Fortran for Scientific Computing

May 29, 2017

Ge Baolai
SHARCNET
Western University

Outline

- Highlights of the language
- Coding 1 – tryout
- *Coffee break*
- Coding 2 – 1D diffusion equation
- Parallel computing with coarrays
- Coding Test – *Lenna*

SUMMERSCHOOL 2017

```
    ! Put pieces together

    if (this_image() == 1) then
       pic(1:nx,1:ny) = pic_p
       do i = 2, num_images()
          row = (i-1) / q
          col = mod((i-1),q)
          i1 = row*ny_p + 1
          i2 = i1 + ny_p[i] - 1
          j1 = col*nx_p + 1
          j2 = j1 + nx_p[i] - 1
          pic(i1:i2,j1:j2) = pic_p(:,:)[i]
       enddo

       img = trim(base_name)//'.'//'pgm'

       open(11,file=img,status='unknown')
       write(11,'("P2")')
       write(11,'(i3,i4)') nx, ny
       write(11,'(i3)') depth
       write(fmt,'(i3)') nx-1
       fmt = '(i3,'//trim(fmt)//'i4)'

       do i = 1, ny
          write(11,fmt) pic(i,:)
       enddo
       close(11)
    endif
```

Before getting started...

You may get the course materials from the URL below:

- To your laptop using browser

  http://www.sharcnet.ca/~bge/ss2017/fortran_ss2017.zip

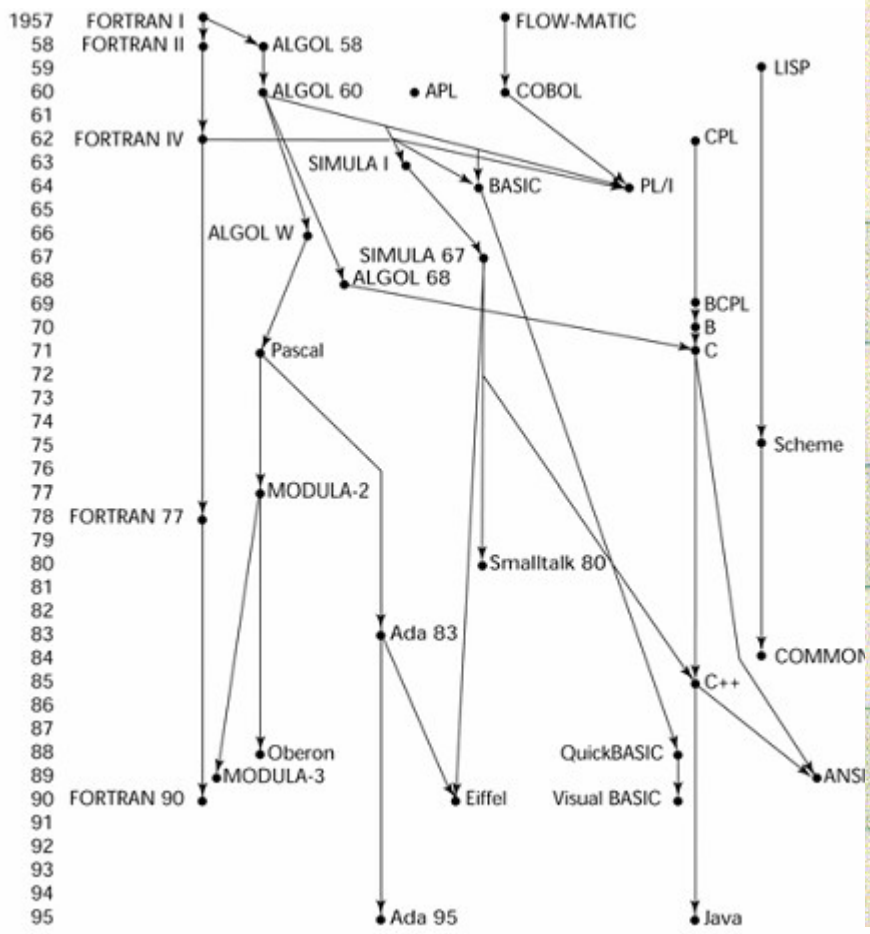
- Or, while on a SHARCNET system, use wget

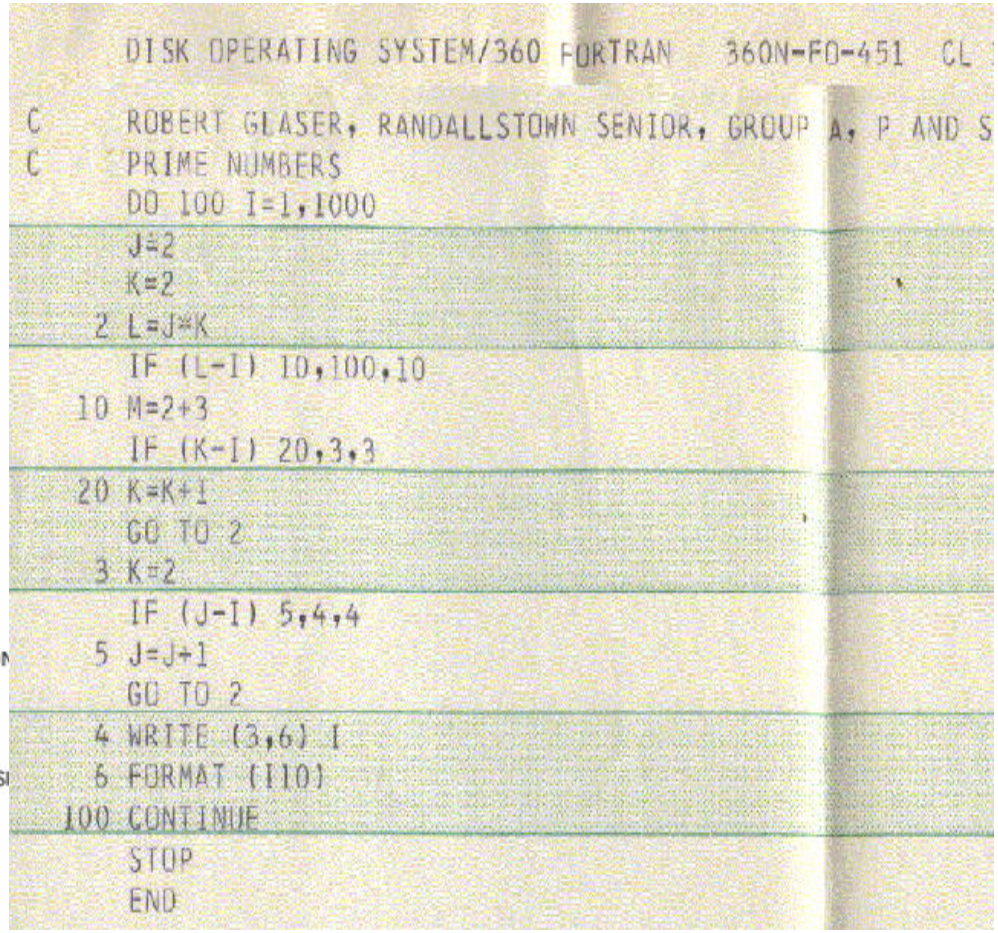
  wget http://www.sharcnet.ca/~bge/ss2017/fortran_ss2017.zip

