Parallel Computing for Seismologists

Jiyao Li
jiyao@ldeo.columbia.edu
Lamont-Doherty Earth Observatory
Columbia University

Advantage of Parallel Computing

1. Computing speed 2. Memory space

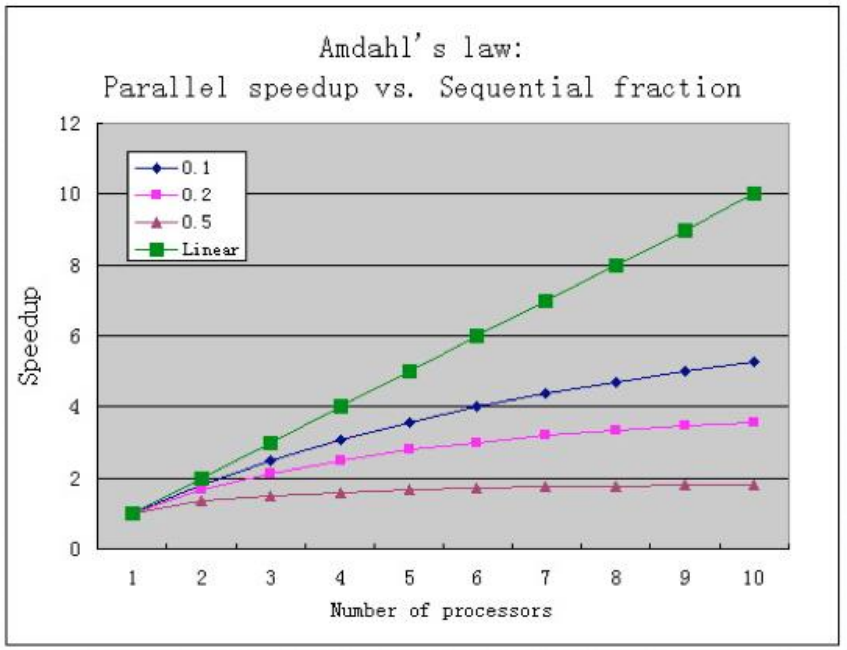
Amdahl's Law

* Describes the time speedup one can expect as a function of the number of processors used and the fraction of parallel code:

speedup = 1/(1-p+p/N)

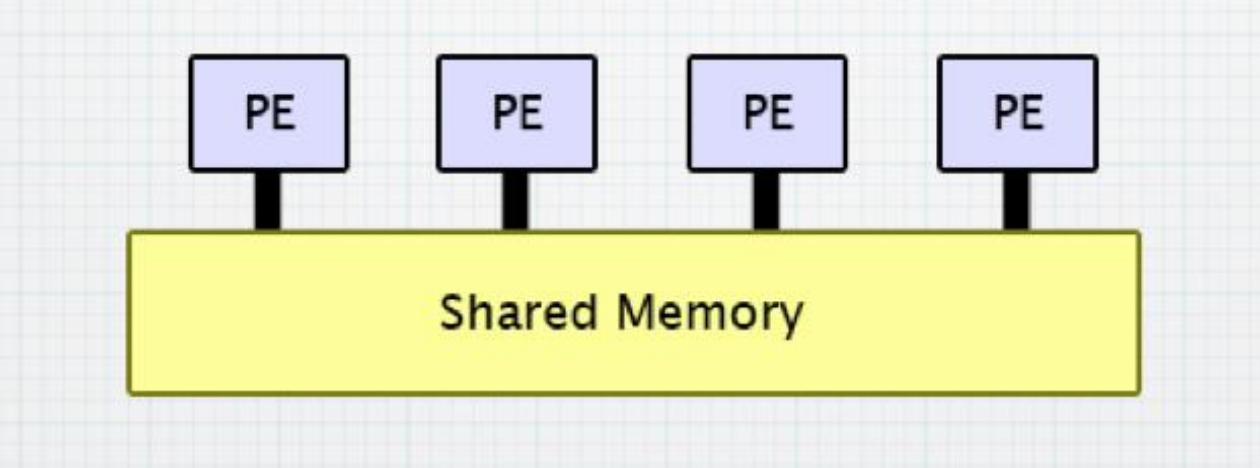
N - number of procs

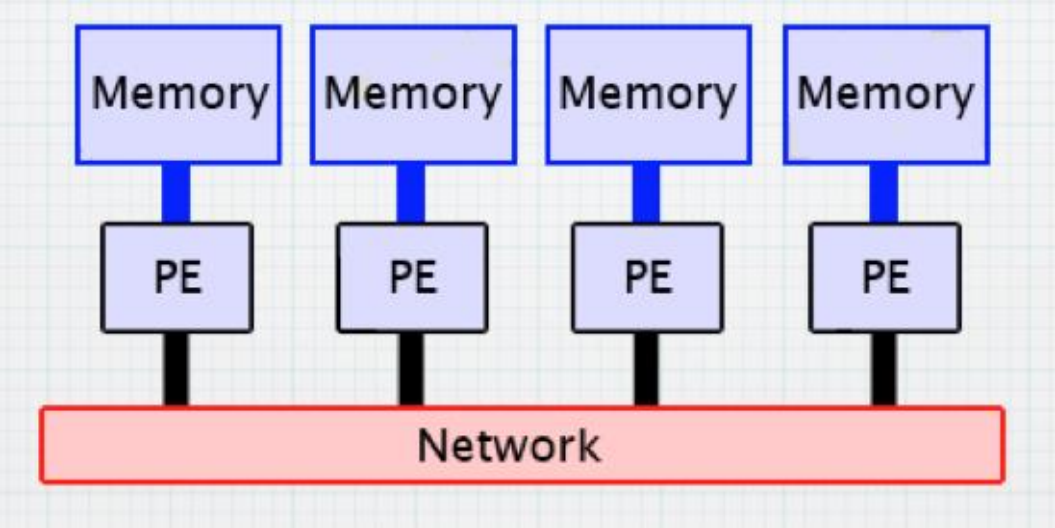
p - fraction of parallel code



Shared Memory vs Distributed Memory

OpenMP vs MPI





Which is better? OpenMP or MPI?

