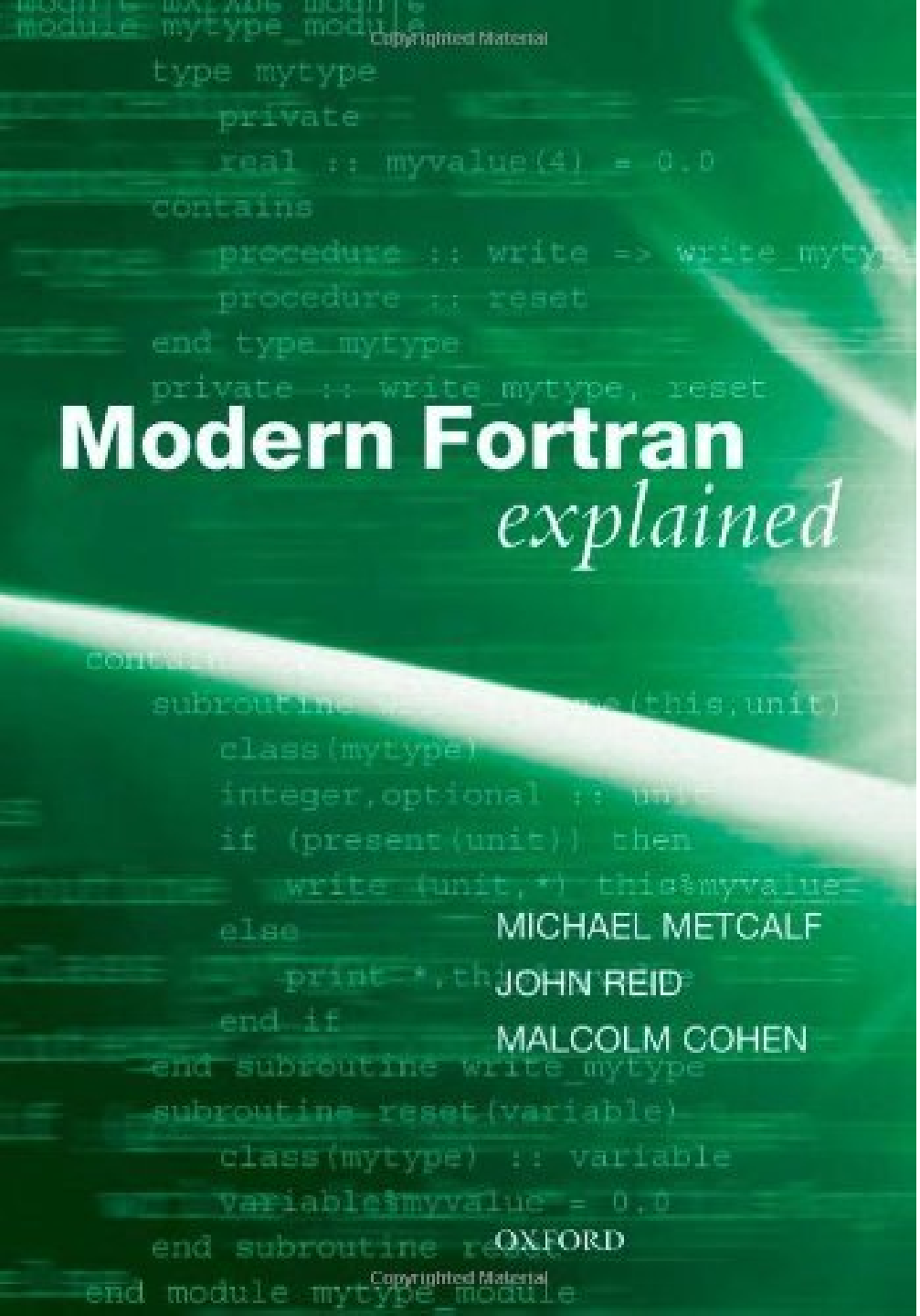
Morden Fortran: Concurrency and parallelism

April 19, 2017

Ge Baolai
SHARCNET
Western University

Outline

- Highlights of some Fortran 2008 enhancement
- Array assignment and concurrency of do loops
- Parallel computing with coarrays



Fortran 2008 Features

- M. Metcalf, J. Reid, M. Cohen, "*Modern Fortran Explained*", Oxford, 2011.
- Fortran 2008 standards draft (latest). You may get it from: http://www.j3-fortran.org/doc/year/10/10-007.pdf
- J. Reid, "The New Features of Fortran 2008": ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1828.pdf

Some Fortran 2008 Enhancement

```
character, parameter:: vowles(*) = [‘a’,’e’,’i’,’o’,’u’] ! Named constant array, size omitted.

real:: a(6), b(100), c(2,3)
a = [1,2,3,4,5,6]                  ! Array assignment, similar to MATLAB.
b(1:6) = a                         ! This implies vectorized operations.
c = reshape(a,[2,3])

a(::2)                             ! Get elements with even indices.
a(5:1:-1)                          ! Traverse elements in reverse order.

a=1; b=2; c=3; d=5                 ! Use of semicolon to separate statements.

do concurrent(i=1:n, j=1:n, i/=j)  ! Do concurrent construct, with condition mask i/=j.
  ... ...
enddo
```

Some Fortran 2008 Enhancement

! New interfaces to trig functions and new hyperbolic functions.

acos(x), asin(x), atan(x), cosh(x), sinh(x), tan(x), tanh(x)

acosh(x), asinh(x), atanh(x)

! New special mathematical functions.

bessel_j0(x), bessel_j1(x), bessel_jn(x)

bessel_y0(x), bessel_y1(x), bessel_yn(x)

erf(x), erfc(x)

gamma(x), log_gamma(x)

! Euclidean norms.

hypot(x,y) $\sqrt{x^2 + y^2}$.

norm2(x[,dim]) $||x||_2$. Using LAPACK routine norm(x,norm_type) is perhaps still the best for performance.

