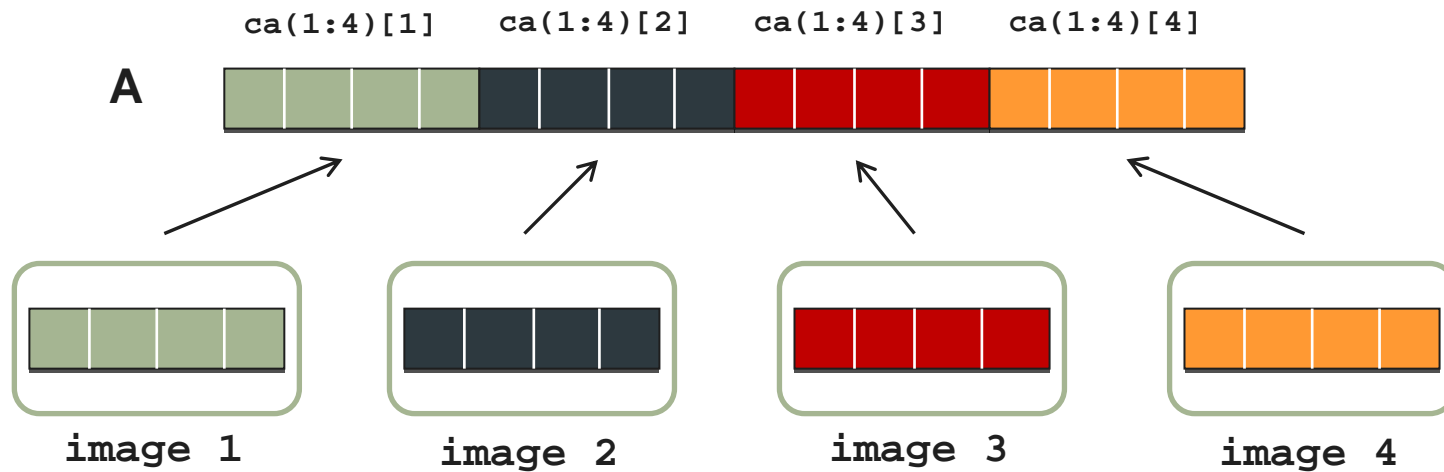
Data usage

- coarrays have the same size on every image
- Declarations:
 - round brackets () describe rank, shape and extent of local data
 - square brackets [] describe layout of images that hold local data
- Many HPC problems have physical quantities mapped to n-dimensional grids
- You need to implement your view of global data from the local coarrays as Fortran does not provide the global view
 - You can be flexible with the coindexing (see later)
 - You can use any access pattern you wish

Data usage

- print out a 16 element “global” integer array A from 4 processors
 - 4 elements per processor = 4 coarrays on 4 images

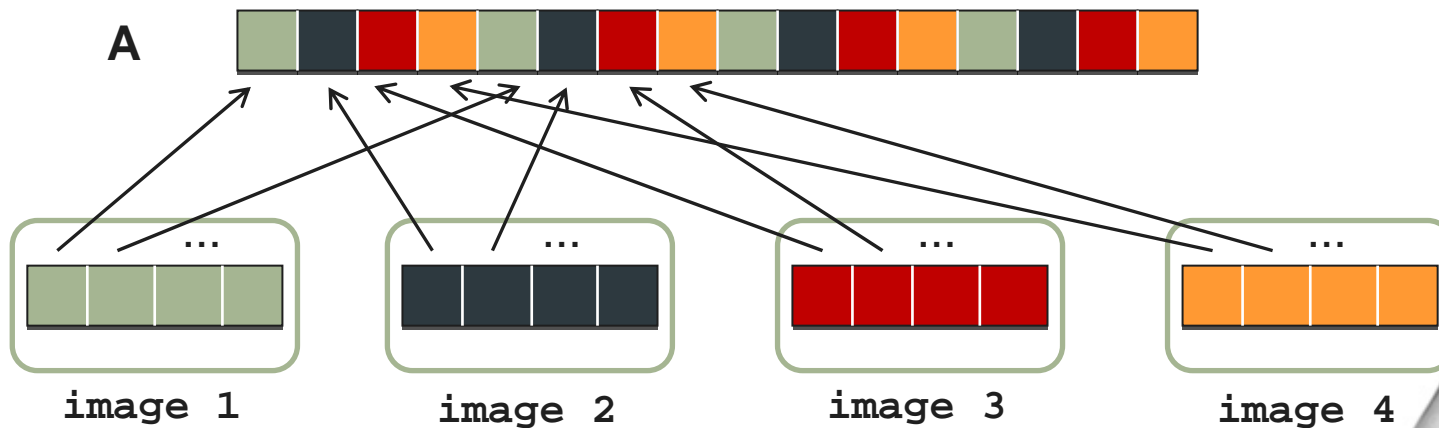
```
integer :: ca(4)[*]  
do image=1,num_images()  
  print *,ca[image]  
end do
```



1D cyclic data access

- coarray declarations remain unchanged
 - but we use a cyclic access pattern

```
integer :: ca(4)[*]  
do i=1,4  
  do image=1,num_images()  
    print *,ca(i)[image]  
  end do  
end do
```



Synchronisation

- code execution on images is independent
 - programmer has to control execution using synchronisation
- synchronise before accessing coarrays
 - ensure content is not updated from remote images before you can use it
- synchronise after accessing coarrays
 - ensure new content is available to all images
- implicit synchronisation after variable declarations at first executable statement
 - guarantees coarrays exist on all images when your first program statement is executed
- We will revisit this topic later