Questions?

- How is parallel computing related to seismologist?
- Processes VS cores?
- How many processes I can/need to run?
- What are “Master” and “Slave”?
- Domain decomposition?
- (How faster can the the parallel process accelerate computing?)

MPI cannot be learned without PRACTICE!

MPI compilers:

MPICH (http://www.mpich.org/)
OpenMPI (http://www.open-mpi.org/)

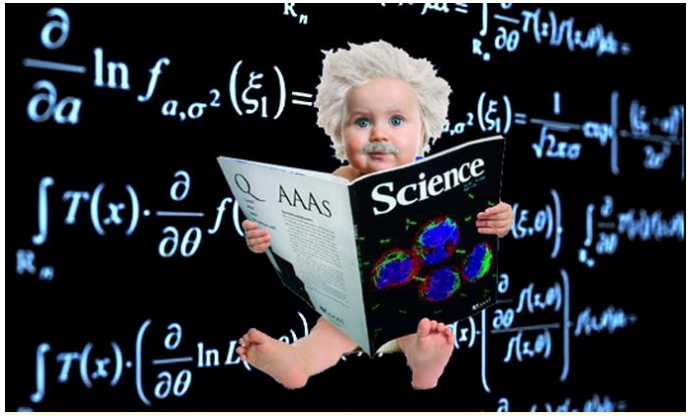
Fortran – ifort, gfortran
C, C++, Java

7 codes (4 categories) to play with:

- Hello1.f90, hello2.f90 (“Hello world!”)
- Pi.f90 (simple integration, broadcast communication)
- Mat_vec.f90 (Ax=b, point to point communication)
- Poisson_mpi.f90 (Poisson equation, Domain decomposition, Nonblocking communication, Virtual topology)
  Poisson_mpi_map.f90, Poisson_serial.f90

Download the codes: https://github.com/lijiyao/MPI_intro

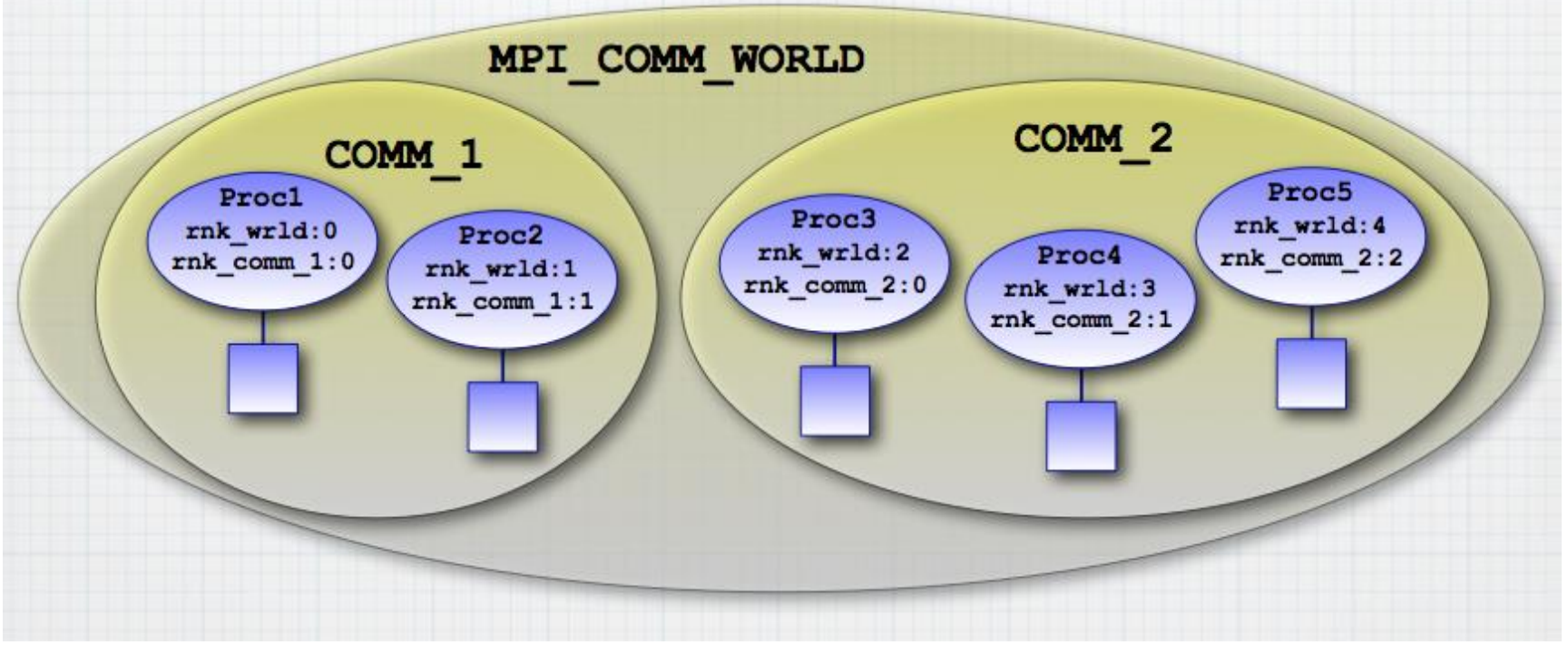
NOT this one!



Hands on

Process, Rank, Communicator

* Processes are identified within a communicator by their rank
  - Rank is an integer
  - Rank defined within the context of a communicator.
  - If a communicator contains n processes, then the ranks are integers from 0 to n-1.