- Create a directory ss2017 (optional)
- Uncompress the file under Unix environment (in folder ss2017 if you have one)
- In folder fortran, source the setup.sh file

  . ./setup

Language Basics





```
      DISK OPERATING SYSTEM/360 FORTRAN   360N-FO-451  CL
C     ROBERT GLASER, RANDALLSTOWN SENIOR, GROUP A, P AND S
C     PRIME NUMBERS
      DO 100 I=1,1000
      J=2
      K=2
    2 L=J*K
      IF (L-I) 10,100,10
   10 M=2+3
      IF (K-I) 20,3,3
   20 K=K+1
      GO TO 2
    3 K=2
      IF (J-I) 5,4,4
    5 J=J+1
      GO TO 2
    4 WRITE (3,6) I
    6 FORMAT (I10)
  100 CONTINUE
      STOP
      END
```

A brief history...

Milestones

- 1954-1957 The birth of FORTRAN by a team led by J. W. Backus at IBM on IBM 704
- FORTRAN II, III, IV and 66
- FORTRAN 77 – a milestone: block structure introduced in the wave of *structured programming*.
- Fortran 90 – a milestone: free format, lots of new features that rejuvenated the language. Fortran more often used instead of FORTRAN. Many Fortran programs we see today were written in the mixed styles of FORTRAN 77 and Fortran 90.
- Fortran 95 – a minor revision, influenced by the *object-oriented programming* concept.
- Fortran 2003 – a minor revision.
- Fortran 2008 – a minor revision: Coarray added for parallel processing.
- Fortran 2015 – to come out in 2016.

History

Old FORTRAN (fixed format)

```
1234567 SOURCE STARTS COLUMN 7
      PROGRAM ARRAY
      INTEGER I, J, M, N
      REAL*8 A(100,100)
C
      DO 10 I=1,100
   10 B(I)=I
C
C COMMENTS START IN COLUMN 1
      DO 10 J=1,N
      DO 11 I=1,M
      A(I,J)=I*J
   11 CONTINUE
   10 CONTINUE
C
      DO 20 I=1,100
      A(I,100)=B(I)
   20 CONTINUE
      PRINT *, B
      STOP
      END
```

Modern Format (free format)

```
! Comments start with an !
program array
   integer :: i, j, m, n
   real(8) :: a(100,100), b(100)

   b = [(i,i=1,100)]

   ! Comment can start anywhere
   do concurrent(j=1:n,i=1:n)
      a(i,j) = i * j
   end do

   ! Block assignment like MATLAB
   a(:,100) = b

   print *, b
end program array
```

Why Fortran

- Designed for scientific computing.
- Simple, takes less time to write (why do you use MATLAB?)
- Performance.
- Widely used and supported.
- Potentially easier for writing parallel code.

Data Types

- **Integer**.
- **Real** – single or double precision.
- **Complex** – single (8 bytes) and double precision (16 bytes).
- **Character** – character and strings.
- **Logical** – e.g. **.FALSE.** and **.TRUE.**
- **Constant** – literal constants, e.g. 123, 3.1415926…
- **Derived types** – data structures, along with numeric ***polymorphism***, lead to ***object-oriented*** programming
- **Pointer** – useful for array referencing and swapping.

Examples

```
integer(2) :: n2        ! Short
integer(4) :: n         ! Integer
integer(8) :: ln        ! Long
real(4) :: mass         ! Single precision
real(8), dimension(10000) :: x, y
logical :: isneighour = .false.
! Coarrays, with [ ], globally accessible
integer :: num_points[*]
real, allocatable :: a(:,:)[:], b(:,:)[:]

type particle
  real :: m
  real :: x, y, z
  real :: u, v, w
end type particle
```

Data Types

- **selected_real_kind**(*N*), e.g.

  real(kind=selected_real_kind(15)) :: v15

  real(kind=selected_real_kind(7)) :: v7

- **selected_int_kind**(*N*), e.g. to hold large integers up to 38 digits

  integer(kind=selected_int_kind(38)) :: n1, n2, n3

Data Types: Literal Constants


```
1
0
-9.78654321
+11
1.02e-4
-1.0d+3

! Complex constants
z = (-1,1)
z = (0.123,.99e-2)
z = (3.0d0,-4.0d0)

! Specified precision in N decimal digits
integer, parameter:: k6=selected_int_kind(6)
-12345_k6
+2_k6
```

```
! Non base-10 literal constants
b'01100110'        ! Base 2
o'076543'          ! Octal
z'10fa'            ! Hexadecimal
```

Intrinsic Functions/Procedures

Elemental Mathematical

abs(x) ! |x|

aimag(z) ! Im(z)

real(z) ! Re(z)

int(a), nint(a)

conjg(z) ! Conjugate

cmplx(x,y) ! x+i*y

aint(a) ! Nearest w.n. towards 0

anint(a) ! Nearest w.n.

ceiling(a), floor(a)

sqrt(x), exp(x), log(x), log10(x)

sin, cos, tan,… ! Trig functions

max/min(a, b[, ...])

mod(a,p), modulo(a,p)

sign(a,b) $|a|\,\mathrm{sgn}(b)$

… …

Unified name for all types

Special Functions

erf(x)

erfc(x)

gamma(x)

log_gamma(x)

bessel_j0(x) ! Bessel, 1st kind, order 0

bessel_j1(x) ! Bessel, 1st kind, order 1

bessel_jn(n,x) ! Bessel, 1st kind, order n

bessel_y0(x) ! Bessel, 2nd kind, order 0

bessel_y1(x) ! Bessel, 2nd kind, order 1

bessel_yn(n,x) ! Bessel, 2nd kind, order n

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

norm2(x) $\|x\|_2$

norm2(x,dim)