Example: maximum of array

```
real :: a(10)
```

```
real :: maximum[*]
```

```
call random_number(a)
```

```
maximum = maxval(a)
```

```
sync all
```

```
if (this_image() == 1) then
```

```
  do image = 2, num_images()
```

```
    maximum = max(maximum, maximum[image])
```

```
  end do
```

```
  do image = 2, num_images()
```

```
    maximum[image] = maximum
```

```
  end do
```

```
end if
```

```
sync all
```

implicit synchronisation

ensure all images set local maximum

ensure all images have copy of maximum value

Recap

We now know the basics of coarrays

- declarations
- references with []
- `this_image()` and `num_images()`
- `sync all`

Now consider a full 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%

Observations so far on coarrays

- Natural extension, easy to learn
- Makes parallel parts of program obvious (syntax)
- Part of Fortran language (type checking, etc)
- No mapping of data to buffers (or copying) or creation of complex types (as we might have with MPI)
- Compiler can optimize for communication

- More observations later...

Exercise Session 1

- Look at the Exercise Notes document for full details
- Write, compile and run a “Hello World” program that prints out the value of the running image’s image index and the number of images
- Extend the simple Fortran code provided in order to perform operations on parts of a picture using coarrays

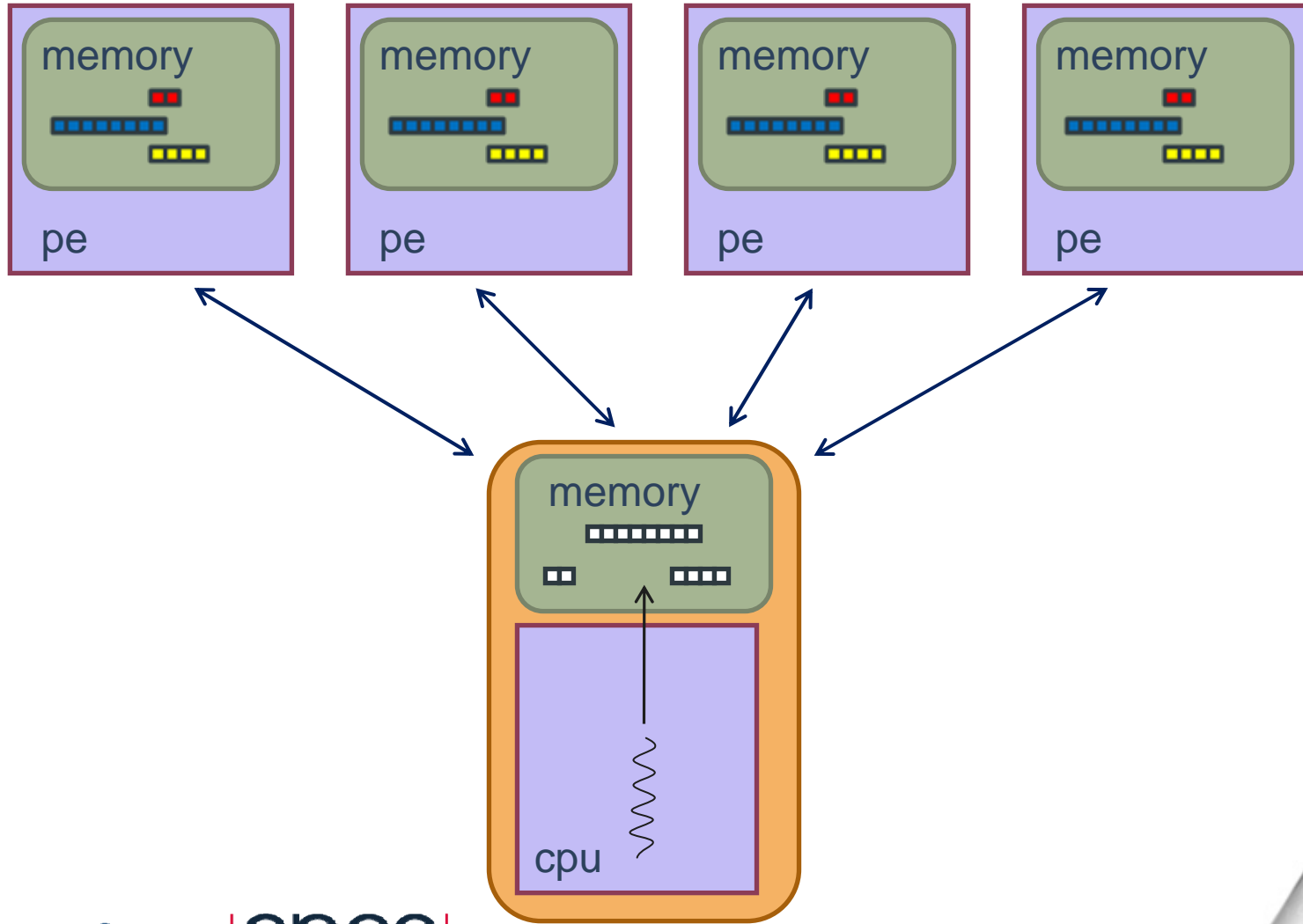
Backup Slides

HPF model

High Performance Fortran (HPF)

- Data Parallel programming model
- Single thread of control
- Arrays can be distributed and operated on in parallel
- Loosely synchronous
- Parallelism mainly from Fortran 90 array syntax, FORALL and intrinsics.
- This model popular on SIMD hardware (AMT DAP, Connection Machines) but extended to clusters where control thread is replicated

HPF



HPF