Some Fortran 2008 Enhancement

```
! A handy routine to execute an external command
call execute_command_line( command[, wait, exitstat, cmdstat, cmdmsg] )
```

where

command	The command to be executed.

The following arguments are optional:

wait	Logical, if FALSE, then the command is executed asynchronously (non blocking).
exitstat	If executed synchronously, set to a processor-dependent exit status.
cmdstat	The status of the execution of the command.
cmdmsg	Contains the message from the command.

Some Fortran 2008 Enhancement

```
! Array and single value variable
real:: x(10000)
real:: a(1000,1000)
real:: b(1000,1000)
integer:: m, n
real:: u, v, w
complex, allocatable:: z(:)
```

Some Fortran 2008 Enhancement

```
! Coarrays – globally accessible objects on shared and distributed memory systems.
real:: x(10000)[16]                    ! x accessible on an array of 16 processors.
real:: a(1000,1000)[*]                 ! a accessible on an array of processors.
real:: b(1000,1000)[16,16]             ! b accessible on a 16x16 processor grid
integer:: m[*], n[*]                   ! Or written as: integer, codimension[*]:: m, n
real:: u[*], v[*], w[*]                ! Or written as: real, codimension[*]:: u, v, w
complex, allocatable, codimension[*] :: z(:)

! Typical SIMD programming model
if (this_image() == 1) then
   input data
   do image = 1, num_images()
      u[image] = u ! Send the value of local u to remote u's on all images
   enddo
endif
```

Vectorization & Concurrency

DO loops

```
! Array assignment
a = (/(i,i=0,n)/)

! Array elemental assignment
do i = 0, n
   a(i) = i
enddo
```

- The first assignment has a simple syntax, similar to MATLAB. But can be slower for large n.
- The loop is traditional, can be vectorized by the compiler.

DO loops

```
! Do concurrent with mask
do concurrent(i = 1:n, a(i) > b(i))
   a(i) = a(i) - b(i)*d(i)
   c(i) = c(i) + a(i)
enddo
```

- Simple, clean, tells compiler explicitly no dependencies between iterations.

```
! Equivalent form
do concurrent(i = 1:n)
   if (a(i) > b(i)) then
      a(i) = a(i) - b(i)*d(i)
      c(i) = c(i) + a(i)
   endif
enddo
```

- The traditional form, with if branch embedded inside the loop.

Array Assignment

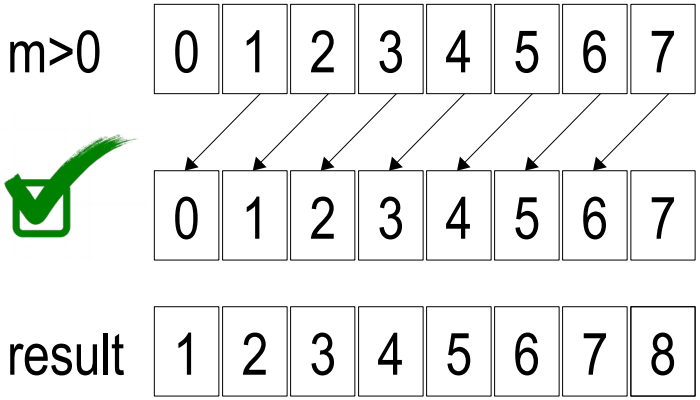
```
real(8):: a(1000000), b(1000000)

! Loop expression
do i = 1,n
   b(i) = a(i)
enddo

! Or in short hand
b(1:n) = a(1:n)

! Or simply
b = a
```

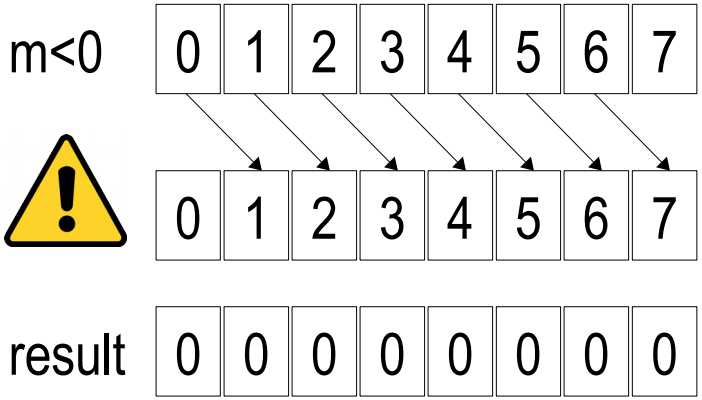
- The compiler can optimize the code in both cases without loops at all.

Array Assignment: Strength and Weakness


```
! Array assignment by copy
a(1:n) = a(1+m:n+m)

! Loop expression
do i = 1,n
   a(i) = a(i+m)
enddo
```

- The array assignment is vectorized and always gives the correct answer.
- The array assignment might be at a cost of overhead – ***use of a temporary store*** on the RHS.
- The loop, however, is error prone. Each iteration has a ***backward dependency*** when m=-1, resulting in wrong answer.