Intrinsic Functions/Procedures

Inquiry Functions

```
digits(x)     ! Significant digits in machine rep
epsilon(x)
huge(x)
tiny(x)
maxexpoent(x), minexponent(x)
precision(x)          ! Decimal precision
radix(x)              ! Base
range(x)              ! Decimal exponent

size(a[,dim=1[,...]]) ! Length of an array
shape(a)              ! Dimensions of a
reshape(a,shape)      ! Same as MATLAB
```

Intrinsic Functions/Procedures

Array Functions

dot_product(x,y)	$x \cdot y$
matmul(A,B)	AB
transpose(A)	A^T
sum(a)	$\sum a_i$
product(a)	$\prod a_i$
maxval(a), minval(a)	
maxloc(a), minloc(a)	
size(a[,dim=1[,...]])	
shape(a)	! Dimensions of a
reshape(a,shape)	! Same as MATLAB
cshift(a,p), eoshift(a,p)	! Shift elements by p
pack(dest,mask,src)	
unpack(src,mask,dest)	
merg(tsrc,fsrc,mask)	

Character, String Functions

len(s)	! Length of string s
trim(s)	! Remove trailing blanks
adjustl(s)	! Leading blanks deleted
adjustr(s)	! Trailing blanks deleted
repeat(s,n)	! Repeat s n times
index(s, t[, back])	! Starting position of t
scan(s, set[, back])	! Position of a char in set
verify(s, set[, back])	! 0 if each char in s is in
	! set; or the position of a
	! char not in set
achar(i)	! ASCII char
iachar(c)	! Position in ASCII

No one can remember all of them. Always keep a reference handy.

Intrinsic Functions/Procedures

Time Functions

```
! Equivalent to gettimeofday()
call date_and_time(
  date,
  time,
  zone,
  value
)

! Measures elapsed time, mostly used
call system_clock(
  count,
  count_rate,
  count_max
)

! Less used
call cpu_time(time)
```

Random Numbers

```
! Seeds are in an array, get its size
call random_seed(size = lseed)

! Create an array for seeds
allocate(seed(lseed))

! Create seeds with clock ticks
call system_clock(count=clock)
seed = clock + 37 * [(i - 1, i = 1, lseed)]

! Plant the seeds
call random_seed(put=seed)

! Generate a sequence of RN's
call random_number(r)
```

Basics: Expressions and Statements

Scalars

```
x + y
x - y
x * y
a/b
x**y                    ! x^y
n = 13.4                ! n = 13
i = 13/2                ! i = 6

real :: a, b
complex :: z

z = cmplx(a, b)         ! convert to complex
z = (-3.3, 4.6)         ! or do this way
a = z                   ! a holds Re(z)
c = (-3.3, 4)           ! c holds -3.3
```

Structure

```
type interval           ! Define a new type
  real :: lo
  real :: up
end type interval
type(interval) :: w, u, v
real :: a, b

w = interval(-1.0, 1.0) ! Initialize a structure
a = w % lo
b = w % up

u = w                   ! Structure assignment
v = 2*w – u             ! Not yet defined
```

Basics: Expressions and Assignments

Scalar, Array Operations

```
pi = 3.1415926
c = 2.0*pi*r
a = b * c + d / e
r = sqrt(x**2 + y**2)

integer :: m = 17, n = 5
real :: q, a
q = m / n
a = q * n              ! What's in a?
```

Character, Strings

```
character(len=5) :: word1
character(len=5) :: word2
character(256) :: grtg_msg
```

String concatenation

```
word1 = 'Hello'
word2 = 'world'
grtg_msg = word1//','//word2//'!'
```

Trimming off trailing blanks

```
trim(grtg_msg)
```

Basics: Expressions and Assignment (cont'd)

Objects of Derived Type

```
type particle
  real :: m
  real :: x, y
  real :: u, v
end type particle

type(particle) :: p, q

p = particle(0.2, -1.0, 3.5, 0.5, 2.7)
q = p

q%x = q%x – x0
q%y = q%y – y0
```

Overloading Operators

```
! Define an interface
interface operator (+)
  function rational_add(x,y)
    type(rational) :: rational_add
    type(rational) :: x, y
  end function rational_add
end interface

! Define the function
function rational_add(x,y)
  type(rational) :: rational_add
  type(rational) :: x, y
  … …
end function rational_add
```

Basics: Expressions and Statement

Arrays

```
A = [1, 2, 3, -1, 4.5]
A = B

! Traditional loop operations
do j = 1, 1000
  do i = 1, 1000
    a(i,j) = b(i,j) + c(i,j)
  end do
end do

! Whole array operations
a = b + c

y(1:n) = x(1:n)
c(1,:) = a(:,1)
```

Pointers

```
real, target, allocatable :: ws_u(:,:), ws_v(:,:)
real, pointer, dimension(:) :: u, v, tmp

! Allocate spaces for arrays
allocate(ws_u(n,n),ws_v(n,n))

! Make pointers point to arrays
u => ws_u; v => ws_v

! Use pointers as if arrays
uxx = (u(i-1,j) – 2.0*u(i,j) + u(i+1,j))/(dx*dx)
uyy = (u(i,j-1) – 2.0*u(i,j) + u(i,j+1))/(dy*dy)

! Swap pointers instead of arrays
tmp => u; u => v; v => tmp
```

Basics: Control Constructs

DO loops

```
do i = 1, n
  y(i) = f(x(i))
end do

do k = 1, 10000, 2
  do something
end do

do j = 1, n
  do i = 1, m
    a(i,j) = mat_setval(i,j)
  end do
end do
```

DO [WHILE] loops

```
eps = 1.0
do while (eps + 1.0 /= 1.0)
  eps = eps / 2
end do

do
  do something
  if (abs(error) <= tol) exit
end do

do
  read *, x
  if (x < 0) cycle
  exit
end do
```

Concurrency and Parallelism

FORALL

```
forall (i=1:m, j=1:n, y(i,j) /= 0.)
  a(j,i) = 1.0/y(i,j)
end forall
```

DO CONCURRENT

```
do concurrent (i=1:m, j=1:n)
  a(i,j) = a(i,j) + alpha*b(i,j)
enddo

do concurrent (i=1:m, j=1:n, i/=j)
  … …
enddo
```

Coarray

- A parallel programming paradigm without explicit interprocess communication calls, e.g. via MPI.
- Data objects globally accessible amongst processes without explicit data transfer operations, e.g. MPI_Send(), MPI_Recv(), etc.
- Uses one-sided communication model, with “fetch” and “push” to get and put data respectively.
- Supports parallel processing on both shared and distributed memory architectures.
- In favour of thinking of algorithms than implementation tediousness.

