Parallel Programming without MPI – Using Coarrays in Fortran

SUMMERSCHOOL 2015

August 5, 2015

Ge Baolai
SHARCNET
Western University

Outline

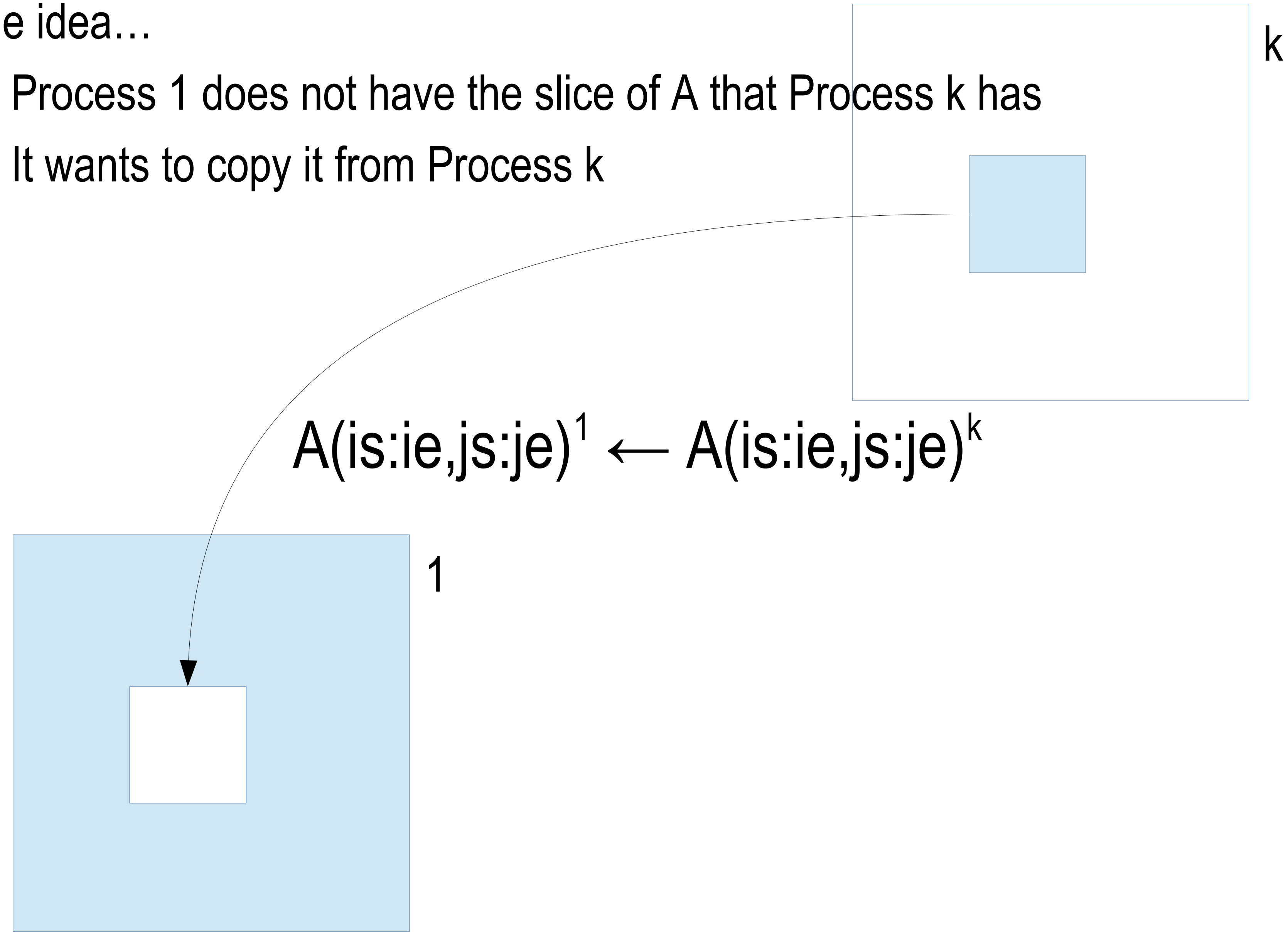
- What is coarray
- How to write: Terms, syntax
- How to compile and run
- A case study
- Performance

Parallel processing and coarrays

Coarray

The idea…

- Process 1 does not have the slice of A that Process k has
- It wants to copy it from Process k