“Hello world” example, hello1.f90

1. Use the **mpi** module, or include the include file called **mpif.h**
2. **Initialize** the MPI environment.
3. Determine **how many processes** are in the current MPI environment.
4. Determine **rank** within the MPI_COMM_WORLD communicator
5. **Terminate** the MPI environment

```
 1 program hello1
 2 use mpi
 3 integer npe_wrld, &! number of processes within the world communicator
 4         rnk_wrld, &! rank of process within the world communicator
 5         ierr
 6
 7 real(kind=8) :: timstart,timend
 8
 9 call MPI_INIT(ierr)  ! initialize MPI environment
10 call MPI_COMM_SIZE(MPI_COMM_WORLD,npe_wrld,ierr) ! determine world size
11 call MPI_COMM_RANK(MPI_COMM_WORLD,rnk_wrld,ierr) ! determine rank within world
12
13 timstart=MPI_WTIME()
14 print *, "Hello world! I'm process ", rnk_wrld," out of ",&
15 npe_wrld, " processes."
16
17 timend=MPI_WTIME()
18 write(*,*) 'elapsed CPU time (s): ',timend-timstart
19
20 call MPI_FINALIZE(ierr)  ! terminate MPI environment
21
22
23 end program hello1
```

Run hello_world

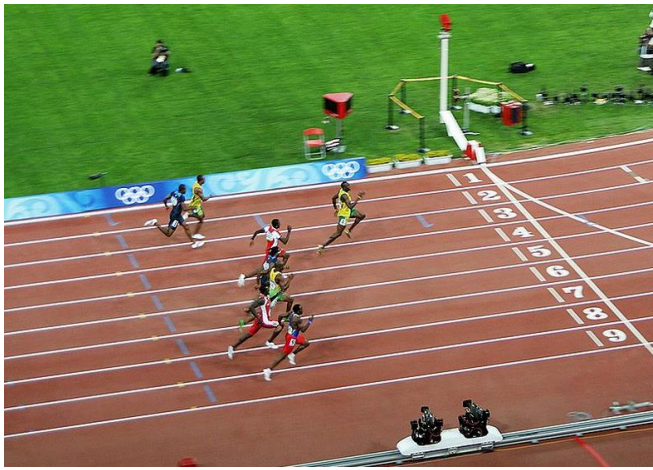
Compile: mpif90 –o hello1 hello1.f90
Run: mpirun –np 8 hello1 .OR. mpirun –np 8 ./hello1

```
jiyao [mpi] 19:01 #29 $mpirun -np 8 hello1
 Hello world! I'm process   0  out of   8  processes.
 Hello world! I'm process   3  out of   8  processes.
 Hello world! I'm process   5  out of   8  processes.
 Hello world! I'm process   6  out of   8  processes.
 Hello world! I'm process   1  out of   8  processes.
 Hello world! I'm process   7  out of   8  processes.
 Hello world! I'm process   2  out of   8  processes.
 Hello world! I'm process   4  out of   8  processes.
```

- # processes != # cores. You can run as many processes as you want (even on your dual-core computer)!
- If # process > # cores, processes will wait and share CPU power (my feeling)
- Execution on each process is not synchronized

Hello2.f90, make the output in order

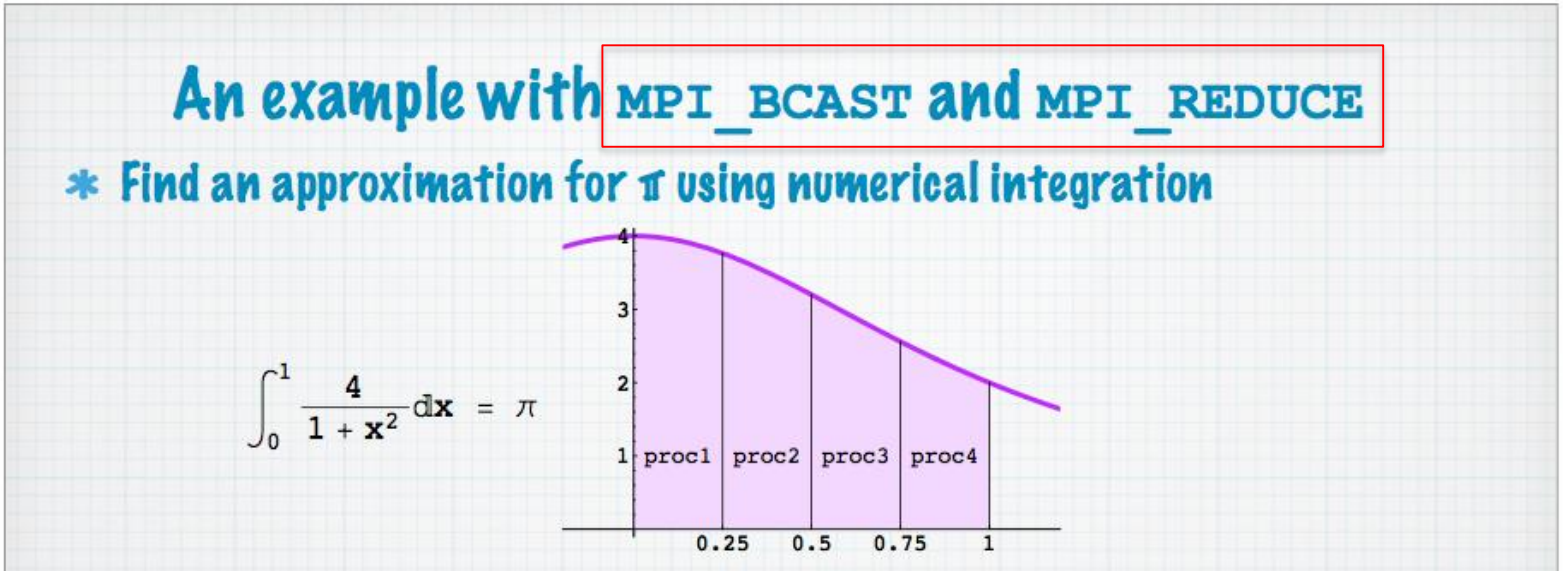
```
27 do n=0,npe_wrld-1          ! do some useless loop, just to make
   the output of process rank in order
28   if(rnk_wrld .eq. n) then
29     time_end=mpi_wtime()
30
31     write(*,*), "hello prom proc =", rnk_wrld," of ", npe_wrld,
    "running on ",trim(proc_name),&
32 " time = ",time_end-time_start
33
34   endif
35   call MPI_BARRIER(MPI_COMM_WORLD,ierr)  ! every process stop h
   ere to wairt for every one to reach this point, then move on
36 enddo
```