DO loops

We use **gfortran** compile, with option -O3 and -fopt-info-optimized

gfortran array_assignment.f90 -O3 -fopt-info-optimized=array_assignment.optrpt

A vectorization report is written to the file array_assignment.optrpt:

```
array_assignment.f90:17:0: note: Loop 3 distributed: split to 0 loops and 1 library calls.
array_assignment.f90:30:0: note: Loop 5 distributed: split to 0 loops and 1 library calls.
array_assignment.f90:36:0: note: Loop 6 distributed: split to 0 loops and 1 library calls.
array_assignment.f90:44:0: note: Loop 7 distributed: split to 0 loops and 1 library calls.
array_assignment.f90:44:0: note: Loop 8 distributed: split to 0 loops and 1 library calls.
array_assignment.f90:51:0: note: Loop 9 distributed: split to 0 loops and 1 library calls.
array_assignment.f90:53:0: note: loop vectorized
array_assignment.f90:53:0: note: loop versioned for vectorization because of possible aliasing
array_assignment.f90:53:0: note: loop peeled for vectorization to enhance alignment
array_assignment.f90:17:0: note: loop vectorized
array_assignment.f90:17:0: note: loop peeled for vectorization to enhance alignment
array_assignment.f90:53:0: note: loop turned into non-loop; it never loops.
array_assignment.f90:53:0: note: loop with 3 iterations completely unrolled
array_assignment.f90:1:0: note: loop turned into non-loop; it never loops.
array_assignment.f90:1:0: note: loop with 3 iterations completely unrolled
array_assignment.f90:17:0: note: loop turned into non-loop; it never loops
array_assignment.f90:1:0: note: loop turned into non-loop; it never loops.
array_assignment.f90:1:0: note: loop with 10 iterations completely unrolled
```

DO loops

```
call system_clock(count_rate=crate)
call system_clock(count=c1)
do concurrent (i = 1:m)
  c(i) = sqrt(a(i) + b(i))
enddo
call system_clock(count=c2)
dt = 1.0*(c2 - c1)/crate
```

- Using loop

```
call system_clock(count=c1)
c = sqrt(a + b)
call system_clock(count=c2)
dt = 1.0*(c2 - c1)/crate
```

- The equivalent (slightly faster)

DO loops

```
! Using loop
do concurrent(i = 1:n, a(i) > b(i))
   a(i) = a(i) - b(i)*d(i)
   c(i) = c(i) + a(i)
enddo
```

- Using loop

```
! Using logical mask and merge()
logical:: mask(n)
... ...
mask = a > b
a = a - merge(b*d,0.,mask)
c = c + merge(a,0.,mask)
```

- The equivalent

Concurrency, OOP, Some Timings...

```
do concurrent(i = 1:n, a(i) > b(i))
   a(i) = a(i) - b(i)*d(i)
   c(i) = c(i) + a(i)
end do
```

- ~0.58s

```
logical:: mask(n)
mask = a > b
a = a - merge(b*d,0.,mask)
c = c + merge(a,0.,mask)
```