$$A(is:ie,js:je)^1 \leftarrow A(is:ie,js:je)^k$$

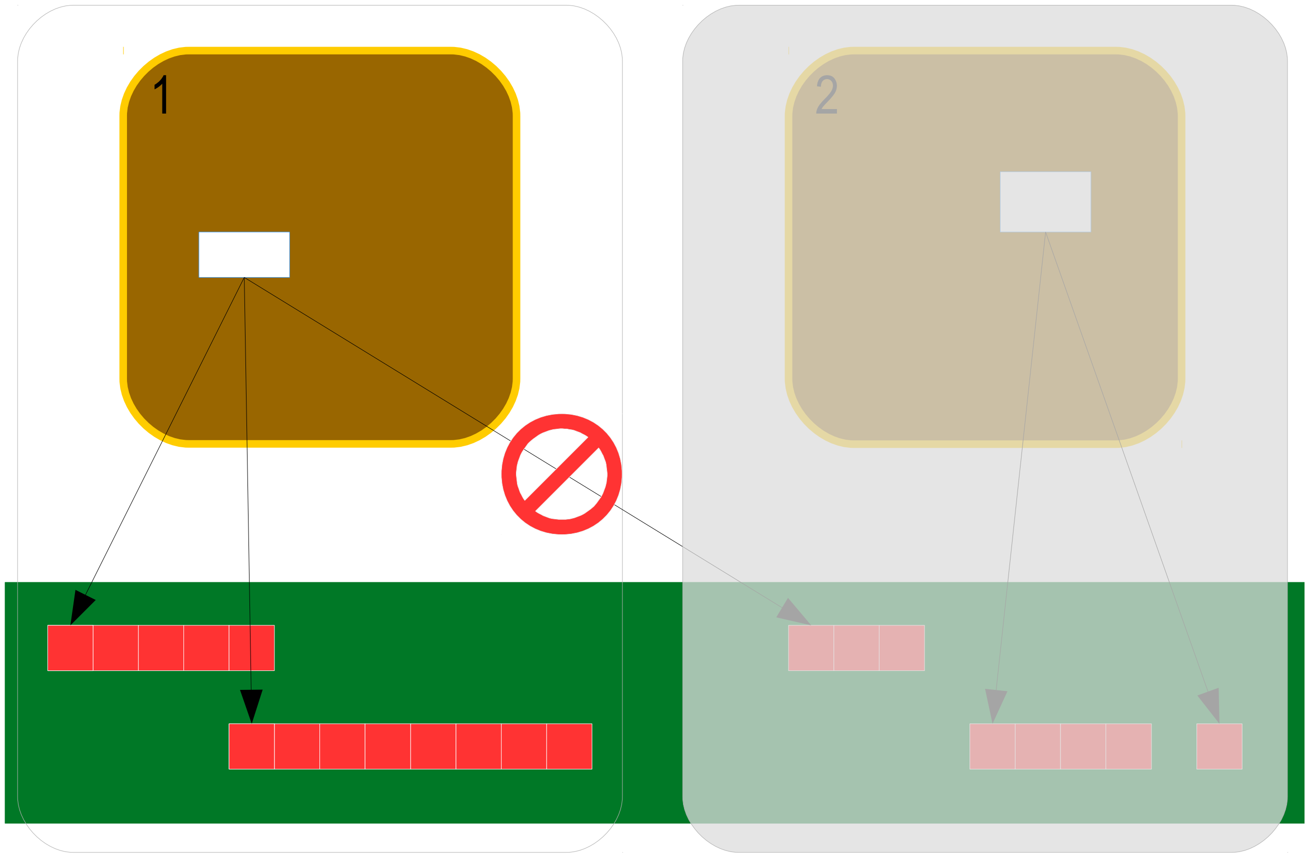
Western

Shared Memory

SHARCNET™ SEMINAR SERIES 2015

Single Processes

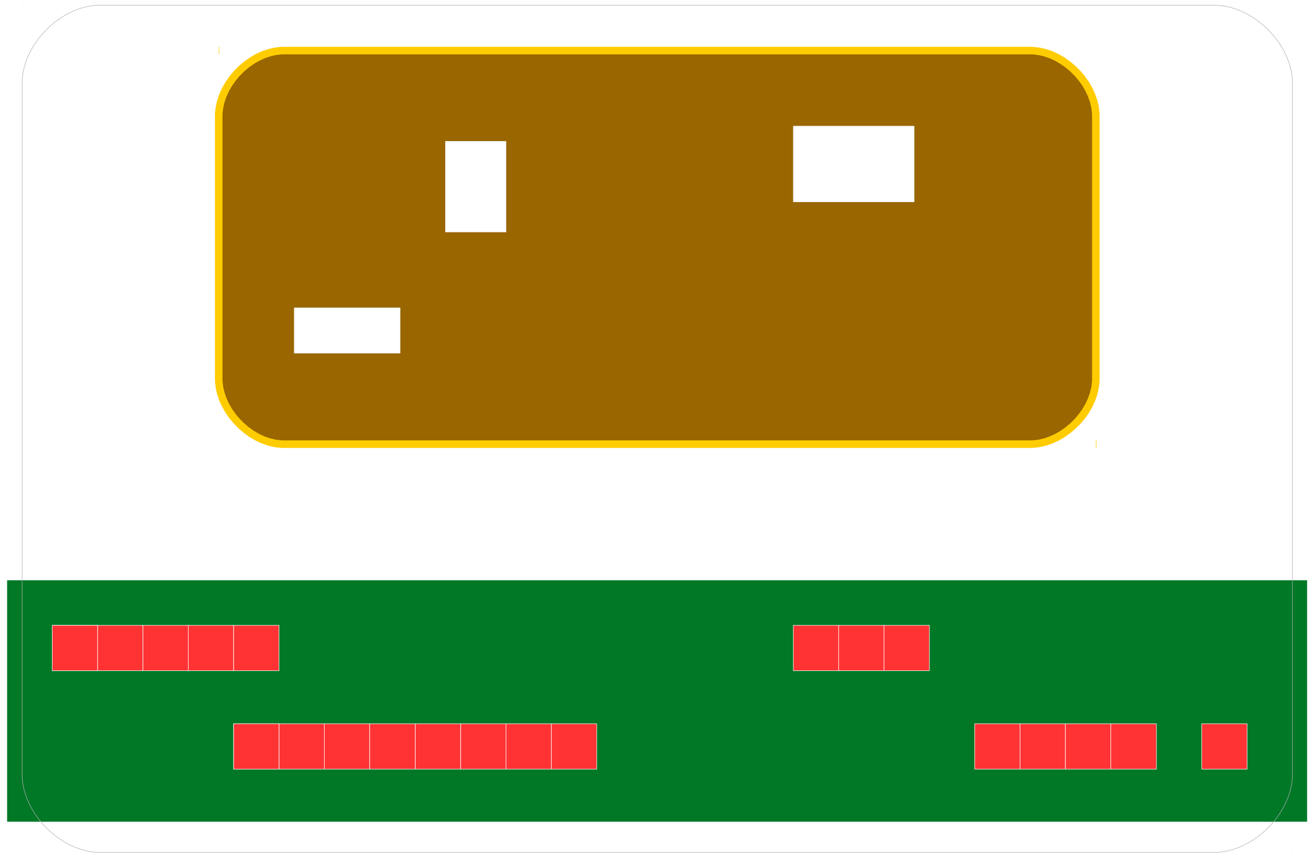
- One process does not see the content of another



Shared Memory

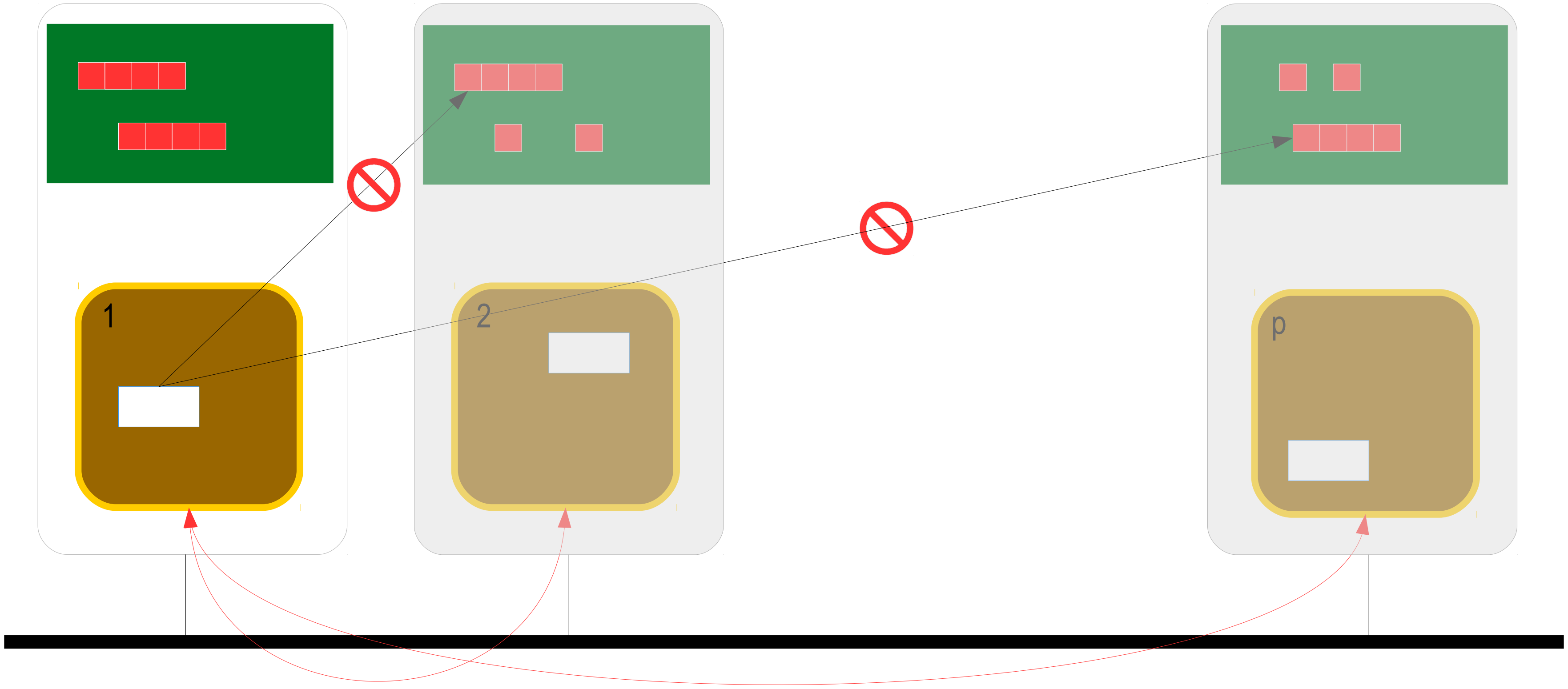
Multithreaded Processes

- Threads on multicores within a process see all data within the process



Distributed/Shared Memory - MPI

- One process does not see the content of others
- A process generally can't access the content of another directly
- Access data held by others is via message passing (e.g. MPI)



Coarray

How do we do it with MPI? we would write

- On rank 1, to receive data from rank k

```
MPI_Recv(A(is:ie,js:je),n,MPI_REAL,k,tag,MPI_COMM_WORLD,status)
```

Or, more generic

```
MPI_Recv(buffer,n,MPI_REAL,k,tag,MPI_COMM_WORLD,status)
```