MPI_BARRIER(COMM,ierr)

```
jiyao [mpi] 19:01 #29 $mpirun -np 8 hello2
 Hello world! I'm process    0  out of    8  processes.
 Hello world! I'm process    1  out of    8  processes.
 Hello world! I'm process    2  out of    8  processes.
 Hello world! I'm process    3  out of    8  processes.
 Hello world! I'm process    4  out of    8  processes.
 Hello world! I'm process    5  out of    8  processes.
 Hello world! I'm process    6  out of    8  processes.
 Hello world! I'm process    7  out of    8  processes.
```

Collective Communication, example: pi.f90



MPI_BCAST (send_buffer, data_count, data_type, root, comm)

MPI_REDUCE(send_buffer, recv_buffer, send_count, send_type, op, root, comm, ierr)

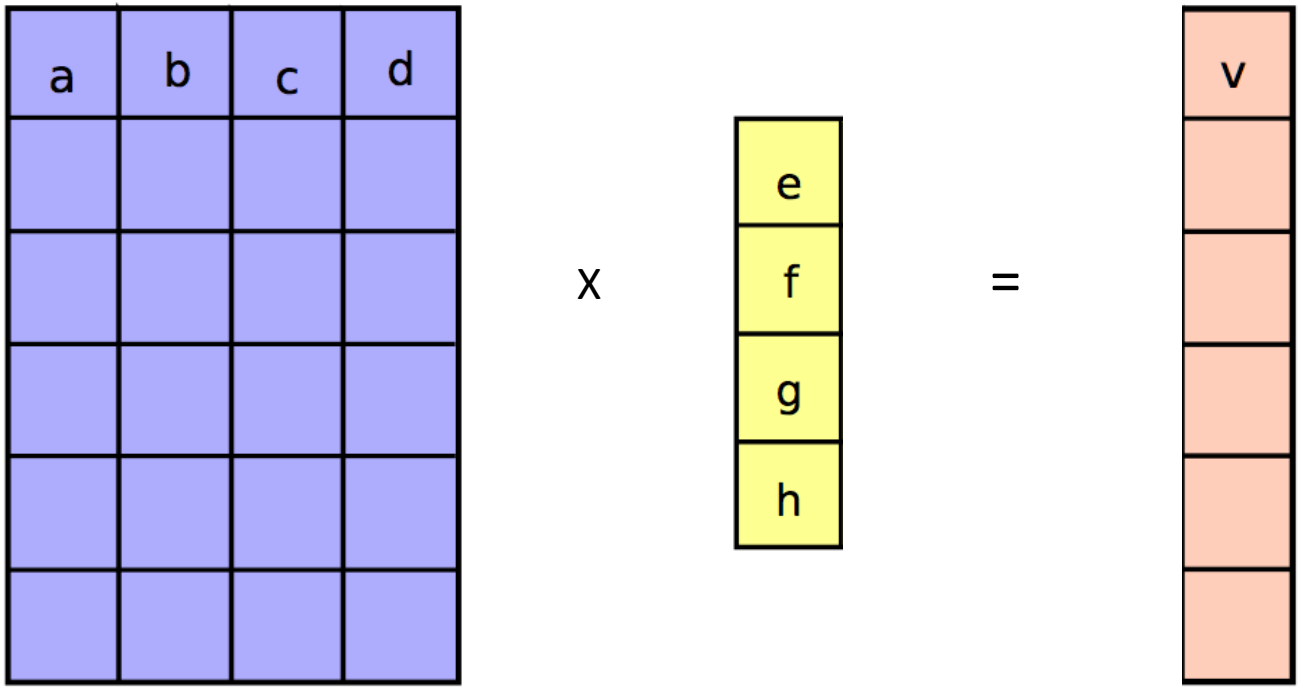
op: MPI_SUM

Point-to-point communication

Matrix-vector multiplication, mat_vec.f90

This is a “master-slave” algorithm. Master is responsible for the coordinating the work for the slaves.

Ax=b



Overall structure of the code, mat_vec.f90

```
PROGRAM mat_vec
USE mpi
IMPLICIT NONE

INTEGER,PARAMETER :: rows=100,cols=100
INTEGER :: npe_wrld,rnk_wrld,master,i,j,count_rows,sender,row_index,ierr
INTEGER :: status(MPI_STATUS_SIZE)
REAL (KIND=SELECTED_REAL_KIND (12)) :: &
   a(rows,cols),b(cols),c(rows),buffer(cols),ans,time_start,time_end

CALL MPI_INIT (ierr)
CALL MPI_COMM_SIZE (MPI_COMM_WORLD,npe_wrld,ierr)
CALL MPI_COMM_RANK (MPI_COMM_WORLD,rnk_wrld,ierr)

master = 0

IF (rnk_wrld==master) THEN ! THE MASTER DOES THIS BLOCK OF CODE
   .
   .
   .
ELSE ! THE SLAVES DO THIS BLOCK OF CODE
   .
   .
   .
ENDIF

CALL MPI_FINALIZE (ierr)

END PROGRAM mat_vec
```