- ~1.53s

```
where (a > b)
   a = a - b*d
   c = c + a
end where
```

- ~3.3s

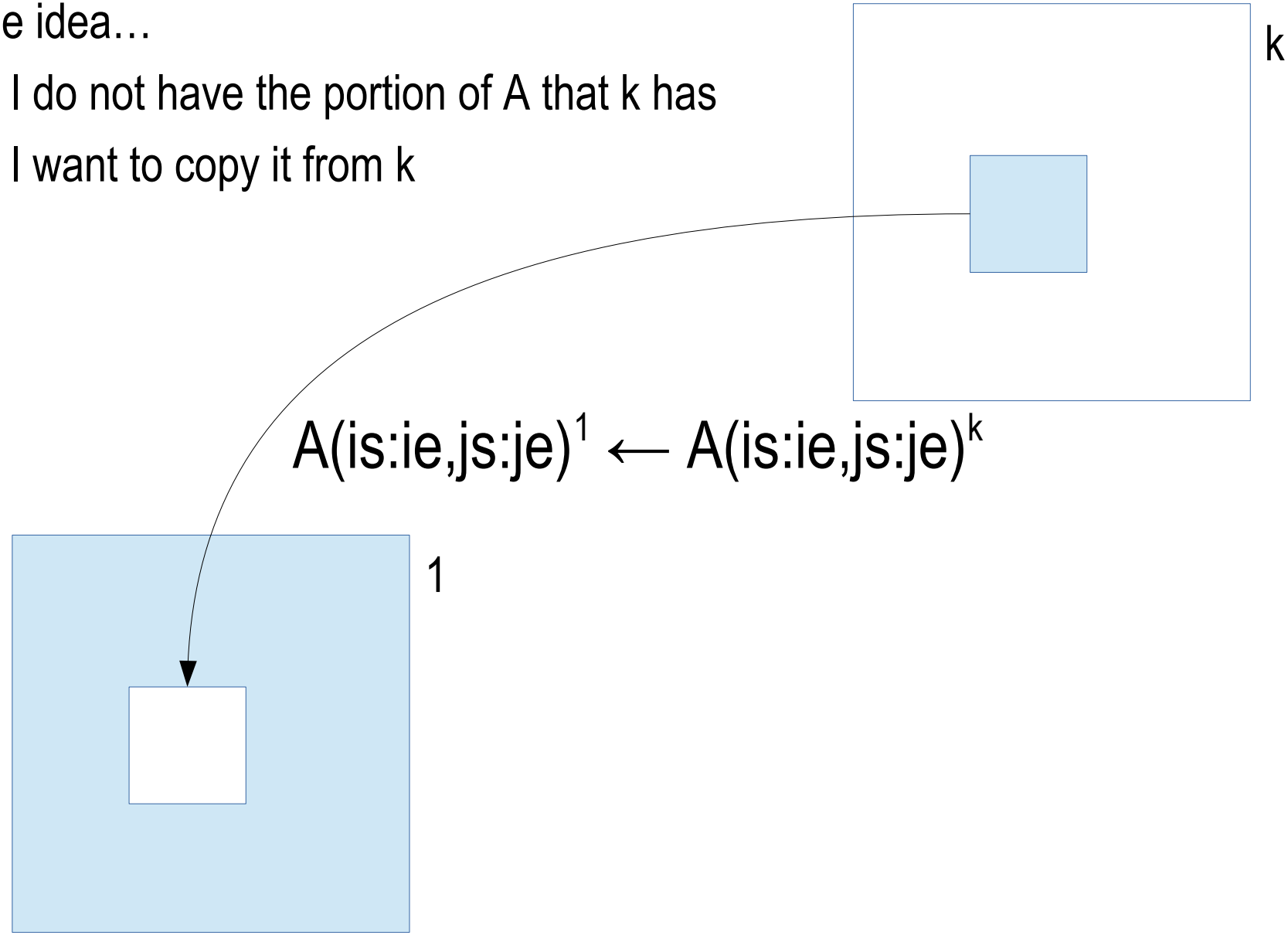
n=100,000,000


Parallel Computing with Coarrays

Coarray

The idea…

- I do not have the portion of A that k has
- I want to copy it from k



Coarray

Use message passing, 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)

  Put 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

  Copy data from local A to the buffer

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

- One must ensure the assembly is correct!

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

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

So here come this notion

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

Coarray

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

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




end program main
```

Coarray

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

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

end program main
```

Coarray

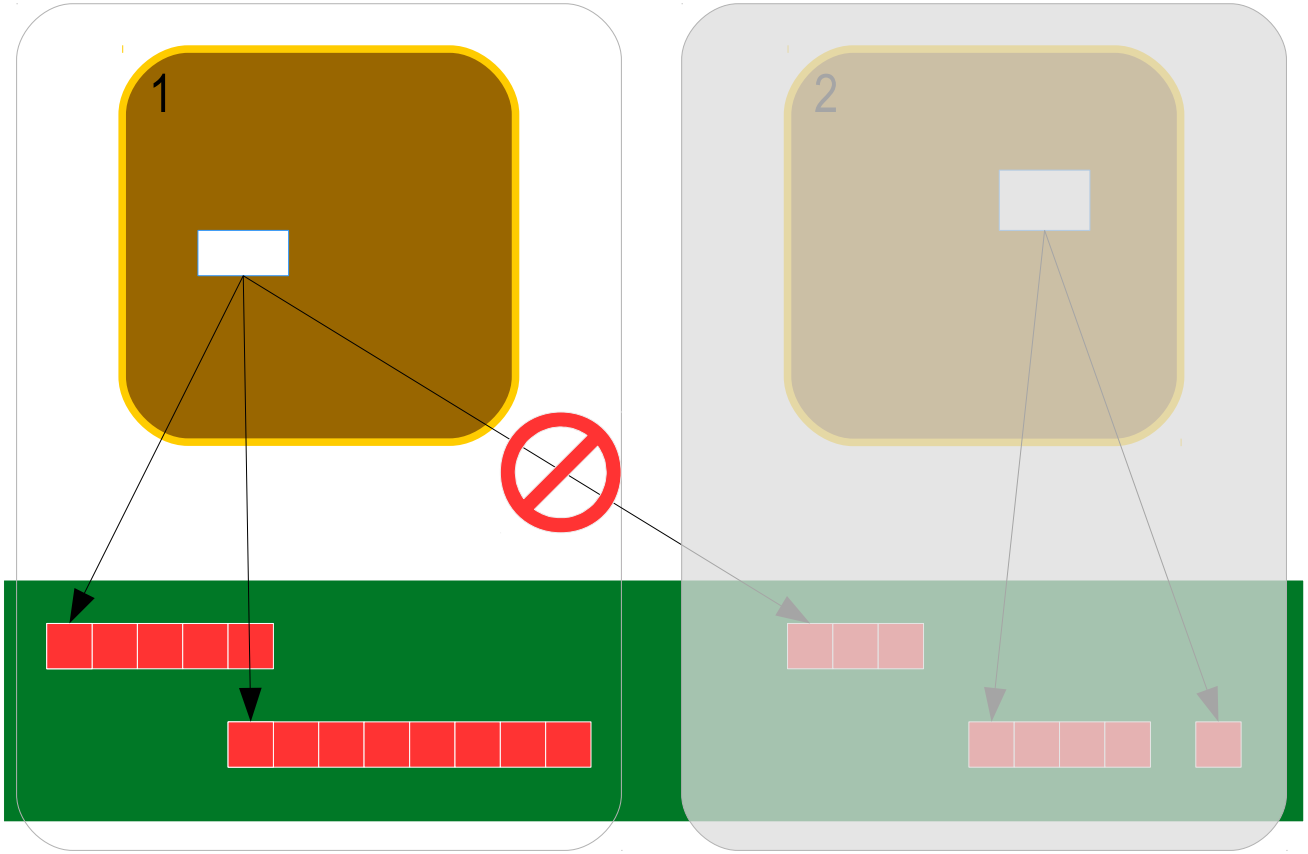
```
program main
   real :: x(10000), u(10000)
   real :: A(1000,1000)[*]
   complex :: y(10000)
   … …
```