Basics: Control Constructs (cont'd)

IF condition statement

The relational operators

.lt.	or	<	less than
.gt	or	>	greater than
.le.	or	<=	less than or equal
.ge.	or	>=	greater than or equal
.eq.	or	==	equal
.ne.	or	/=	not equal

Logical expressions

.not.	Negation
.and.	Logical intersection
.or.	Logical union

IF..THEN..ELSE IF..ELSE..ENDIF

```
if (x == y) then
  do something
end if

if (x == y1) then
  do case1
else if (x == y2) then
  do case2
else if (x == y3) then
  do case3
else
  default action
end if
```

! Also allow arrays

Coding Exercise 1 – *Tryout*

Basics: Control Constructs (cont'd)

SELECT..CASE

```
select case expr
case val1
   process case1
case val2
   process case2
…
case default
   process default case
end select
```

Example

```
! Select on individual values
select case j
case 1
   call sub1
case 2
   call sub2
end select

! Select on a range
select case x
case (:-1)              ! All <= -1
   call sub1
case (1:5)              ! 1,2,3,4,5
   call sub2
…
end select
```

Basics: Array Comparison and Mask

ANY, ALL

```
if (any(a > b)) then
   print *, 'An elem in A > an elem in B'
endif

if (all(a > b)) then
   print *, 'All elem in A > elems in B'
else if (all(b > a)) then
   print *, 'All elem in B > elems in A'
else
   print *, 'Set A and B overlap'
endif
```

WHERE [..*ELSEWHERE*..*ENDWHERE*]

```
where (b < 0) a = b
print *, 'where (b<0) a=b:'
print '(10f5.0)', (a(i), i = 1,10)

where (b /= 0)
   a = a/b
elsewhere
   a = 0
endwhere
```

Basics: Array Comparison and Mask

Mask

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

- If branches can be avoid.
- Some compilers might be able to better job.

Basics: Array Constructs

MATLAB

```
v = [-1, 0, 2 5, 14, 3.5, -0.02];

n = 1000;
x = 1:n;

y = cos(x)
```

Fortran

```
v = [-1, 0, 2 5, 14, 3.5, -0.02]
v = (/-1, 0, 2, 5, 14, 3.5, -0.02/)

n = 1000
x = [(i,i=1,n)]          ! or x = (/(i,i=1,n)/)

y = cos(x)               ! x, y are arrays
```

Basics: Array Constructs (cont'd)

MATLAB

```
v = 1:n;
a = reshape(v, n1, n2);

% It's in column order
   1   5   9  13
   2   6  10  14
   3   7  11  15
   4   8  12  16
```

Fortran

```
v = [i=1,n]
a = reshape(v, [n1,n2])

! It's in row order
   1   2   3   4
   5   6   7   8
   9  10  11  12
  13  14  15  16
```

Basics: Array Inquiry

MATLAB

```
asize = size(a);
n1 = asize(1);
n2 = asize(2);
```

Fortran

```
asize = shape(a)
n1 = asize(1)
n2 = asize(2)

nrows = size(a,dim=1)
ncols = size(a,dim=2)
num_elems = size(a)

x_len = size(x)
```

Basics: Array Operations

MATLAB

```
a = zeros(100,100);
b = zeros(100);

n = input("Enter n (n <= 100): ");

a(:,:) = 1.0;
b(:) = 2.0;
c = a(1,:) ./ b;

a(::2)   % Get even index elements.
a(5:1:-1) % Traverse in reverse order.
```

Fortran

```
real, dimension(100,100) :: a
real, dimension(100) :: b, c

print *, 'Enter n (n <= 100):'
read *, n

a = 1.0
b = 2.0
c = a(1,:) / b

a(::2) ! Get event index elements.
a(5:1:-1) ! Traverse in reverse order.
```

Basics: Array Operations

MATLAB

```
a = [1,2,3,4,5,6,7,8,9,10]

a(3)
% N/A
% N/A
a(2:5)
% N/A
% N/A
% N/A
a(:)
```

Fortran

```
a = [1,2,3,4,5,6,7,8,9,10]

print '(10i5)', a(3)     ! 3
print '(10i5)', a(4:)    ! 4 5 6 7 8 9 10
print '(10i5)', a(:6)    ! 1 2 3 4 5 6
print '(10i5)', a(2:5)   ! 2 3 4 5
print '(10i5)', a(::3)   ! 1 4 7 10
print '(10i5)', a(2:4:2) ! 2 4
print '(10i5)', a(1::2)  ! 1 3 5 7 9
print '(10i5)', a(:)     ! 1 2 3 4 5 6 7 8 9 10
```

Format

a([istart]:[iend][:incr])

Basics: Intrinsic Array Operations

New functions

- dot_product(a, b)

$$\langle a, b \rangle$$

- product(a)

$$\prod_{i=1}^{n} a_i$$

- sum(a)

$$\sum_{i=1}^{n} a_i$$

- maxcal(a)/minval(a)

Common intrinsic apply to arrays

- SIN, COS, etc apply to array arguments as well.

Coding Exercise 2 – *Diffusion Equation*

Coding Exercise – *1D Diffusion Equation*

Solve

$$\frac{\partial u}{\partial t} = a\frac{\partial^2 u}{\partial x^2}$$

with initial and boundary conditions

$$u(x,0) = f(x),$$
$$u(-L,t) = u(L,t) = 0.$$

Numerical solution:

Using a grid with time step Δt chosen properly and $\Delta x = 2L/(N-1)$

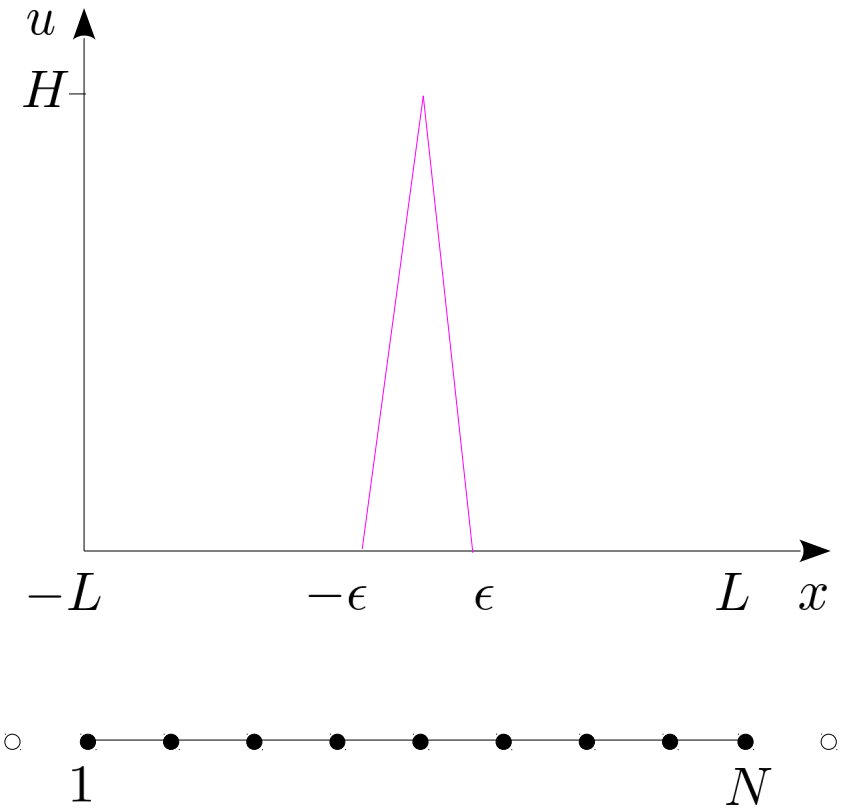
$$x_i = \Delta x(i-1),\ i = 1,\ldots,N,$$
$$t_n = \Delta t n,\ n = 1,\ldots$$

Denote the approximation

$$U_i^n \approx u(x_i, t_n)$$

Exercise: Set

$$f(x) = \begin{cases} H - \frac{|x|}{\epsilon} & -\epsilon \le x \le \epsilon, \\ 0 & \text{otherwise.} \end{cases}$$



Coding Exercise

- Apply (explicit scheme) finite difference approximation to obtain

$$\frac{U_i^{n+1} - U_i^n}{\Delta t} = a\left(\frac{U_{i-1}^n - 2U_i^n + U_{i+1}^n}{\Delta x^2}\right)$$

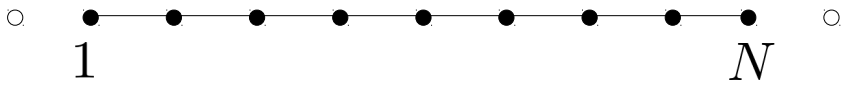
- Or, in an iterative form, for $i = 2, \ldots, N-1, \;\; n = 0, \ldots$

$$U_i^{n+1} = \left(1 - 2\frac{a\Delta t}{\Delta x^2}\right) U_i^n + \frac{a\Delta t}{\Delta x^2} U_{i-1}^n + \frac{a\Delta t}{\Delta x^2} U_{i+1}^n$$

or simply noted

$$U_i^{n+1} = (1 - 2\lambda)U_i^n + \lambda U_{i-1}^n + \lambda U_{i+1}^n$$

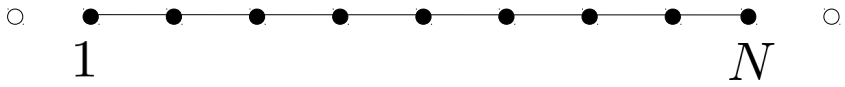
- Compute all U_i^{n+1} for $i = 1, \ldots, N, \;\; n = 0, \ldots$
- Generate graph of U_i^{n+1} every 50 or so time steps.



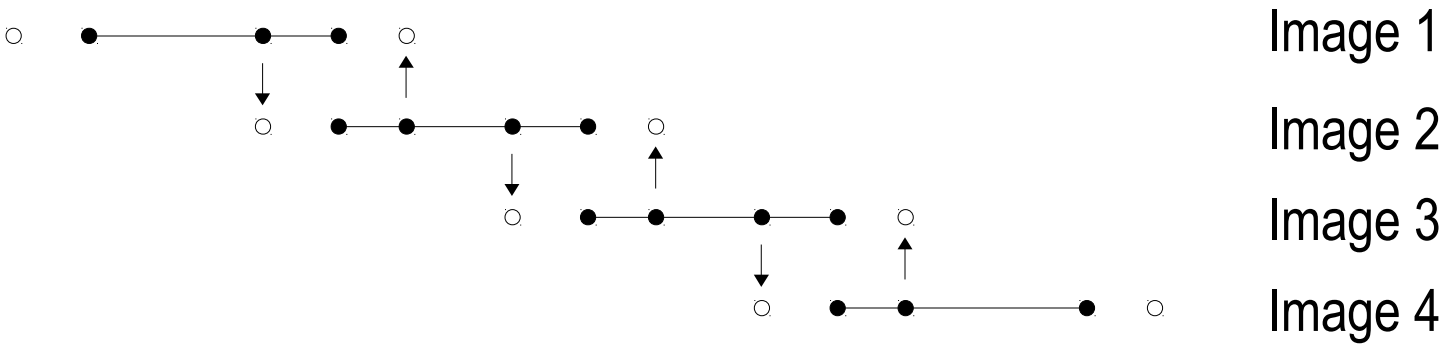
Homework: Modify the 1d diffusion code for parallel processing.