Blocking communication

- **MPI_SEND**
- Control does not return until the message data has been safely stored away so that the sender is free to overwrite the send buffer.
- MPI_SEND (BUFFER,DATA_COUNT,DATA_TYPE,DEST,TAG, COMM,IERR)

- **MPI_RECV**
- Control returns only after the receive buffer contains the newly received message.
- MPI_RECV (BUFFER,DATA_COUNT,DATA_TYPE,SOUR,TAG,COMM, STATUS,IERR)

Algorithm for Master and Slave

- **Master**
- 1. **Send** npe_wrld number of rows, with row_index as tag
- 2**. Receive** message from MPI_ANY_SOURCE, get the slave rank and the row_index
- 3. **if** row_index < max row number, **keep working**
  send the previous slave one row to process
  else row_index >= max row number, **finish working**,
  tell the previous **slave not to send** further information (tag == 0)

- **Slave**
- **If** tag != 0: **work and send result back**, with the row number
  else tag == 0: **finish** (exit)

- After all the computed b is collected and after all the slaves has been told to stop, the code will be finished

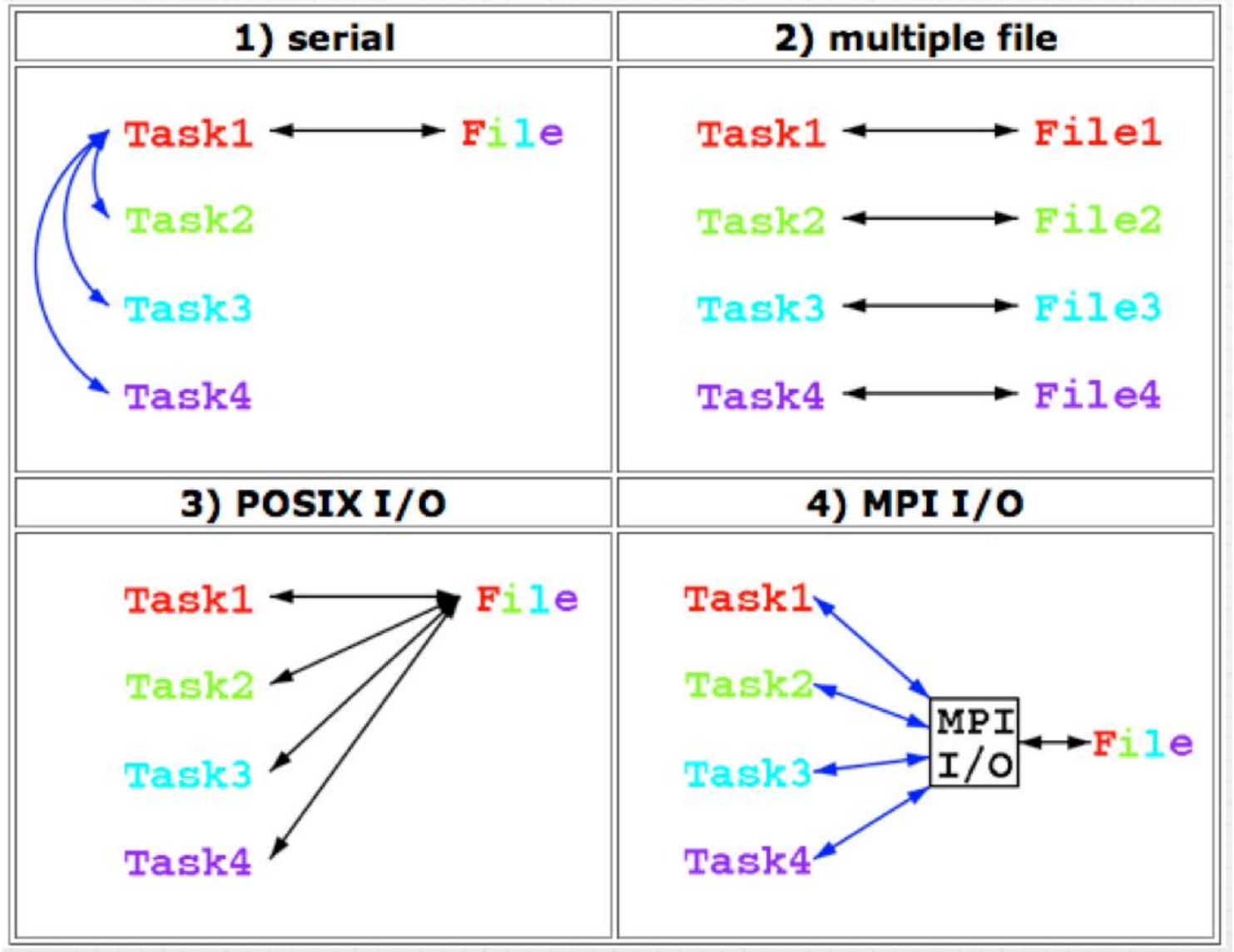
Master code

```
DO j = 1,cols ! make an arbitrary matrix a and vector b
   b(j) = 1.0_8
   DO i = 1,rows
      a(i,j) = DBLE (i+j)
   ENDDO
ENDDO
CALL MPI_BCAST (b,cols,MPI_DOUBLE_PRECISION,master,MPI_COMM_WORLD,ierr)

count_rows = 0
DO i = 1,npe_wrld-1
   DO j = 1,cols
      buffer(j) = a(i,j)
   ENDDO
   CALL MPI_SEND (buffer,cols,MPI_DOUBLE_PRECISION,i,i,MPI_COMM_WORLD,ierr)
   count_rows = count_rows+1
ENDDO

DO i = 1,rows
   CALL MPI_RECV (ans,1,MPI_DOUBLE_PRECISION, &
                  MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,status,ierr)
   sender    = status(MPI_SOURCE)
   row_index = status(MPI_TAG) ! tag value in status is the row index
   c(row_index) = ans
   IF (count_rows < rows) THEN ! more work to be done.  send another row
      DO j = 1,cols
         buffer(j) = a(count_rows+1,j)
      ENDDO
      CALL MPI_SEND (buffer,cols,MPI_DOUBLE_PRECISION, &
                                 sender,count_rows+1,MPI_COMM_WORLD,ierr)
      count_rows = count_rows+1
   ELSE ! tell sender that there is no more work
      CALL MPI_SEND (MPI_BOTTOM,0,MPI_DOUBLE_PRECISION,sender,0,MPI_COMM_WORLD,ierr)
   ENDIF
ENDDO
```