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

```
end program main
```

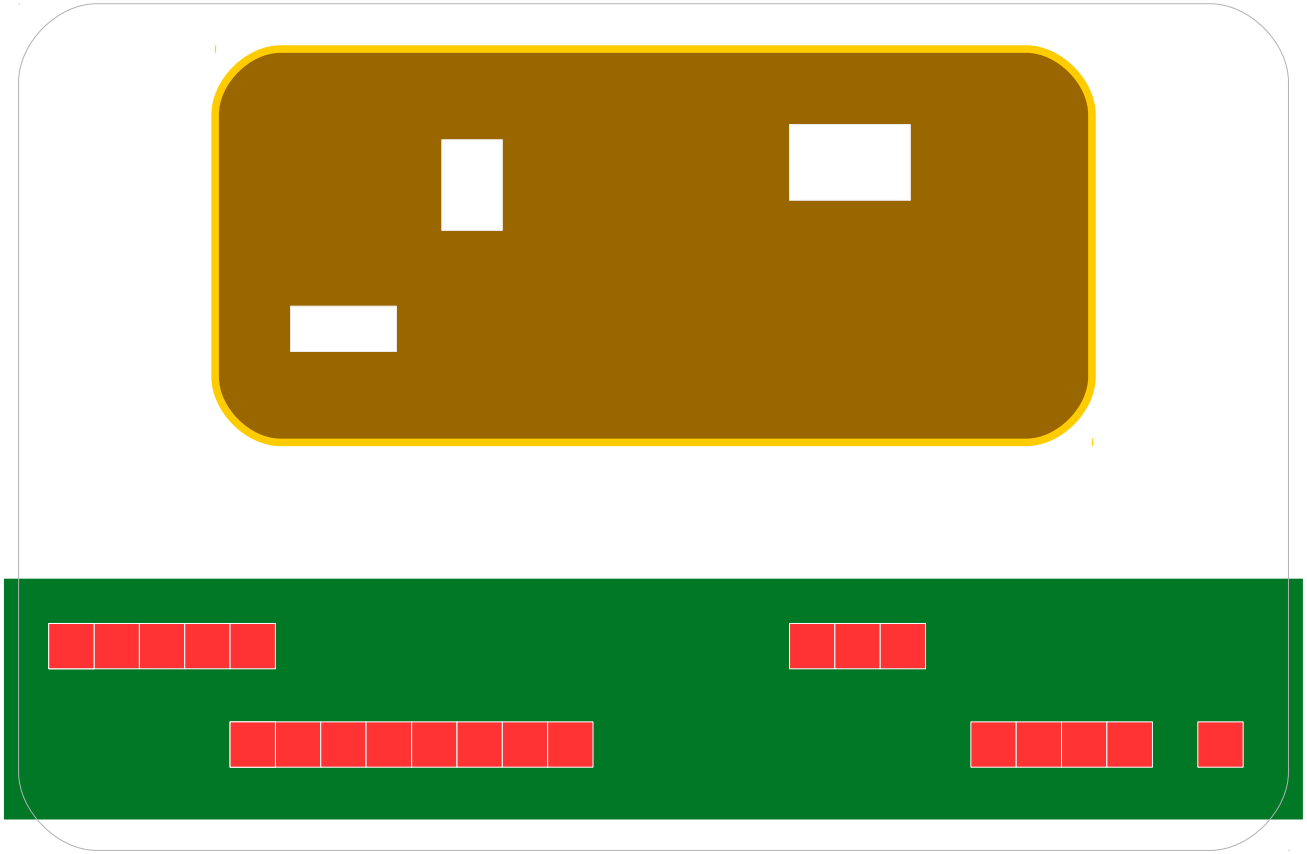
Single Processes

- One process does not see the content of others



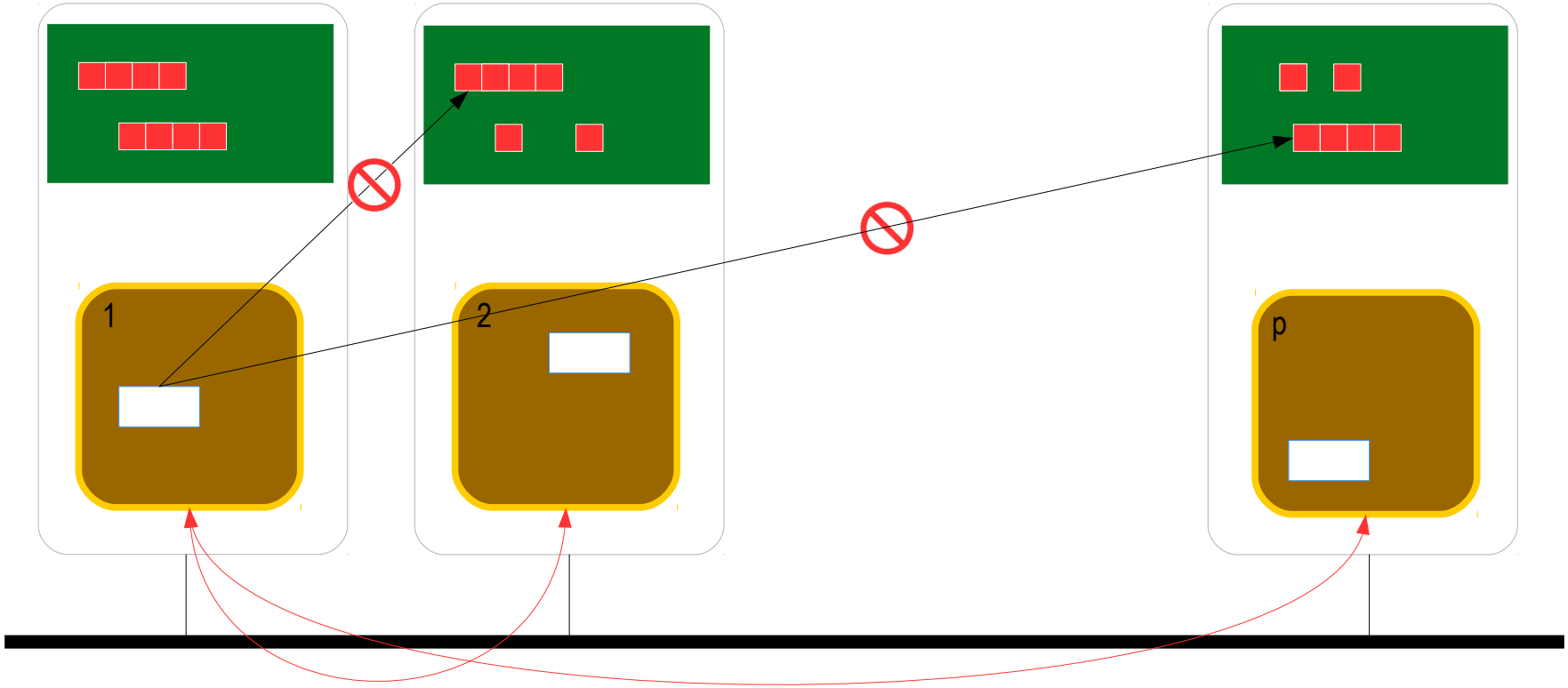
Multithreaded Processes

- Threads within a process see all data within the process



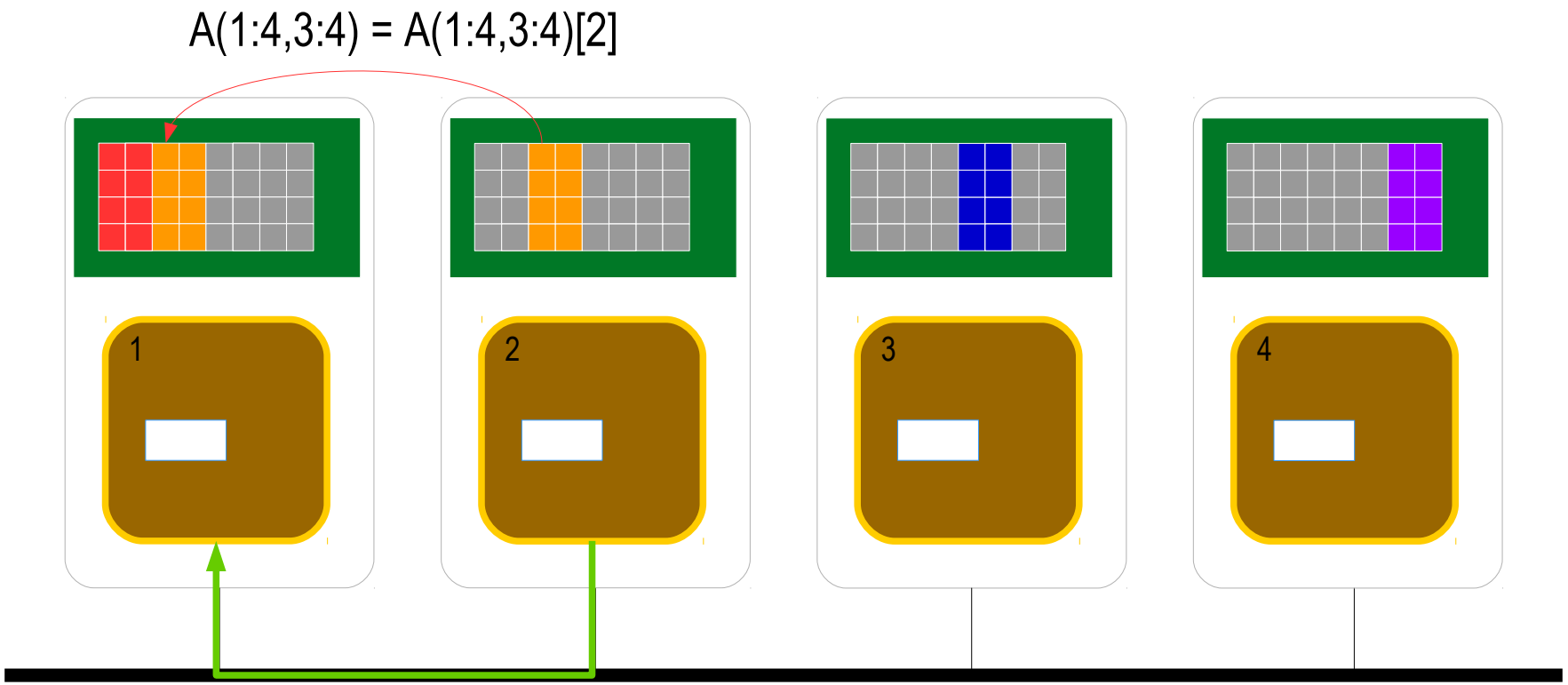
Distributed/Shared Memory - MPI

- 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)



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.



History

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)
- Global arrays, SHMEM

} Partitioned Global Address Spaces (PGAS)

- OpenAcc, OpenMP

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(:)
! 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[*]