Unmarshal buffered data into A

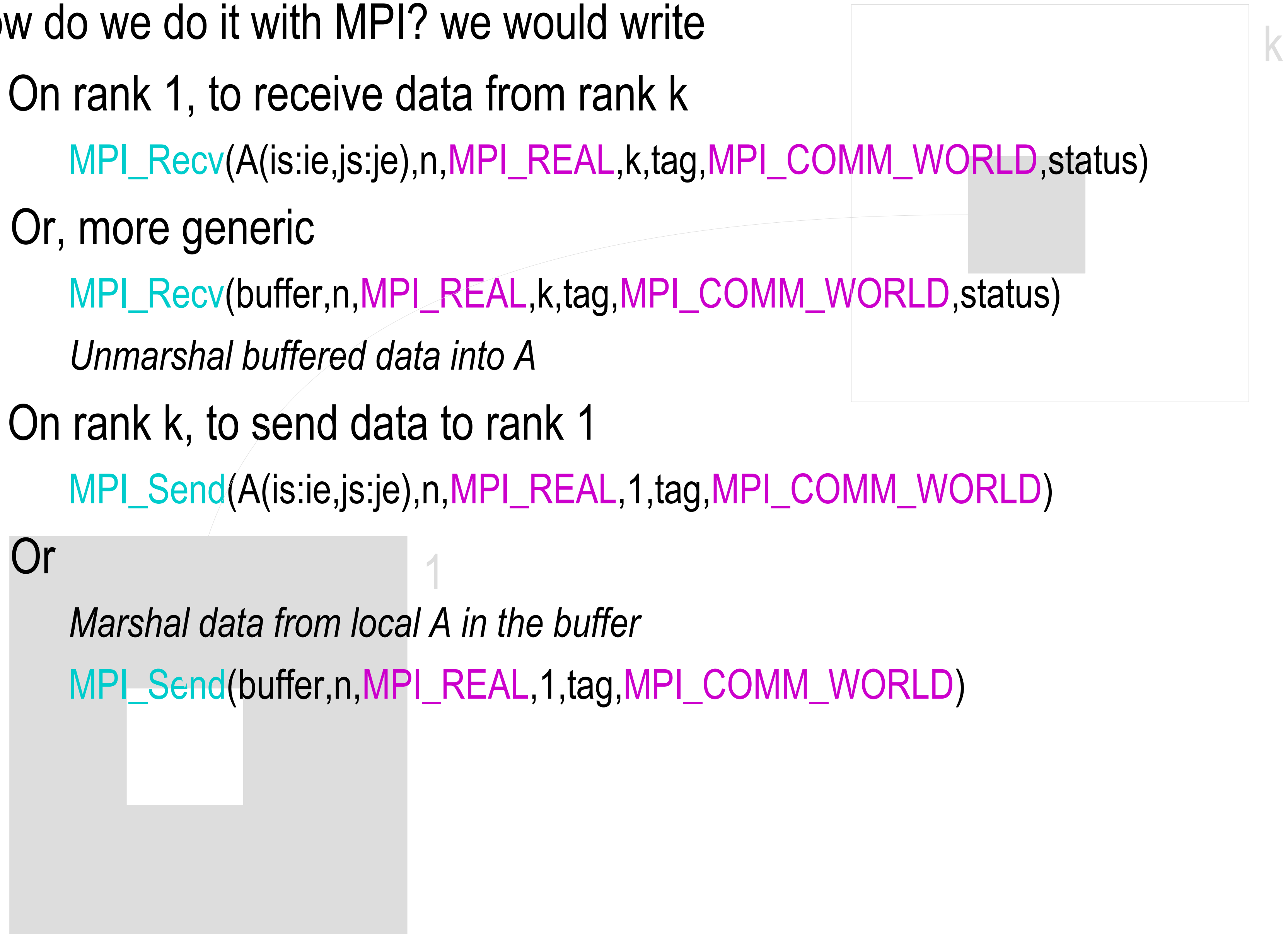
- On rank k, to send data to rank 1

```
MPI_Send(A(is:ie,js:je),n,MPI_REAL,1,tag,MPI_COMM_WORLD)
```

Or

Marshal data from local A in the buffer

```
MPI_Send(buffer,n,MPI_REAL,1,tag,MPI_COMM_WORLD)
```



Coarray

But what we really want is symbolically as simple as this…

$$A(is:ie,js:je) \leftarrow A(is:ie,js:je)^k$$

Coarray

So here comes this

```
A(is:ie,js:je) = A(is:ie,js:je)[k]
```

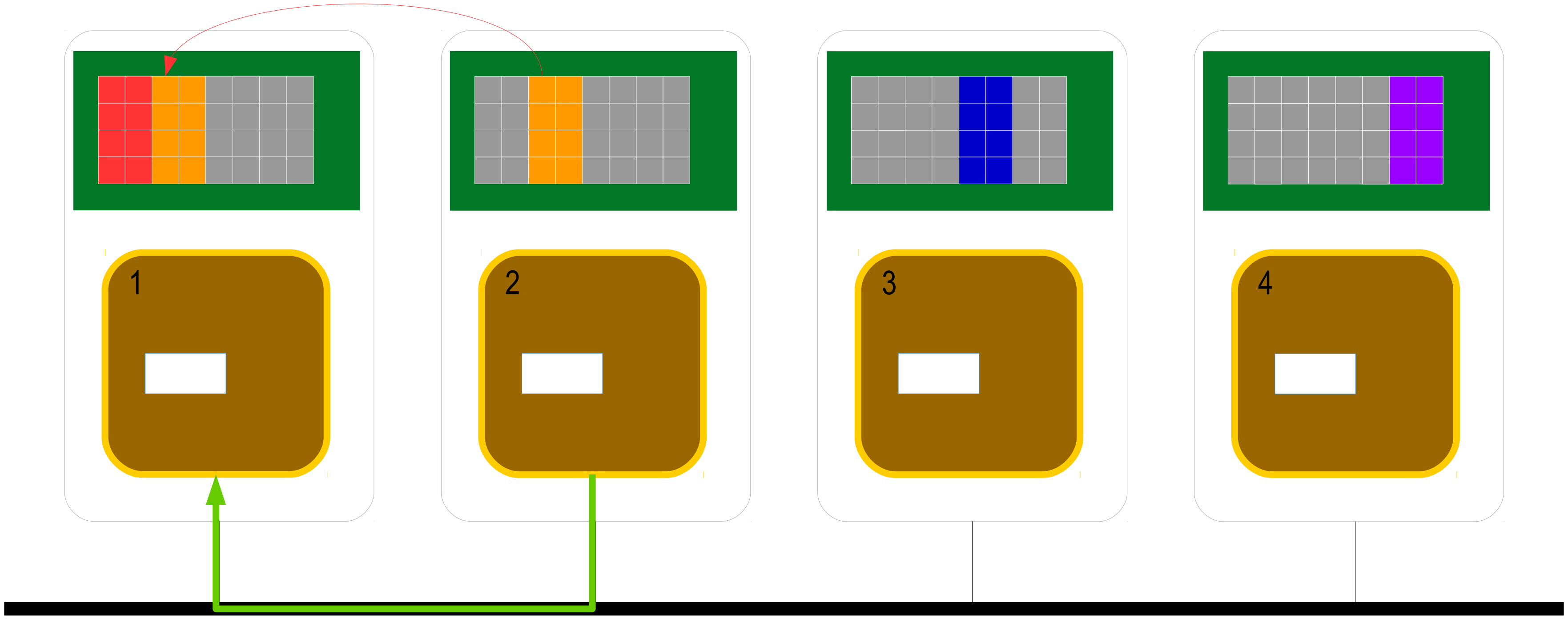
Coarray

```
program main
   real :: x(10000), u(10000)
   complex :: y(10000)
   real :: A(1000,1000)[*]          ! Indicate to be possessed by every process

   ... ...
                  A(is:ie,js:je) = A(is:ie,js:je)[k]







end program main
```

Distributed Shared Memory

- Every process – *image* – holds the same size object A
- A is local to the image; A[k] references to the A on image k.
- Access to A[k] invokes underlying data communications, e.g. on 1

A(1:4,3:4) = A(1:4,3:4)[2]



Coarray

```
program main
   real :: x(10000), u(10000)
   complex :: y(10000)
   real :: A(1000,1000)[*]

   … …
                A(i1:i2,j1:j2) = A(i3:i4,j3:j4)[k]

end program main
```

Coarray

```
program main
   real :: x(10000), u(10000)
   complex :: y(10000)
   real :: A(1000,1000)[*]

   … ...
                 A(i1:i2,j1:j2) = A(i3:i4,j3:j4)[k]

end program main
```

History and Current Development

- Introduced by R. W. Numrich and J. Reid in 1998.
- Many years of experience, as an extension to Fortran, mainly on Cray hardware.
- Adopted as a language feature as part of the ISO standard (2008).
- Additional features expected to be published in due course.
- Compilers are catching up, e.g. popular ones
  - **Intel**
  - **GCC**
  - **G95** project
- Support libraries
  - Opencoarrays project
  - Rice University

History: Trend

Models and tools for the next generation of HPC architectures?