Slave code

```
CALL MPI_BCAST (b,cols,MPI_DOUBLE_PRECISION,master,MPI_COMM_WORLD,ierr)
DO
   CALL MPI_RECV (buffer,cols,MPI_DOUBLE_PRECISION,master, &
                                    MPI_ANY_TAG,MPI_COMM_WORLD,status,ierr)
   IF (status(MPI_TAG)==0) EXIT ! there is no more work
   row_index = status(MPI_TAG)  ! tag value status is the row index
   ans = 0.0_8
   DO i = 1,cols
      ans = ans + buffer(i)*b(i)
   ENDDO
   CALL MPI_SEND (ans,1,MPI_DOUBLE_PRECISION, &
                                   master,row_index,MPI_COMM_WORLD,ierr)
ENDDO
```

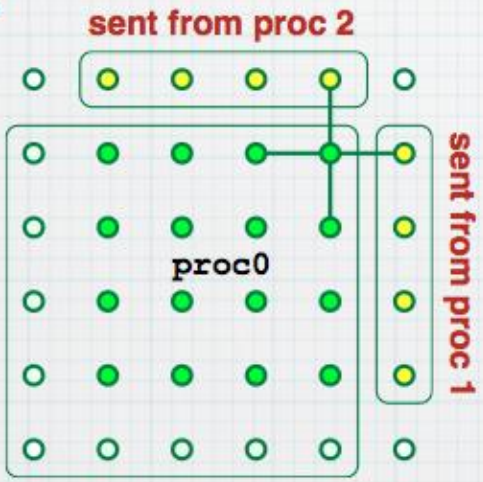
Parallel I/O

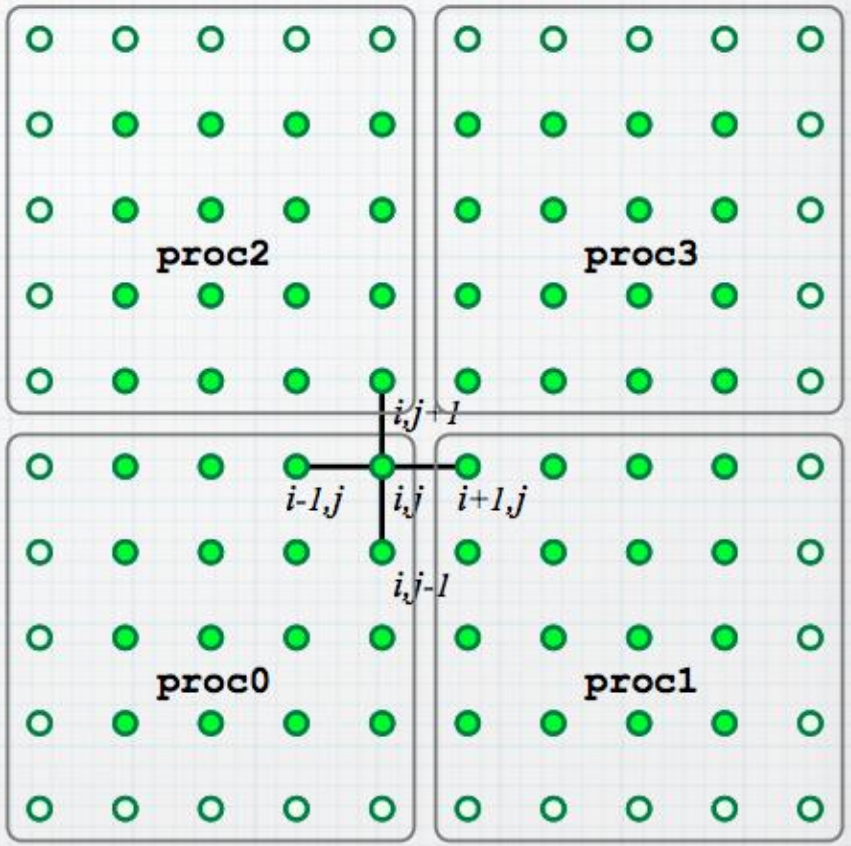

1) serial
Task1
File
Task2
Task3
Task4
2) multiple file
Task1
File1
Task2
File2
Task3
File3
Task4
File4
3) POSIX I/O
Task1
File
Task2
Task3
Task4
4) MPI I/O
Task1
Task2
MPI
I/O
File
Task3
Task4

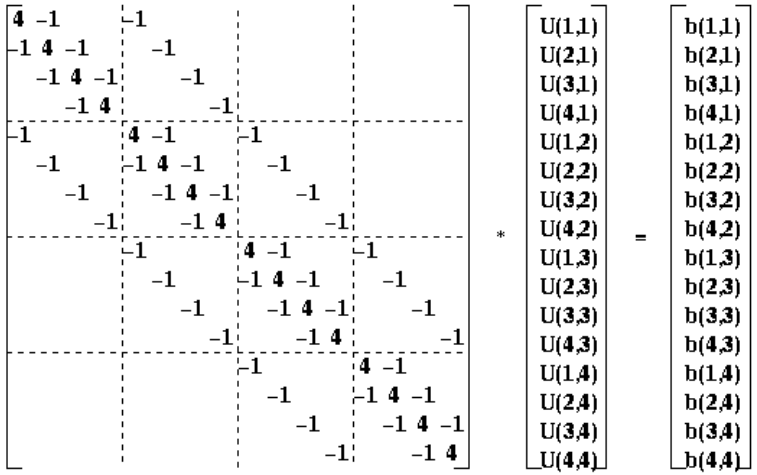
Domain Decomposition
solve discrete Poisson problem,

poisson_mpi.f90

* For the case when n = 8, the 10 X 10 grid looks like this:
  - Solid green is a interior grid points
  - open circle is a boundary point
* Suppose we divide the grid to four processes.
* Then, for example, `proc0` is assigned an array 6X6 like this:







Poisson's equation

$$\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = f(x,y)$$

$$\left(\frac{\partial^2 u}{\partial x^2}\right)_i = \frac{u_{i+1} - 2u_i + u_{i-1}}{(\Delta x)^2} + \mathcal{O}(\Delta x)^2$$

u(i-1,j)+u(i+1,j)-2*u(i,j)+u(i,j-1)+u(i,+1)-2*u(i,j)=b(i,j)
b(i,j)=f(x,y)*h^2, h is the space interval

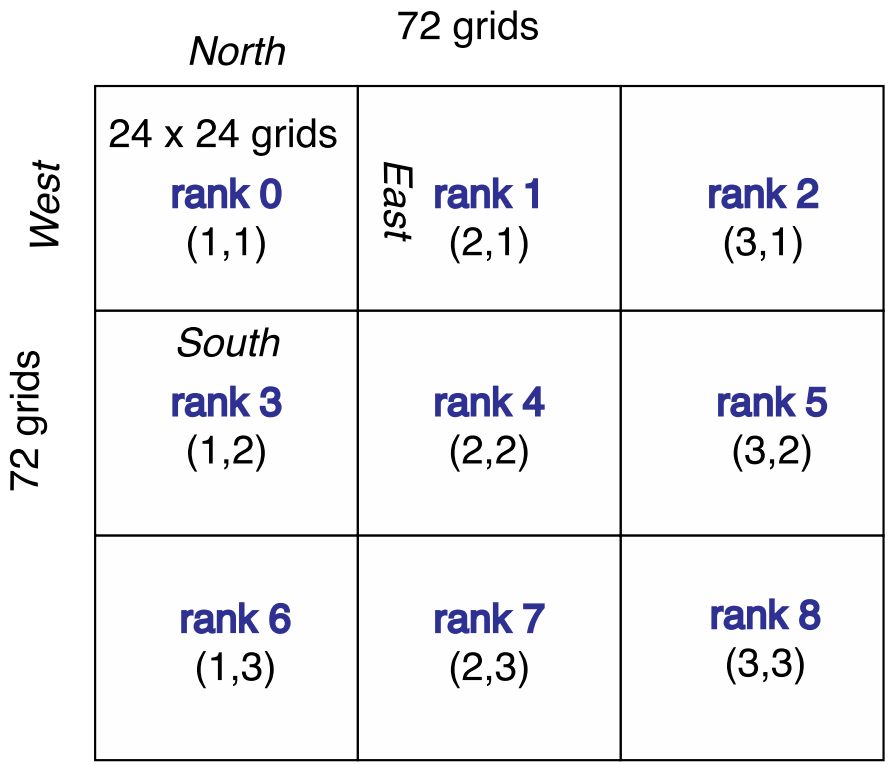
Jacobi's Method

u(i,j)=[u(i-1,j)+u(i+1,j) +u(i,j-1)+u(i,j+1)-b(i,j)]/4

We let u(i,j,m) be our approximation for u(i,j) after the m-th iteration. We can use any initial guess u(i.j,0); u(I,j,0)=0 will do. u(i,j,m+1) is just a weighted average of its four neighboring values and b(i,j)

u(i,j,m+1)=[u(i-1,j,m)+u(i+1,j,m) +u(i,j-1,m)+u(i,j+1,m)-b(i,j)]/4

This is now similar to time-dependent PDEs (e.g. wave equation, heat flow)



Virtual Topology

rank 0 (1)
rank 1 (2)
rank 2 (3)
rank 3 (4)
rank 4 (5)
rank 5 (6)
rank 6 (7)
rank 7 (8)
rank 8 (9)

Virtual Topology

Output from poisson_mpi_map.f90:

```
        0        9
iblk        1 jblk        1
nghbr_list :
        1        3       -1       -1
nghbr_count        2
**************************
        1        9
iblk        2 jblk        1
nghbr_list :
        2        4        0       -1
nghbr_count        3
**************************
```

Neighbor list:
East, South, West, North

Nonblocking communication

- **MPI_ISEND**
- MPI_ISEND (BUFFER,DATA_COUNT,DATA_TYPE, DEST,TAG,COMM,REQUEST,IERR)

- **MPI_IRECV**
- MPI_IRECV (BUFFER,DATA_COUNT,DATA_TYPE, SOUR,TAG,COMM,REQUEST,IERR)

- **MPI_WAITALL**
- MPI_WAITALL(COUNT,ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES,IERR)

Algorithm

- Topology (Make the map)
- 1. Communicate information to fill ghost cells
  a. Initiate nonblocking sends
  b. Initiate nonblocking receives
  c. Wait for message to be completed
- 2. Perform one sweep of the Jacobi iteration for all the grid points
- 3. GOTO 1.