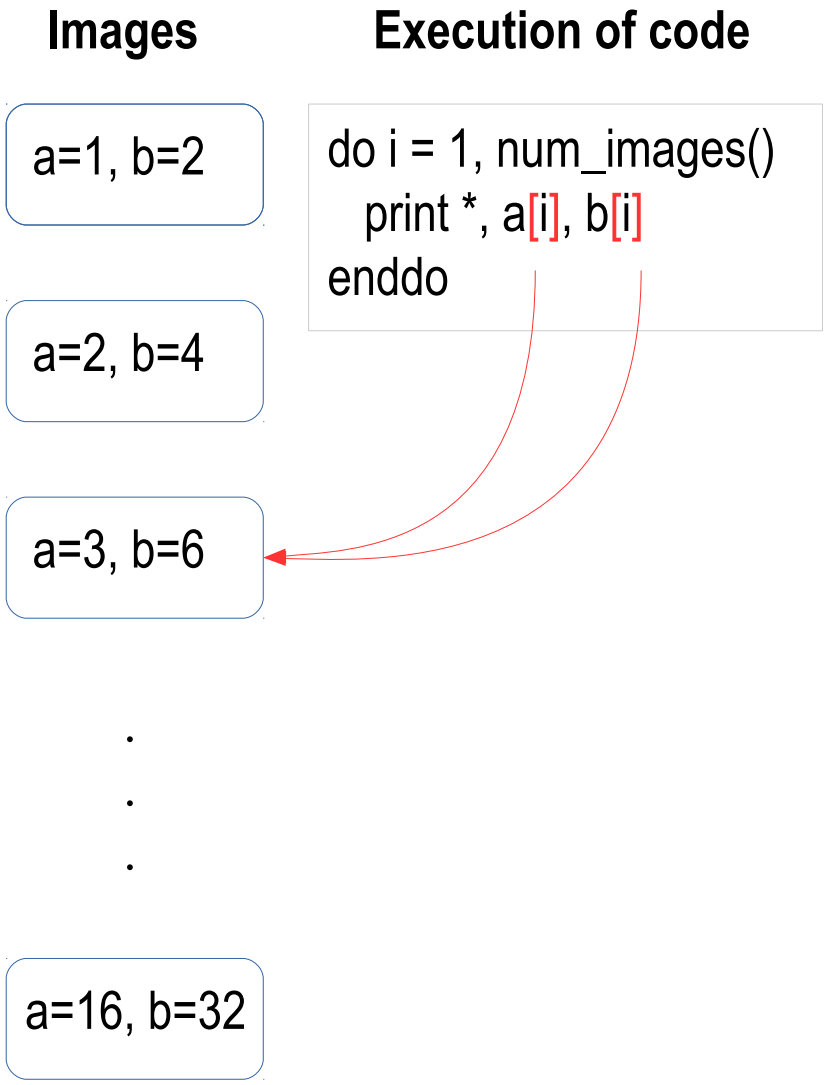
  type(*type2*), allocatable :: u[:]

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



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

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

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

- Note the following 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!

Example: Broadcast


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

  print '("Image",i4,": before: z=",f10.5)', this_image(), z
  sync all
  if (this_image() == 1) then
    read *, 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: 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
```

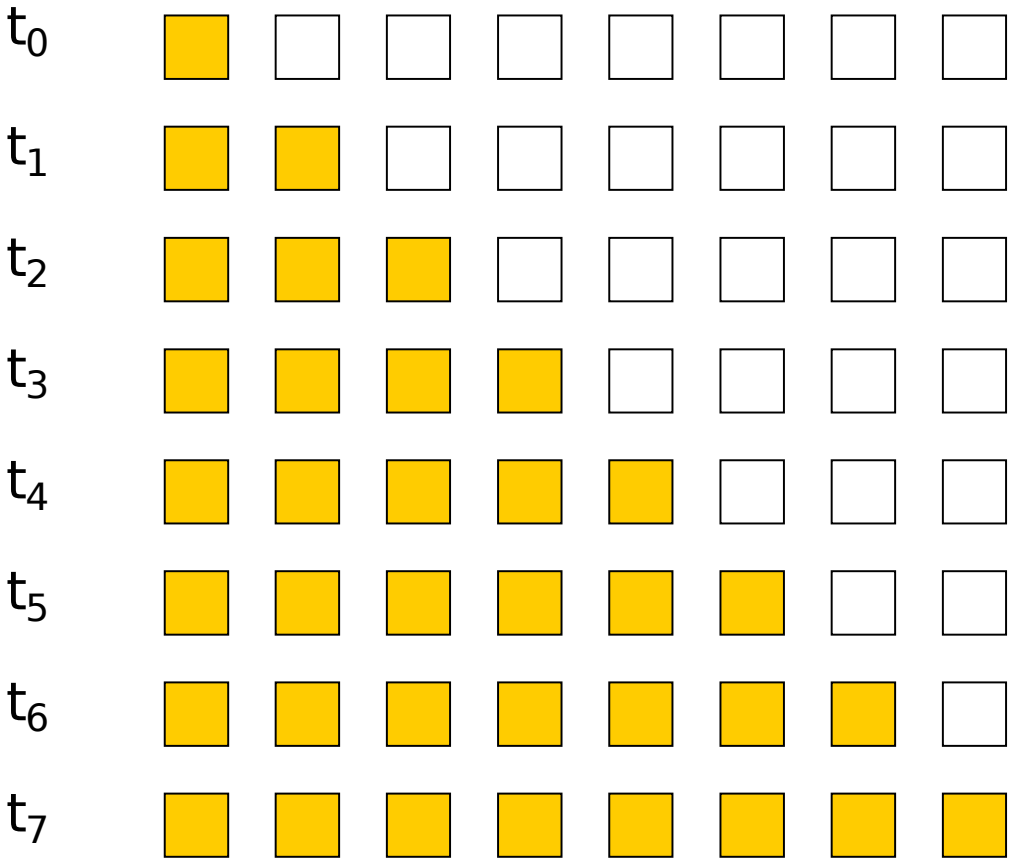
Reflection: Broadcast/Reduction

- Any comments on the broadcast operation?

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

Broadcast: Complexity

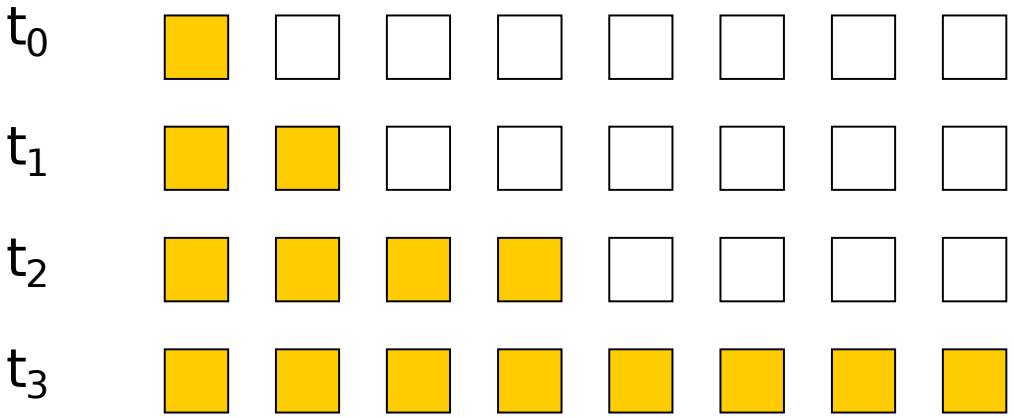
Linear



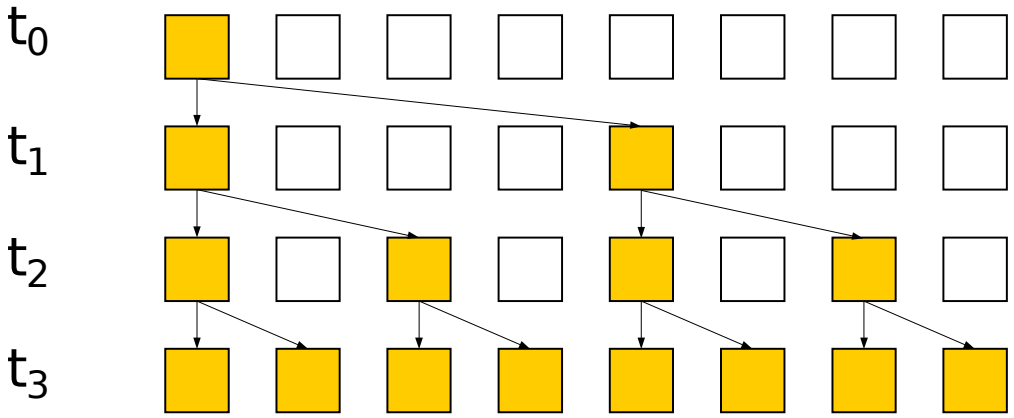
$O(N)$

Broadcast: Complexity

Improved



$O(\log N)$



How: Summary of SPMD

- 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

Compiling Coarray Fortran

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/)
- To compile

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

- To run

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

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 FOR_COARRAY_NUM_IMAGES=8
./mycode
mpirun -n num_procs ./mycode
```