- Coarray
- Unified Parallel C (UPC) } Partitioned Global Address Spaces (PGAS)
- Global arrays, SHMEM
- OpenACC, OpenMP
- New languages – for programmability and performance? For example
  - Chapel
  - X10
  - Fortress (ceased)

How does it work?

Coarray: Parallel Programming without MPI!

Coarray Syntax

- Globally addressible arrays amongst processes – ***images***.
- Each image holds the same size copies of data objects – **coarrays**.
- Data objects with subscripts in square brackets indicates coarray, in any of the following forms

```
– X[*]          ! Upper bound not set
– X[16]         ! Max images 16
– X[p,q]        ! p-by-q images
– X[p,*]        ! Last bound not set
– X[8,0:7,1:*]  ! Three codimensions
```

- [*identifier*] defines the number of images (and topology)
- Upper bound usually not defined.

Example

```
! Array coarrays
real :: a(1000,1000)[*]
real :: b(1000,1000)[16,16], x(10000)[16]
complex, allocatable, codimension[:] :: z(:)
complex, allocatable :: zz(:,:)[:]
! Scalar coarrays
integer :: m[*], n[*]

if (this_image() == 1) then
  input data
  do image = 1, num_images()
    u[image] = u ! Send u to all images
  enddo
endif
```

Coarray: Parallel Programming without MPI!

Coarray Syntax (cont'd)

- Objects of derived types

  type(*type1*) :: p[*]

  type(*type2*), allocatable :: u[:]

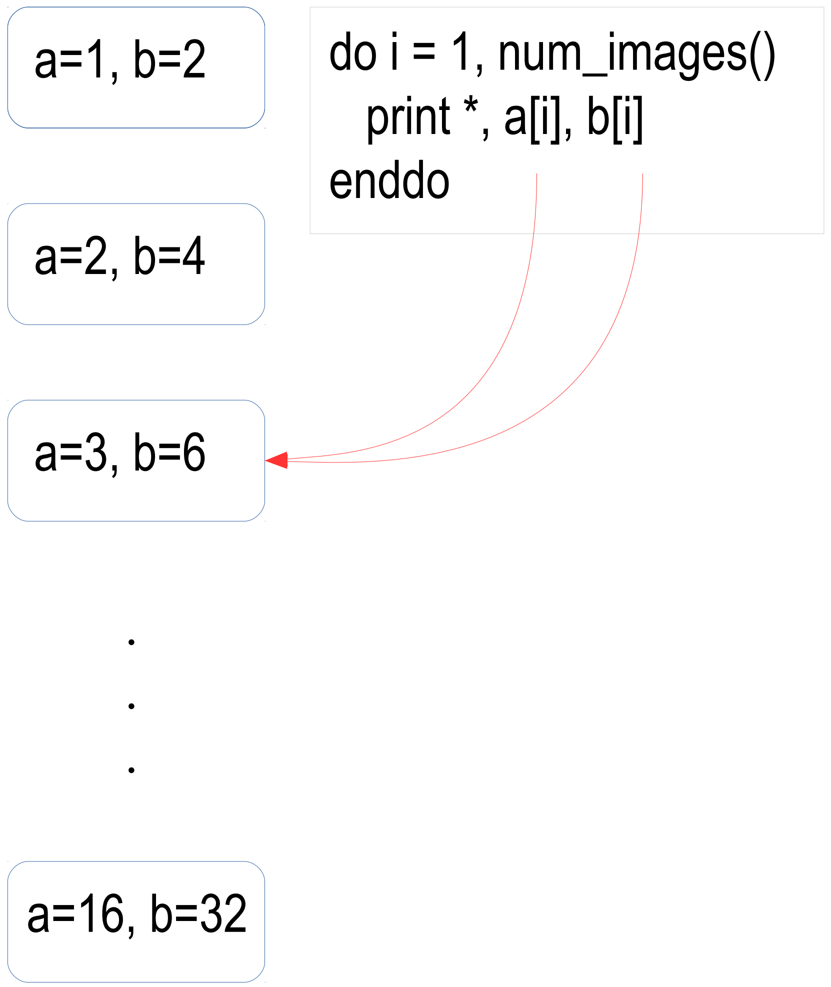
Example

```
! Derived data types
type particle
   real :: m
   real :: x, y, z
   real :: u, v, w
end type particle

! Static storage
type(particle):: p(1000000)[*]

! Dynamic storage
type(particle), allocatable:: p(:)[:]

u = p(k)[16]%u
v = p(k)[16]%v
```

Coarray: Parallel Programming without MPI!

Concept

Images **Execution of code**



```
do i = 1, num_images()
   print *, a[i], b[i]
enddo
```

Example

```
program try_coarray
  real :: a[*]       ! Declare a as coarray obj
  real, codimension[*] :: b ! Or this way

  ! a and b below are local to the iamge
  a = this_image()
  b = this_image()*2

  ! Access a and b on other images
  if (this_image() == 1) then
    do image = 1, num_images()
      print *, 'Image', this_image(), a[i], b[i]
    enddo
  endif
end program try_coarray
```

Coarray: Accessing Coarrays

SEMINAR SERIES 2015

- Access coarray objects by referencing to the object with an image index in square [ ], e.g.

```
x[i] = y          ! Put local value y to x on image i
z = z[i]          ! Get value of z on image i and assign it to local z
a(:,:)[i] = b(:)  ! Whole array assignment not used in coarrays
```

- Note this is executed by every image (due to SPMD model)

```
x[16] = 1
```

- For selective execution

```
if (this_image() == 16) then
  x = 1
endif
```

- Note Fortran arrays use ( ) for array elements, not [ ], so there is no confusion!

Examples

We are now ready to write our first complete parallel code

Example: Broadcast

```
program ex1
   implicit none
   real :: z[*]
   integer :: i

   sync all
   if (this_image() == 1) then
      read *, z
      print '("Image",i4,": before: z=",f10.5)', this_image(), z
      do i = 2, num_images()
         z[i] = z
      enddo
   endif
   sync all
   print '("Image",i4,": after: z=",f10.5)', this_image(), z
end program ex1
```

Example: Broadcast

```
program ex1
  implicit none
  real :: z[*]
  integer :: i

  sync all
  if (this_image() == 1) then
    read *, z
    print '("Image",i4,": before: z=",f10.5)', this_image(), z
    do i = 2, num_images()
      z[i] = z
    enddo
  endif
  sync all
  print '("Image",i4,": after: z=",f10.5)', this_image(), z
end program ex1
```

Coarray: Synchronization

sync images (image-set)

- Sync with one image

  ```
  sync images (16)
  ```
- Sync with a set of images

  ```
  sync images ([1,3,5,7])
  ```
- Sync with every other

  ```
  sync images (*)
  ```
- Sync all

  ```
  sync all
  if (this_image() == 1) then
     do image = 1, num_images()
        u[image] = u
     enddo
  endif
  sync all
  ```

sync all and sync images(*)

- sync images (*) and sync all (see right) are not equivalent:

  ```
  if (this_image() == 1) then
     Set data needed by all others
     sync images (*)
  else
     sync image (1)
     Get data set by image 1
  endif
  ```

Coarray: Locking and Critical Region

Locking

- Although frequent lock unlock are not expected in numerical computations, they are useful in some operations, such as push and pop operations of a queue and stack, etc.
- Use of ISO Fortran intrinsic modules are recommended, e.g.

```
subroutine job_manager(...)
  use, intrinsic :: iso_fortran_env, only: lock_type
  type(lock_type) :: stack_lock[*]
  … ...
  lock (stack_lock)
  if (stack_size > 0) then
    job = pop(stack)
  endif
  unlock (stack_lock)
  … ...
end subroutine job_manager
```