Results:

Gravity anomaly (field) caused by density anomaly (source)

The results from parallel code (poisson_mpi.f90) are exactly the same as the one from the serial code (poisson_serial.f90)

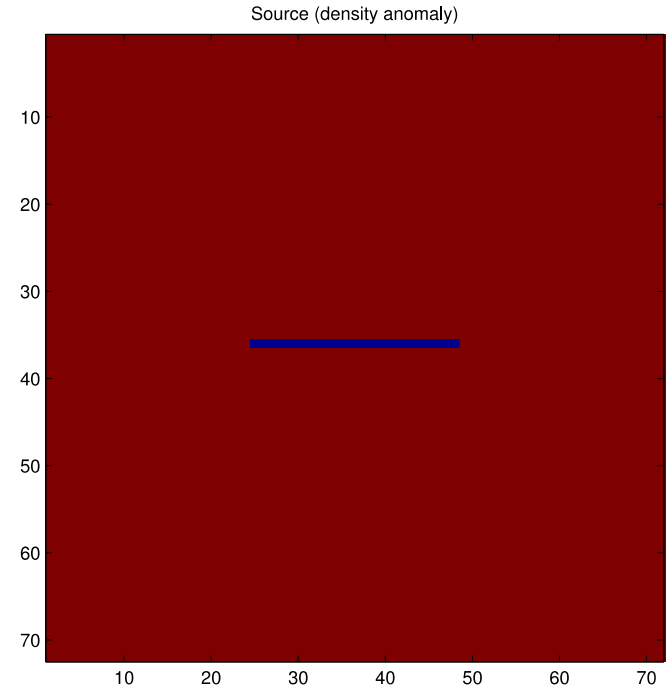
Speeding up? (could not be measured with my dual-core Macbook)

a

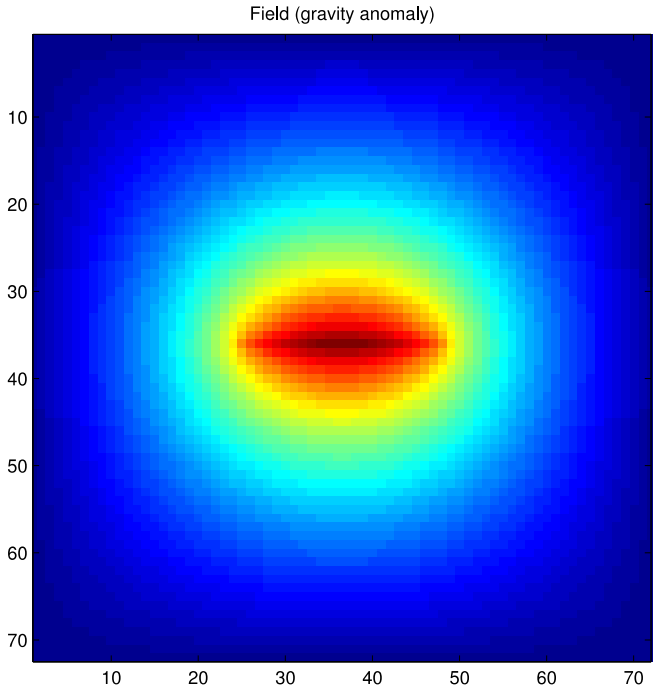
72 grids

	North	
West	24 x 24 grids **rank 0** (1,1)	*East* **rank 1** (2,1)	**rank 2** (3,1)
72 grids	*South* **rank 3** (1,2)	**rank 4** (2,2)	**rank 5** (3,2)
	rank 6 (1,3)	**rank 7** (2,3)	**rank 8** (3,3)

b

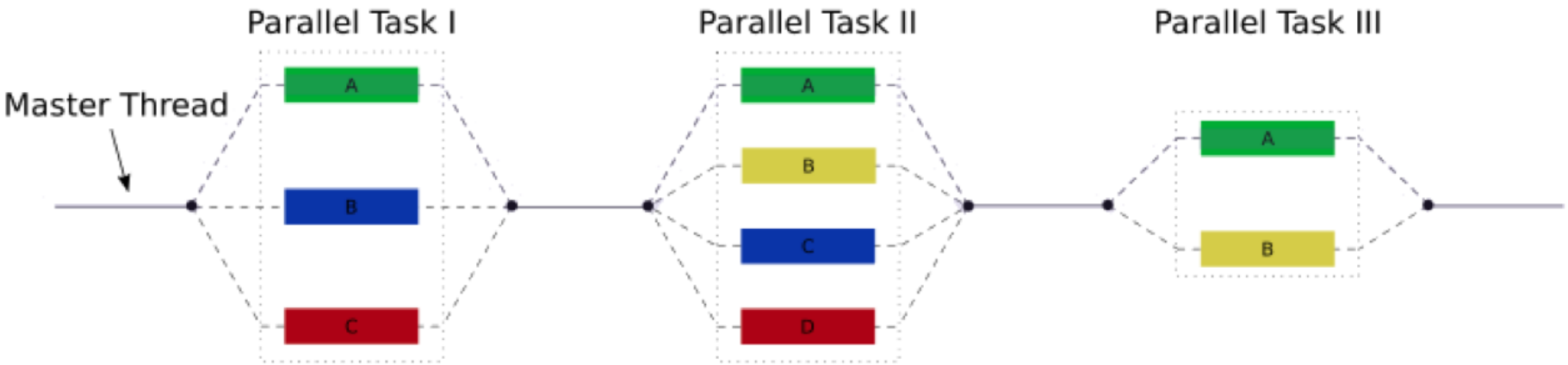


c



Summary

- Seismologist should and could know MPI.
- Processes VS cores?
- How many processes I can/need to run?
- What are “Master” and “Slave”?
- Domain decomposition?
- (How faster can the the parallel process accelerate computing?)
- Parallel codes are complicated and could have nasty bugs, so always start with serial codes and put them into the simple MPI code structures.



OpenMP
example

Matrix Multiplication in OpenMP

```
CALL OMP_SET_NUM_THREADS(whatever)
!$OMP PARALLEL DO
! note: loops on J and K are serial
DO i = 1, N
  DO J = 1, M
    DO K = 1, P
      C(i,K) = C(i,K) + A(i,J) * B(J,K)
    ENDDO
  ENDDO
ENDDO
!$OMP END PARALLEL DO
```