Synchronizations

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
```

Critical Section

- Multiple images try to update the object, but only one at a time

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

Problem 1: Assembling Lenna

The problem

- Each process (image) posses a small (square) portion of Lenna.
- To have the main process collect portions of Lenna and assemble them into the whole image.

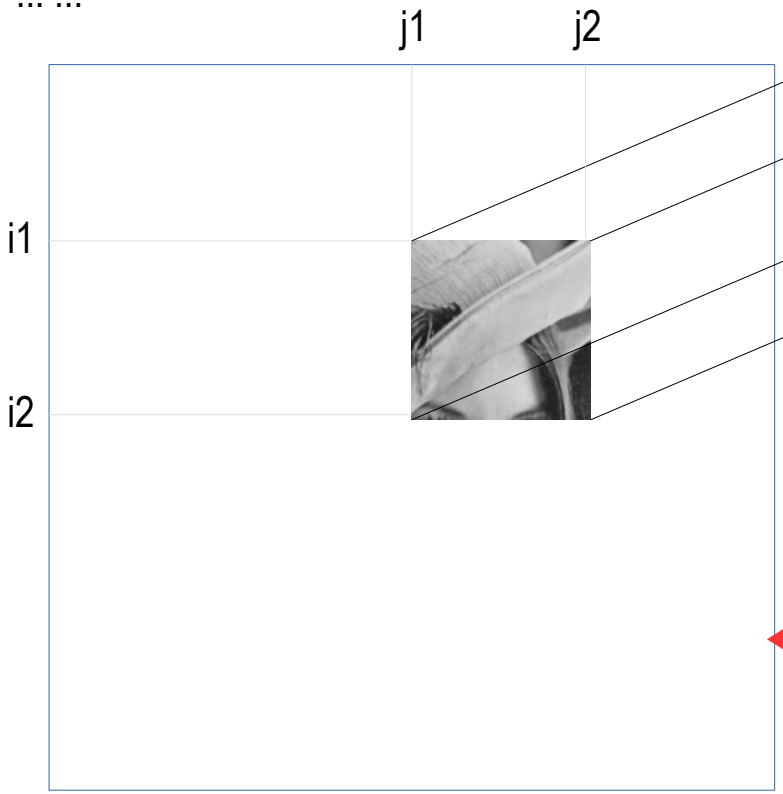
The implementation

- Use **pic(:,:)** for the whole and **pic_p(:,:)[ ]** for local portion.
- The main process loops over processes, collects the portion from each process and assembles it in the whole array.

```
if (this_image() == 1) then
  do i = 1, num_images()
    ... ...
    pic(i1:i2,j1:j2) = pic_p[i] ! Fetch the portion from image i
  enddo
endif
```

Problem 1: Assembling Lenna


```
! Import the portion, e.g.
allocate(pic_p(nx_p,ny_p)[*])
do i = 1, ny_p
   read(10,*) pic_p(i,:)
enddo

! Perform some tasks
... ...
```



```
if (this_image() == 1) then
   do i = 1, num_images()
      ... ...
      pic(i1:i2,j1:j2) = pic_p[i] ! Fetch the portion from image i
   enddo
endif
```

Summary

Summary – Choosing A Language

- Ubiquitous
  - Multicores, even on your laptop
  - Clusters in your department, institution
  - Supercomputers
- Expressive
- Productive
  - Easy, takes less time to write
  - Easy to read and maintain
  - Reusable
- Efficient
- Having a promise future of availability and longevity
- Supported by tools

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