Coarray: Locking and Critical Region

Critical Section

- Multiple images try to update the object p ***on image 6***, but only one at a time

```
critical
   p[6] = p[6] + 1
   … …
end critical
```

Example: Harvest

```
program ex2
   character(80) :: host[*]  ! Note: host – local; host[i] – on image i
   integer :: i

   call get_environment_variable("HOSTNAME”,value=host)

   if (this_image() == 1) then
      do i = 1, num_images()
         print *, 'Hello from image', i, 'on host ', trim(host[i])
      enddo
   endif
end program ex2
```

Compiling coarray code

Compilers

GNU gfortran Compiler

- Requirements
  - Version 5.1 and newer
  - An MPI library compiled with GCC 5.1
  - A recent CAF (Coarray Fortran) MPI library libcaf_mpi, provided by the Opencoarrays project (http://www.opencoarrays.org/)
- *GCC 5.1: if to build yourself, include the essential options*

```
./configure --prefix=/opt/gcc/5.1.0 --disable-bootstrap --enable-static --enable-shared --enable-shared-libgcc --enable-languages=c,c++,fortran --disable-symvers --enable-threads=posix --enable-libatomic --enable-libgomp --enable-libquadmath --enable-libquadmath-support
```

- To compile

```
mpifort -std=f2008 -fcoarray=lib mycode.f90 -o mycode \
    -L${LIBCAF_MPI_PATH} -lcaf_mpi
```

- To run

```
mpirun -n num_procs ./mycode
```

Compilers

Intel Compiler

- Requirements
  - Intel compiler 14 and newer
  - Intel MPI runtime suite
  - Intel Cluster Toolkit (for distributed memory coarray, licenced)
- To compile

```
ifort -coarray=shared [ -coarray-num-images=8 ] mycode.f90 -o mycode
ifort -coarray=distributed mycode.f90 -o mycode
```

- To run

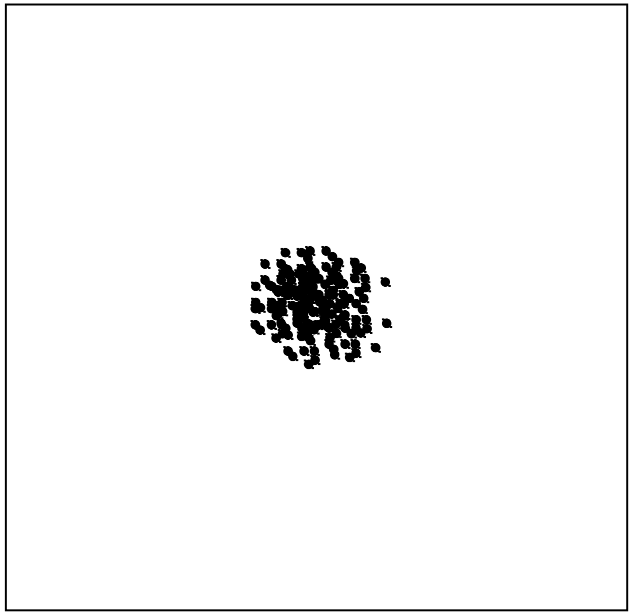
```
export PATH=$BIN_INTEL_MPIRT:$PATH
export LD_LIBRARY_PATH=$LIB_INTEL_MPIRT:$LD_LIBRARY_PATH
export FOR_COARRAY_NUM_IMAGES=8
./mycode
mpirun -n num_procs ./mycode
```

A case study: Diffusion

Diffusion

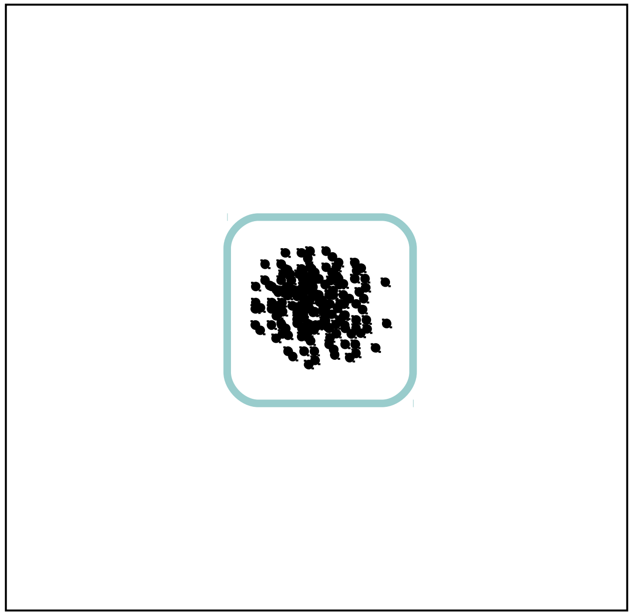
Problem

- Consider the density of some substances made of large number of particles.
- What's the density of the substance after some time?



Diffusion

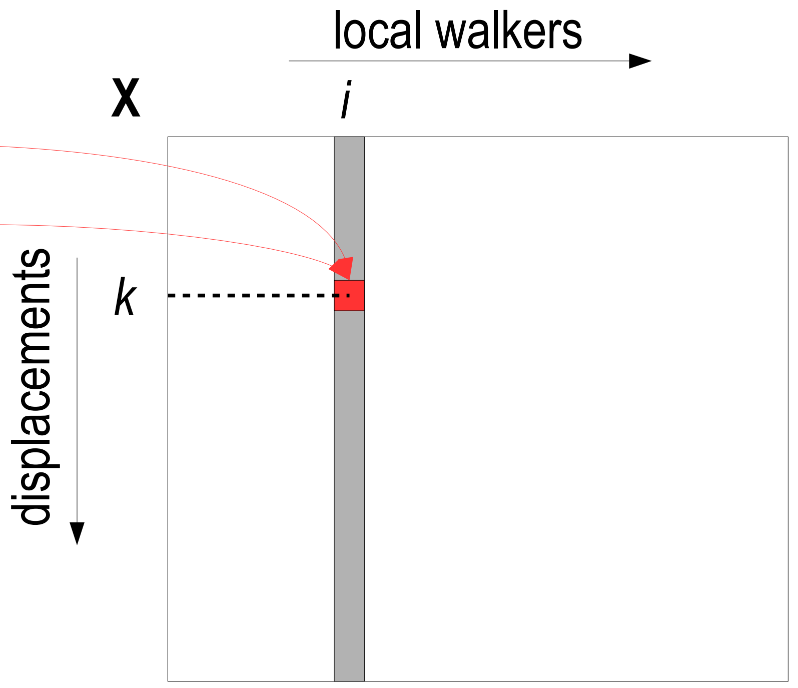
Implementation: We simulate the process – the displacements of particles from the origin over time – by random walks



Implementation (Serial) on one processor

- Use a 2D array **x**(num_steps,num_walkers) to store displacements of walkers over time steps.
- Set each walker to start from the origin. Simulate the position of each walker:

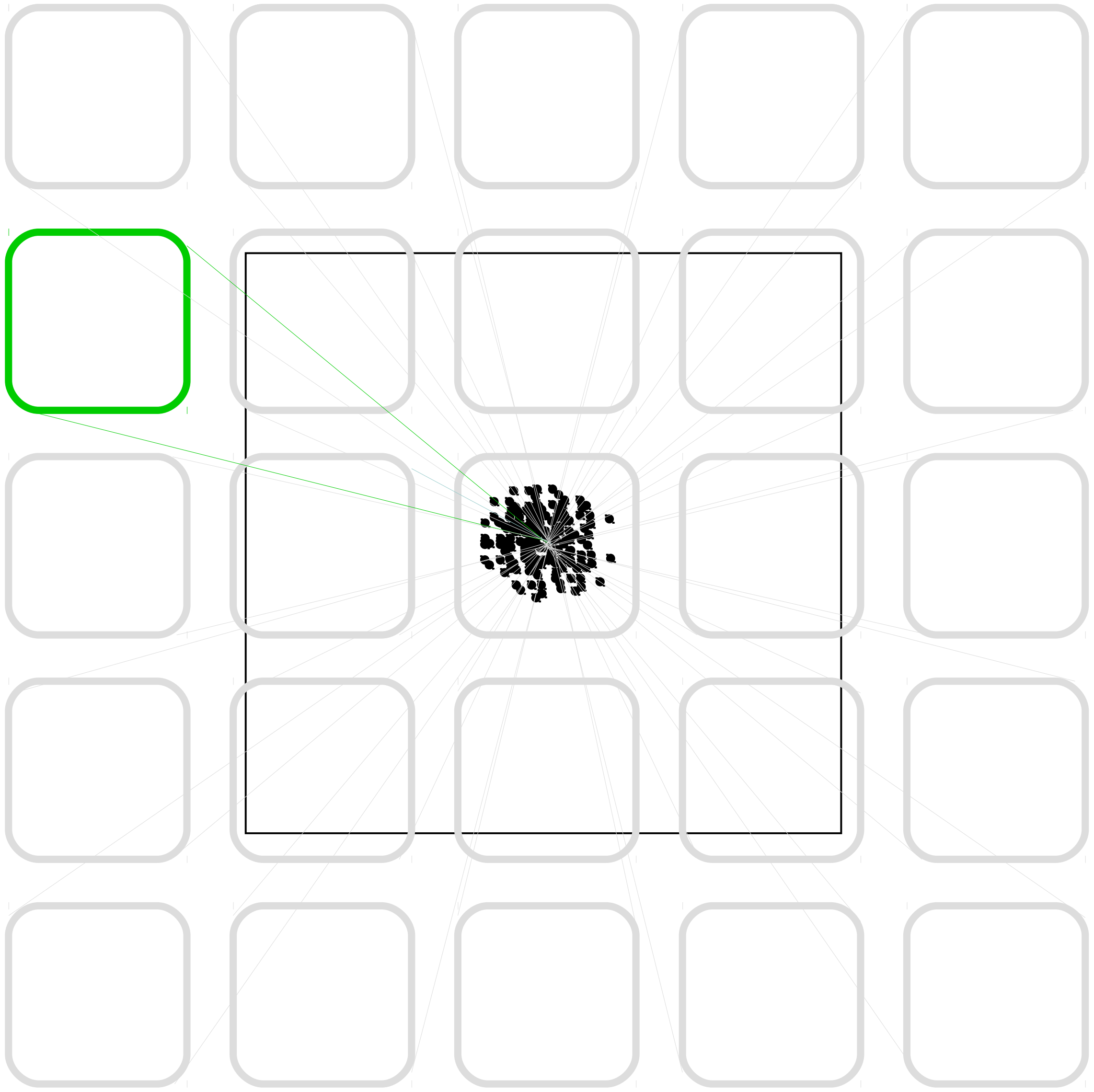
```
do i =1, num_walkers
   do k = 1, num_steps
      toss a coin
      if (heads up) then
         x(k,i) = x(k,i) + dx
      else
         x(k,i) = x(k,i) – dx
      endif
   enddo
enddo
```



2D array X of displacements

Diffusion

Implementation: Using multiple processors

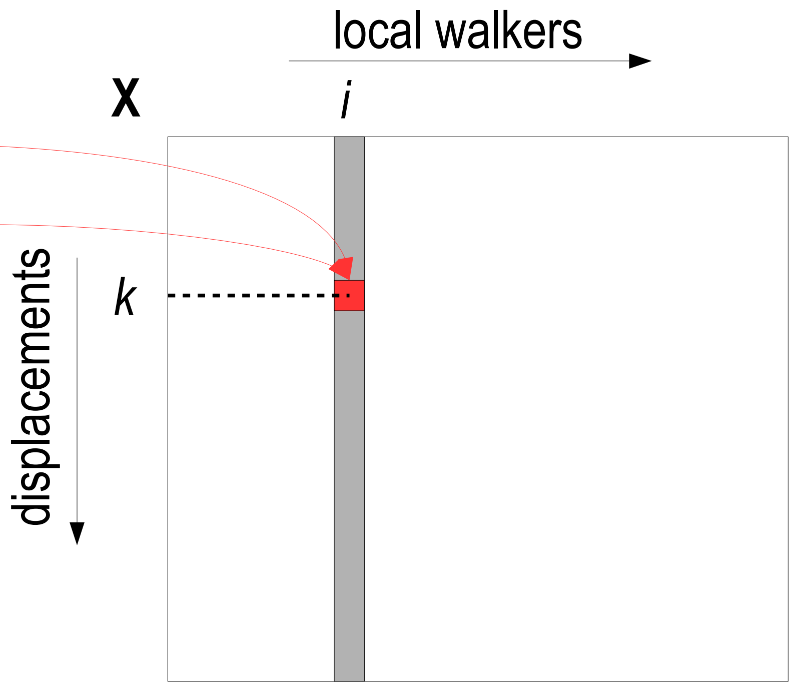


Diffusion

Implementation (Parallel) using multi-processors

- Use a 2D array **x**(num_steps,*local_walkers*) on each process – images – to store displacements over time steps.
- Set each walker to start from the origin. Simulate the position of each walker:

```
do i =1, local_walkers
   do k = 1, num_steps
      toss a coin
      if (heads up) then
         x(k,i) = x(k,i) + dx
      else
         x(k,i) = x(k,i) – dx
      endif
   enddo
enddo
```

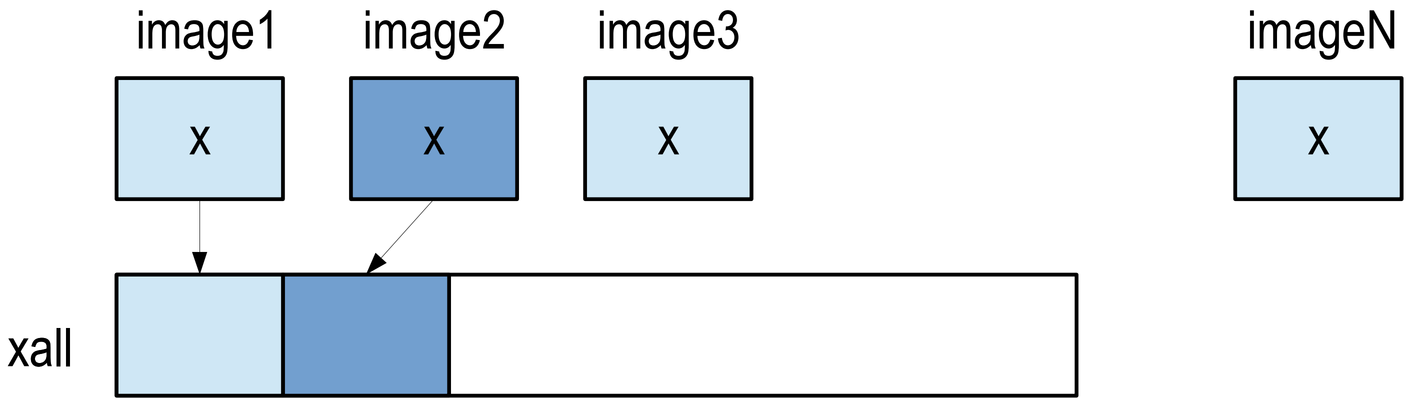


2D array X of displacements

Implementation (Parallel) – cont'd

- On image 1, use array **xall**(num_steps,num_walkers) to harvest local **x** from all

```
sync all
do i = 1, num_images()
   xall(:,local_walkers*(i-1)+1:local_walkers*i) = x(:,:)[i]
enddo
sync all
```



- Imsage 1 to perform post processing, e.g. the mean square displacement and histogram of **x**, etc.