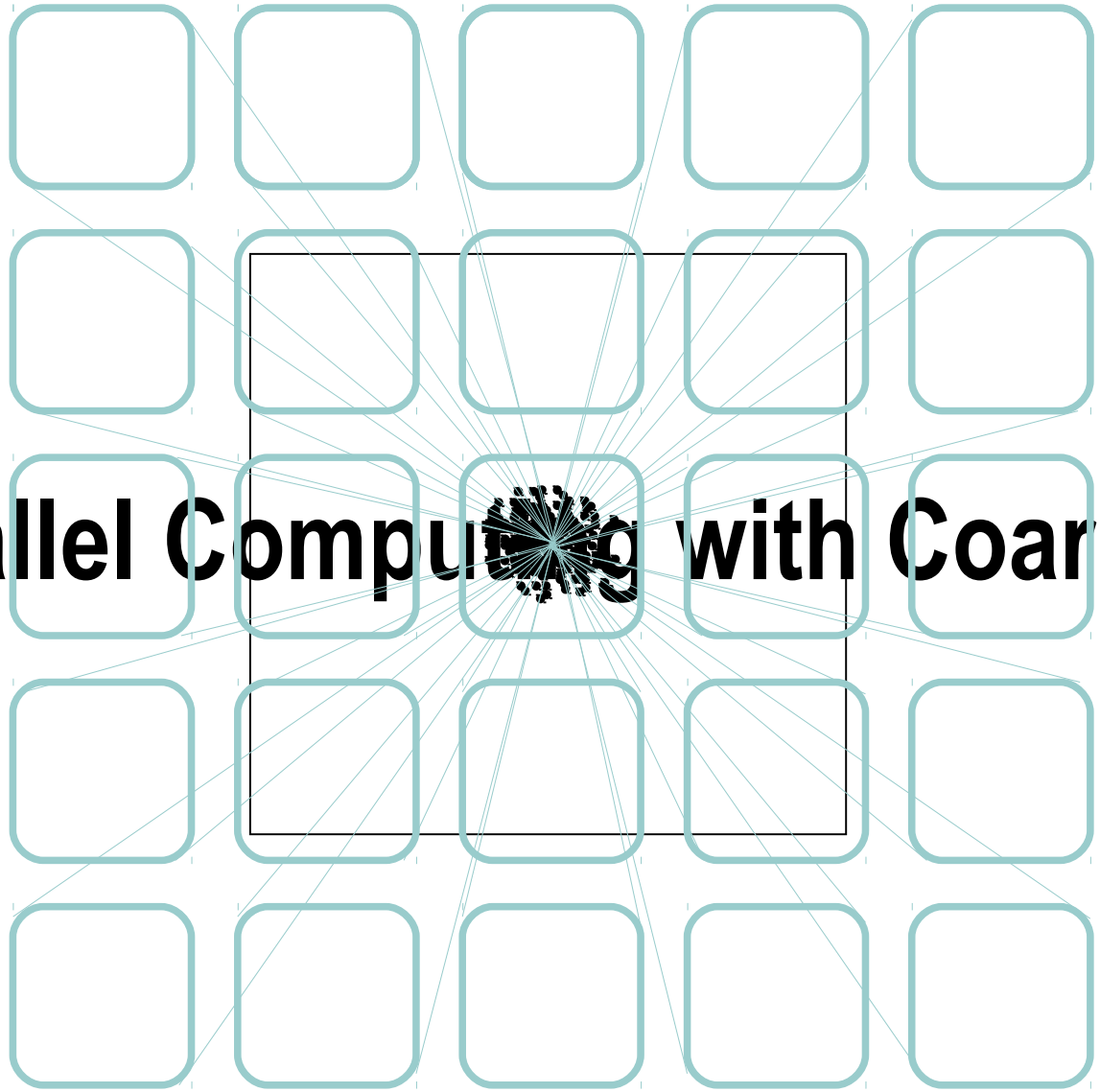
Hint: Divide the interval [0,L] into p sub-intervals. Assign each to a process (image). Each works on its own, image 1 collects final results. See the slides on coarray for implementation details.

Serial: Sweep through all points



Parallel: Each image sweeps through all points on its own portion. End points need values from its immediate neighbours.