Diffusion

```
program rwalk_p
  implicit none
  integer :: i, k, myid, nsteps[*], nwalkers, lwalkers[*]
  real, allocatable :: x(:,:)[:], x2(:), xall(:,:)
  real :: r

  sync all
  if (1 == this_image()) then
    read *, nwalkers, nsteps
    lwalkers = nwalkers / num_images()
    do i = 2, num_images()
      lwalkers[i] = lwalkers
      nsteps[i] = nsteps
    enddo
    allocate(xall(nsteps,nwalkers),x2(nsteps))
  end if
  sync all
  allocate(x(nsteps,lwalkers)[*])

  call random_init(this_image())
  x(1,:) = 0
  do i = 1, lwalkers
    do k = 2, nsteps
      call random_number(r)
      if (r < 0.5) then
        x(k,i) = x(k-1,i) + 1;
      else
        x(k,i) = x(k-1,i) - 1;
      endif
    enddo
  enddo
```

Image 1 reads parameters and broadcasts parameters All images initialize local storage

Every image performs random walks

```
  sync all
  if (1 == this_image()) then
    do i = 1, num_images()
      xall(:,lwalkers*(i-1)+1:lwalkers*i) = x(:,:)[i]
    enddo

    do k = 1, nsteps
      x2(k) = sum(xall(k,:)*xall(k,:))/nwalkers;
    enddo

    write xall, x2  out to files for plots.
  end if
  sync all
end program rwalk_p
```

Image 1 collects results from others and performs post processing

Performance?

Coarray and MPI

- Note, on distributed systems, the "get" operation

  ```
  A(:,:) = A(:,:)[p]     ! Copying data on image p to local storage
  ```

  is equivalent to

  ```
  call MPI_Recv(buf,n*n,MPI_REAL,p,tag,comm,status,ierr)
  ```

  Unmarshall data in buf to A

Coarray and MPI

- And the "put" operation

```
A(:,:)[p] = A(:,:)     ! Push data to image p from local storage
```

is equivalent to

Marshall data from A into buf

```
call MPI_Send(buf,n*n,MPI_REAL,p,tag,comm,ierr)
```

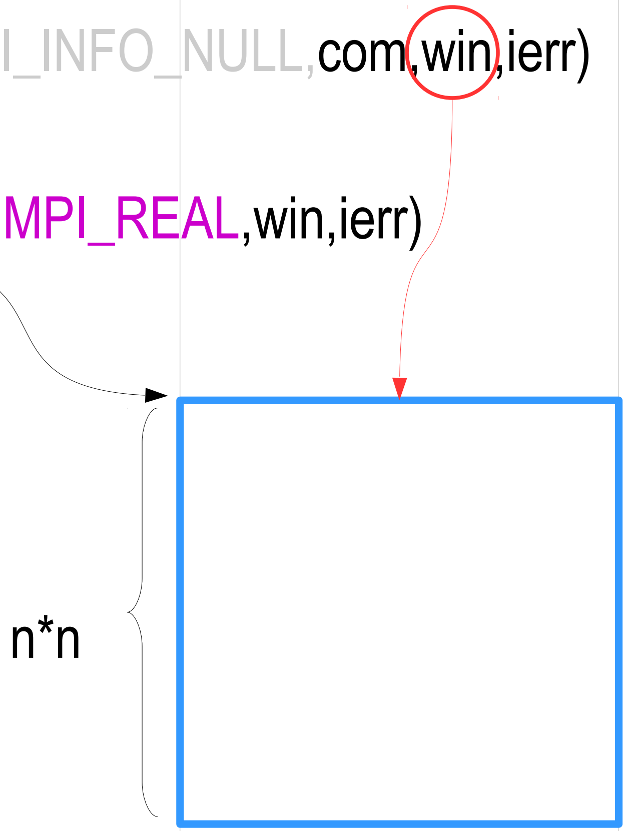
Coarray and MPI

- Technically coarray operations are closely related to ***one sided communication*** (in MPI). This assignment on image other than p

```
    A(:,:)[p] = A(:,:)     ! Push data to image p from local storage
```

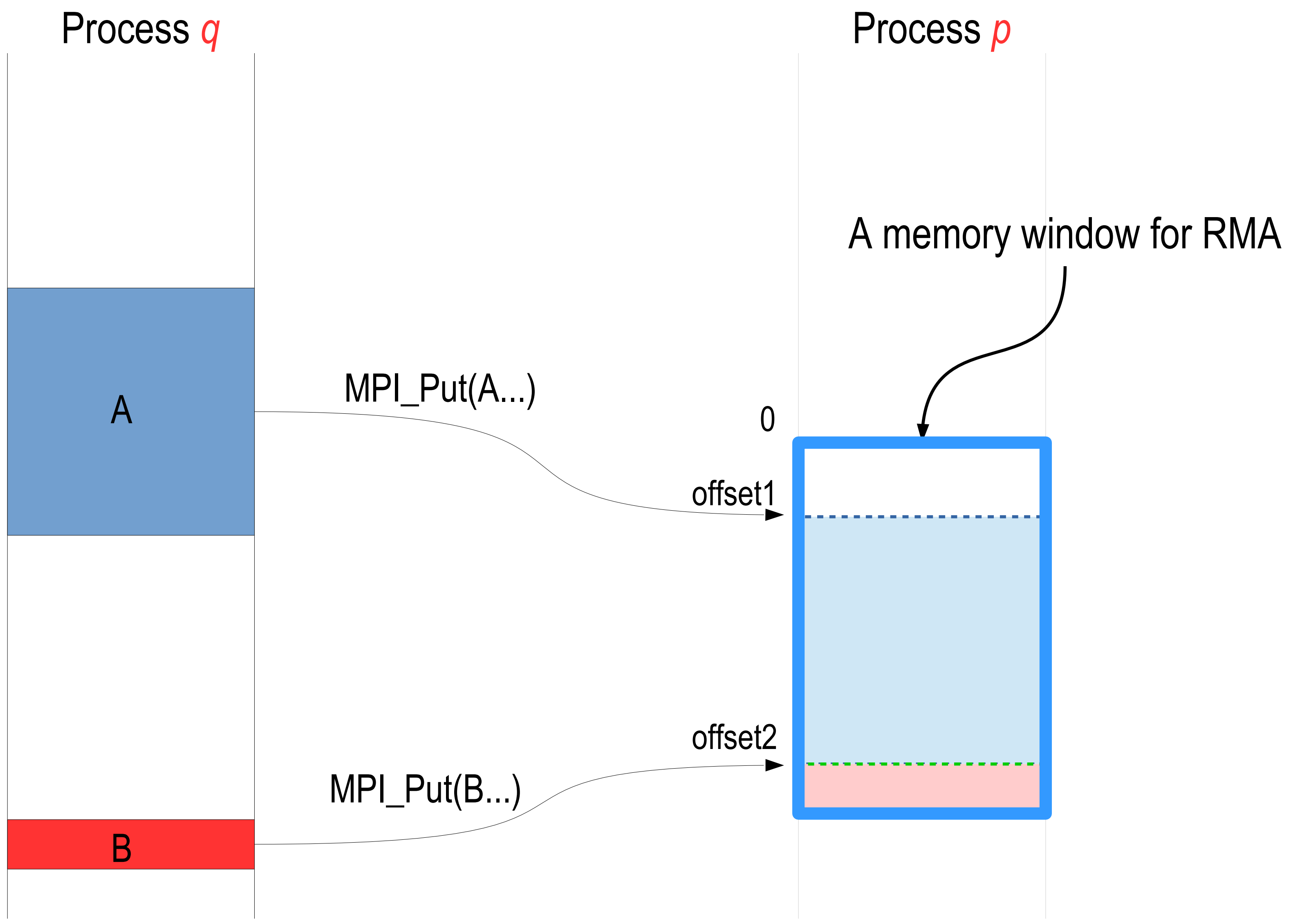
is equivalent to the following

```
    call MPI_Win_create(A,ws,MPI_REAL,MPI_INFO_NULL,com,win,ierr)
    call MPI_Win_fence(0,win,ierr)
    call MPI_Put(A,n*n,MPI_REAL,p,start,n*n,MPI_REAL,win,ierr)
    call MPI_Win_fence(0,win,ierr)
    call MPI_Win_free(win,ierr)
```



A memory window on process *p*

Remote Memory Access (RMA)



Performance Concerns

- In serial code

  A(:,:) = A(:,:) + B(:,:)

  or simply

  A = A + B

  involves two loads and one store operations.
- While the parallel code

  A(:,:)[p] = A(:,:) + B(:,:)

  might involve the use of a temporary storage to hold the result of the RHS operation A + B before a *long haul store* – send data to image p.
- Our recent tests show this operation is more expensive than using native MPI calls directly.

Performance Concerns

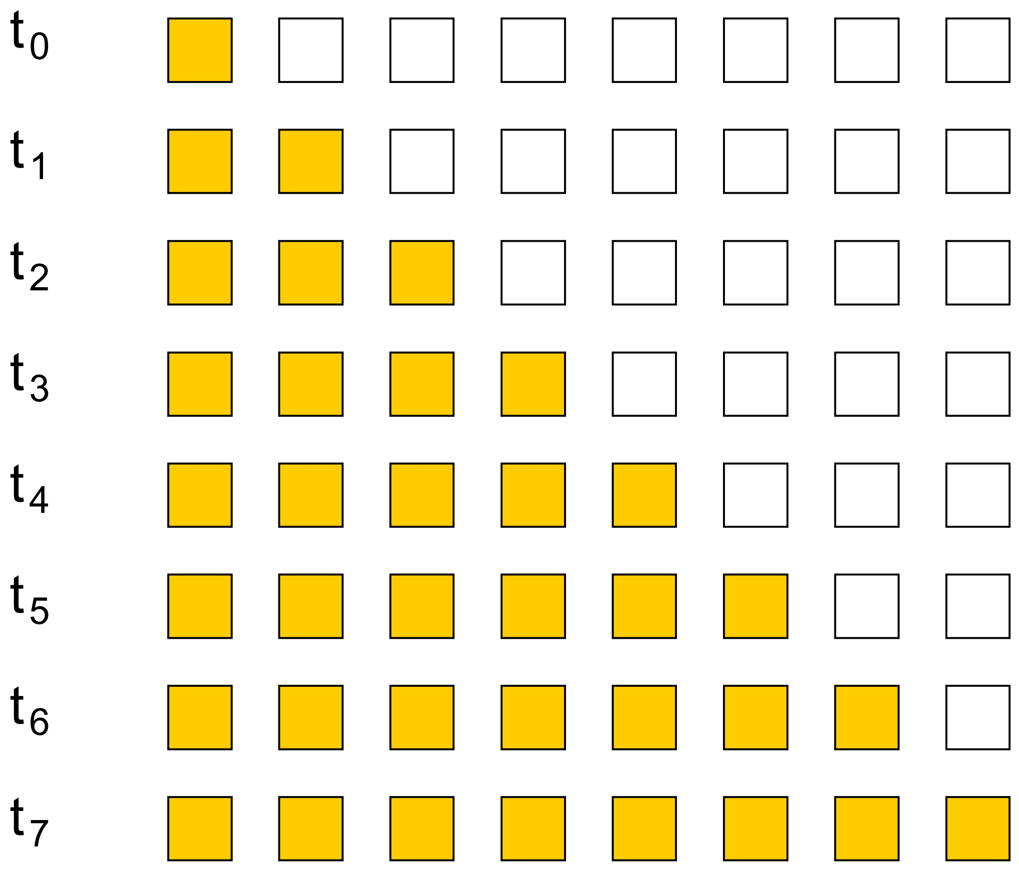
- Any comments on the broadcast operation?

```
do i = 2, num_images()
   z[i] = z
enddo
```

Broadcast: Complexity

SEMINAR SERIES 2015

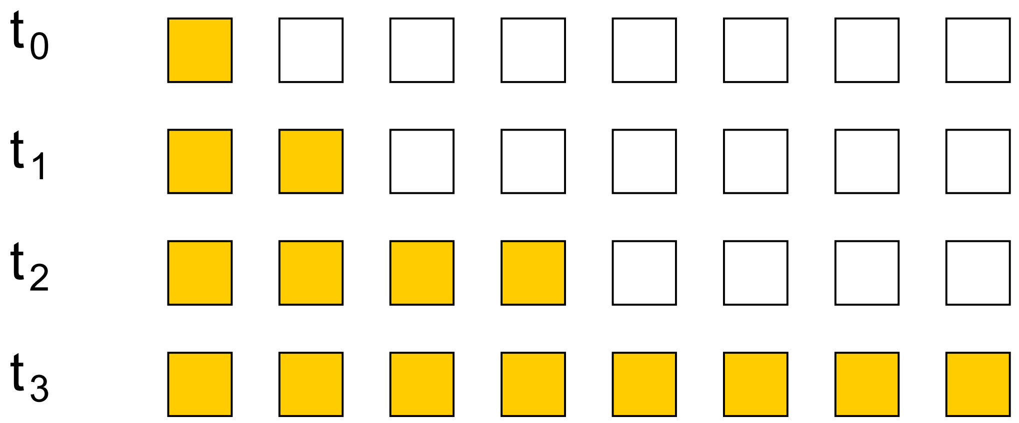
Linear



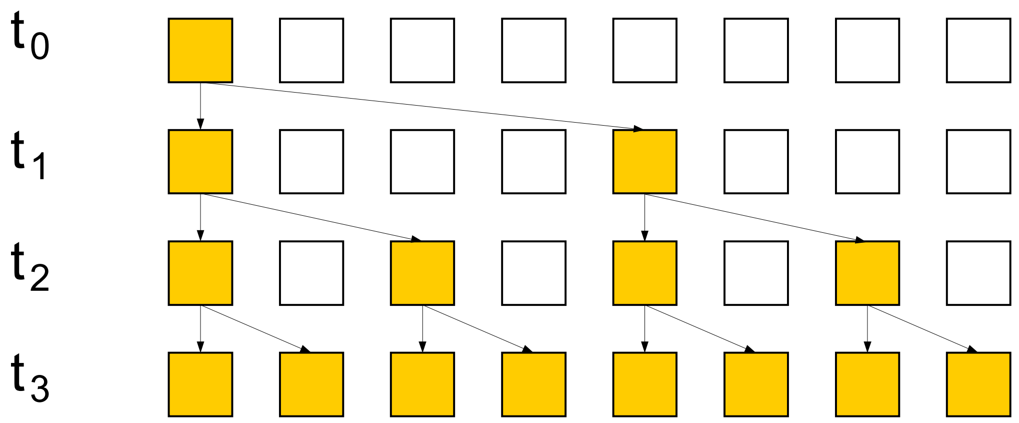
$O(N)$

Broadcast: Complexity

Improved



$O(\log N)$



Summary

Summary

- The SPMD model is assumed, i.e. every image executes the same program.
- The SPMD model assumes coarrays on every image, e.g.

```
real :: a(10000,10000)[*]
integer :: ma[*], na[*]
```

- The SPMD model requires self identification (“this image”) and others, via
  - this_image()
  - num_images()
- The control of work flow is done by the selection logics, e.g.

```
if (1 == this_image()) then
   call manager()
else
   call worker()
endif
```

- Memory coherence is not assured until you want to (e.g. via remote copies)
- Synchronizations

Summary

- Progammable for both shared (multicore) and distributed (cluster) memory environment
- Easy to write high level code
- Expressive
- Productive
  - Easy, takes less time to write
  - Easy to read and maintain
  - Reusable
- Efficient (yet to test)
- Having a promise future of availability and longevity
- Fortran and MATLAB users should consider in particular.

References

[1] Michael Metcalf, John Reid and Malcolm Cohen, ***"Modern Fortran Explained"***, Oxford University Press, New York, 2011.

[2] R. W. Numrich, J. Reid, "Co-array Fortran for parallel programming", ACM SIGPLAN Fortran Forum, Vol.17, Iss. 2, 1998, pp. 1-31.

[3] **JTC1/SC22** – The international standardization subcommittee for programming languages (http://www.open-std.org/jtc1/sc22/).

[4] The Fortran standards committee (http://www.nag.co.uk/sc22wg5/).

[5] William Gropp et al, ***"Using MPI-2"***, The MIT Press, 1999.

[6] Jonathan Dursi, "HPC is dying, and MPI is killing it", his blog, http://www.dursi.ca/.