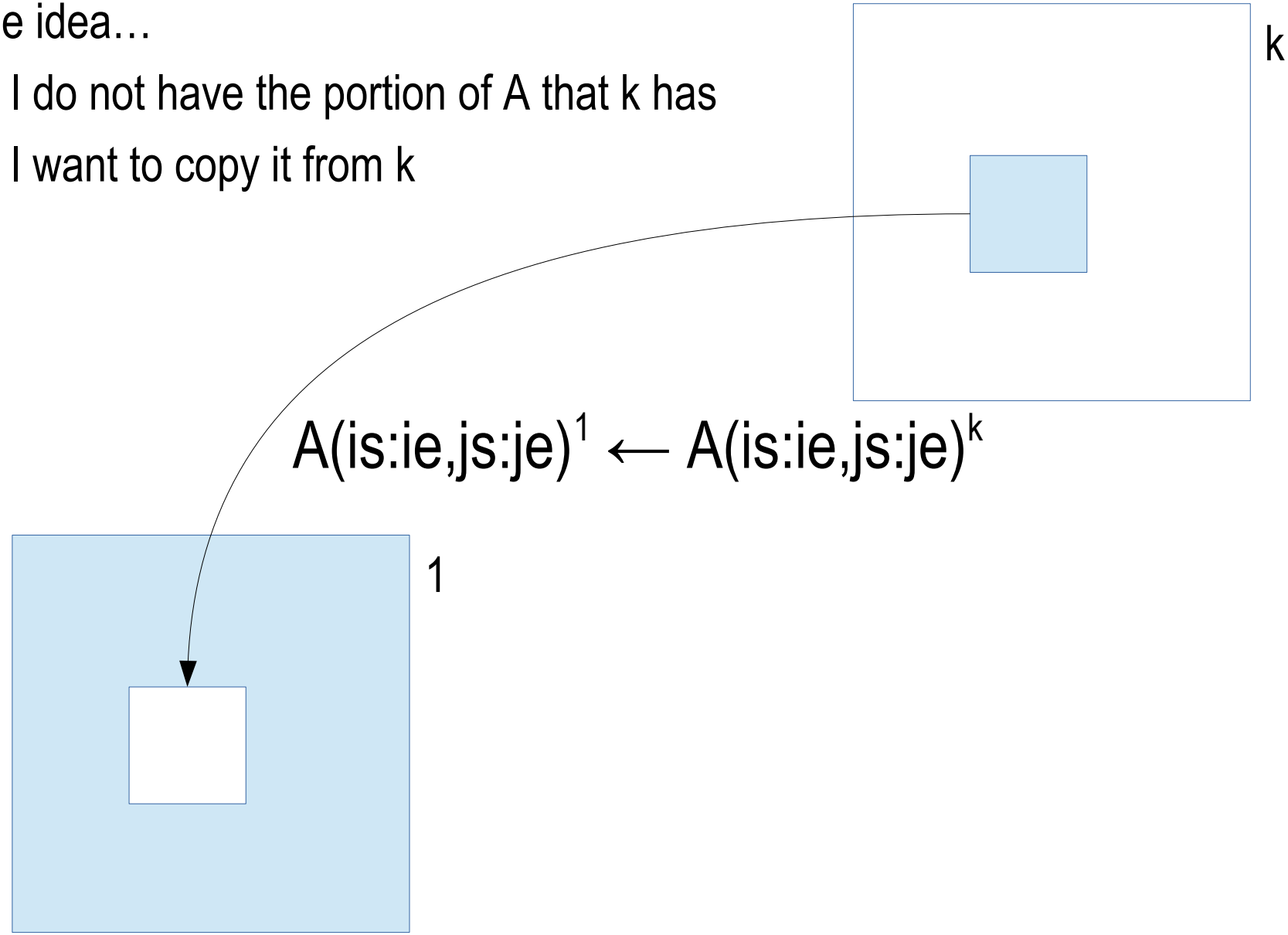




Parallel Computing with Coarrays

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
```

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[*]

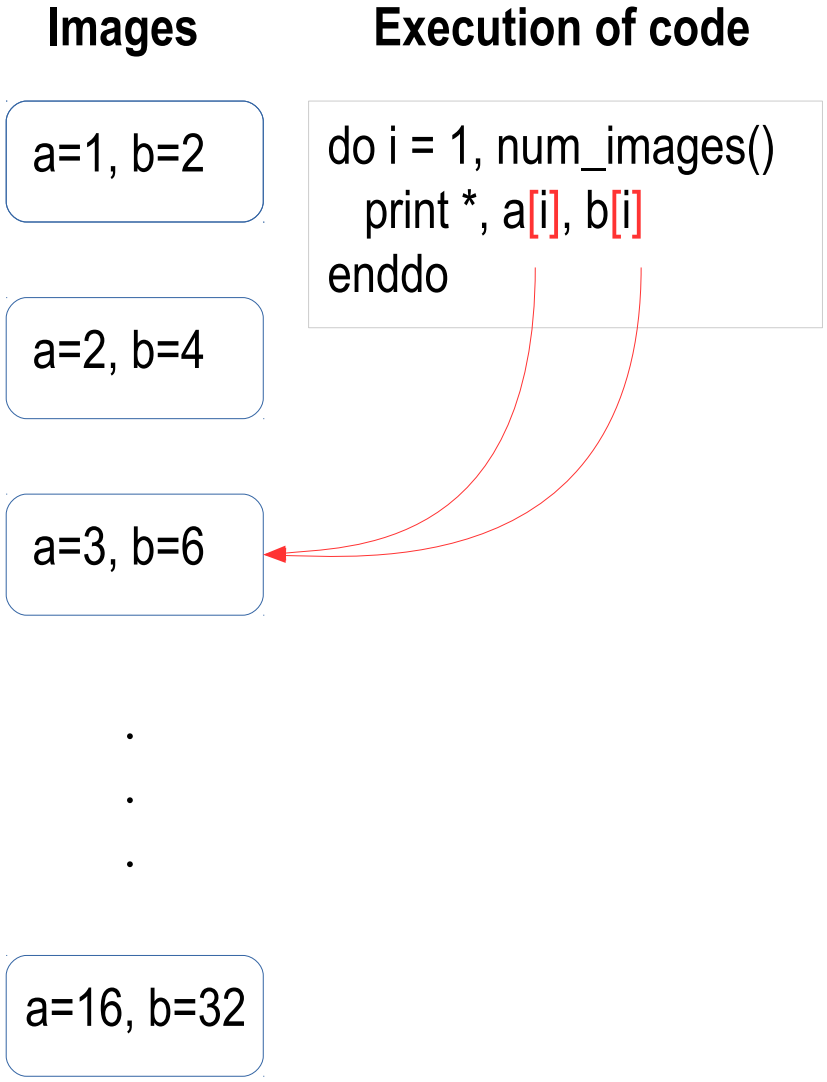
  type(*type2*), allocatable :: u[:]

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
```

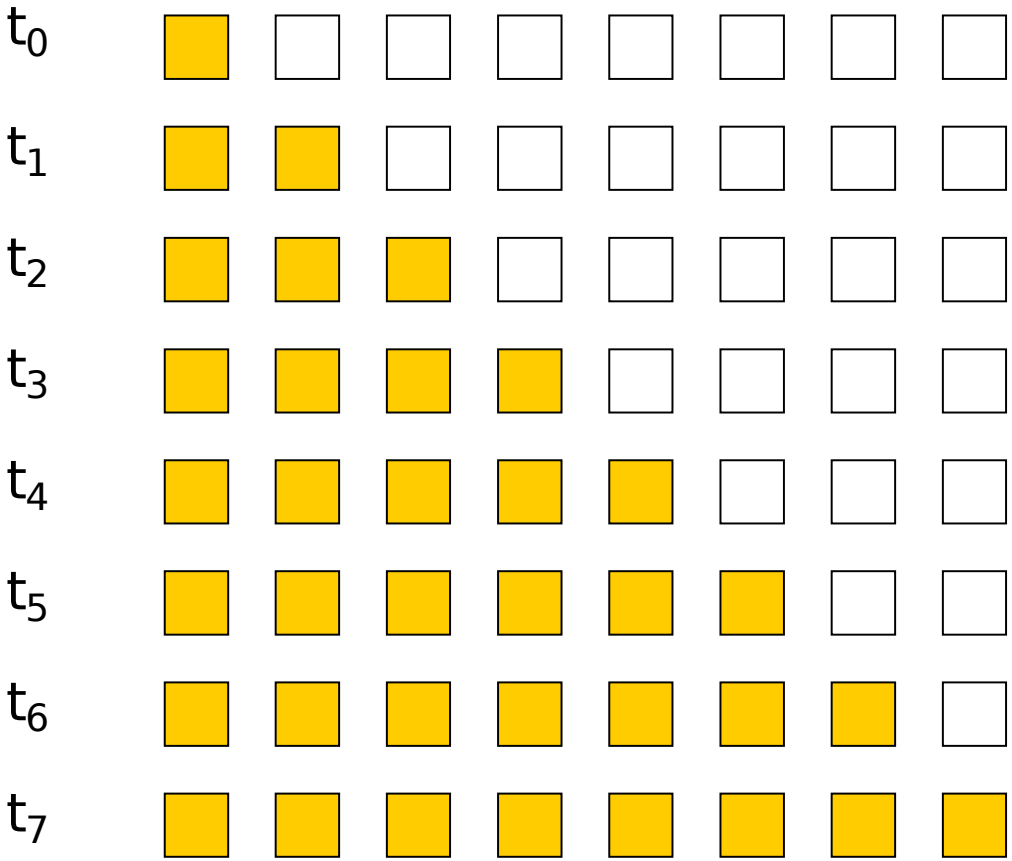
Reflection: Broadcast/Reduction

- Any comments on the broadcast operation?

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

Broadcast: Complexity

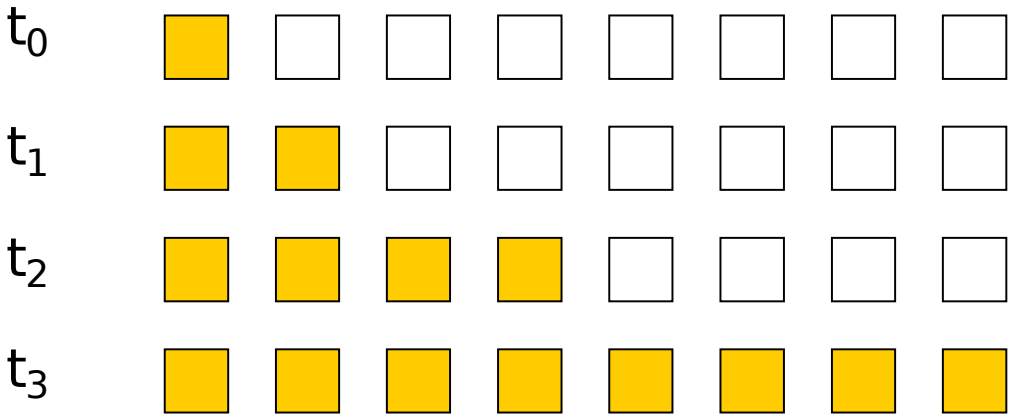
Linear



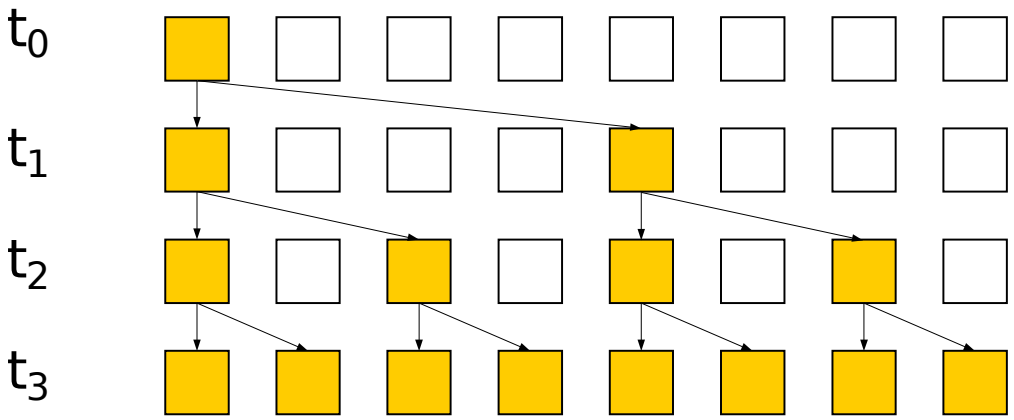
$O(N)$

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

Coding Test – *Lenna*

Coding Test: Assembling Lenna

The problem

- Each process (image) reads and posses a small (square) portion of Lenna, labelled sequentially.
- To have the main process collect portions of Lenna and assemble them into the whole image.
- The main process then writes it out to a PGM file.

The implementation

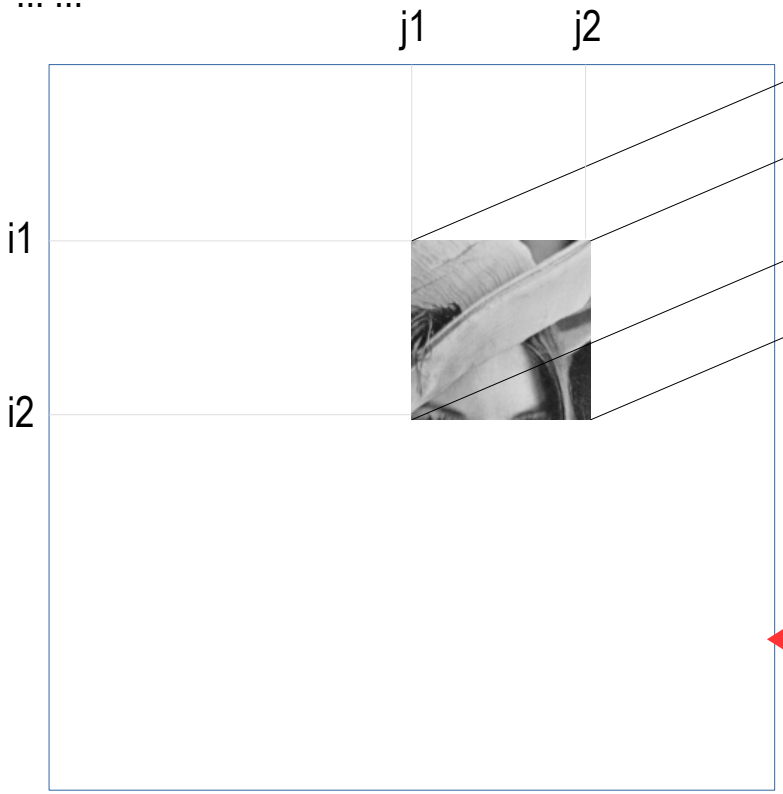
- Use **pic(:,:)** for the whole and **pic_p(:,:)[ ]** for local portion.
- The main process loops over processes, collects the portion from each process and assembles it in the whole array accordingly.

Source: https://en.wikipedia.org/wiki/Lenna

```
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
```

Coding Test: 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
```

Coding Test

The Procedure

- Use the base source provided and complete it;
- Each process – *image* – reads a file containing the distinct portion of Lenna into an array **pic_p** declared as coarray. This is already implemented.
- Image 1 will fetches a portion of Lenna from each of the rest images and assembles the pieces into whole array **pic.**
- You need to calculate the start and end indices of the rows and columns in **pic** where the fetched picture is to be inserted.
- Image 1 then writes the restored picture stored in **pic** to a PGM file (implemented**);**
- Compile the program using the Makefile provided to generate executable lenna

  make
- Run the program with command

  mpirun -n 4 ./lenna
- Use Unix command **display** to view your result

  display Lenna.pgm

References

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

[2] Sun Microsystems, Inc., "***Fortran Programming Guide***", 2005.

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