Introduction to Programming by MPI for Parallel FEM Report S1 & S2 in C (1/2)

Kengo Nakajima
Information Technology Center
The University of Tokyo

Motivation for Parallel Computing (and this class)

- Large-scale parallel computer enables fast computing in large-scale scientific simulations with detailed models. Computational science develops new frontiers of science and engineering.
- Why parallel computing ?
  - faster & larger
  - “larger” is more important from the view point of “new frontiers of science & engineering”, but “faster” is also important.
  - + more complicated
  - Ideal: Scalable
    - Solving N^x scale problem using N^x computational resources during same computation time.

Scalable, Scaling, Scalability

- Solving N^x scale problem using N^x computational resources during same computation time
  - for large-scale problems: **Weak Scaling, Weak Scalability**
  - e.g. CG solver: more iterations needed for larger problems
- Solving a problem using N^x computational resources during 1/N computation time
  - for faster computation: **Strong Scaling, Strong Scalability**

Overview

- What is MPI ?
- Your First MPI Program: Hello World
- Collective Communication
- Point-to-Point Communication

What is MPI ? (1/2)

- Message Passing Interface
- “Specification” of message passing API for distributed memory environment
  - Not a program, Not a library
    - http://www.mcs.anl.gov/mpi/www/
    - https://www.mpi-forum.org/docs/
- History
  - 1992 MPI Forum
    - https://www.mpi-forum.org/
  - 1994 MPI-1
  - 1997 MPI-2: MPI I/O
  - 2012 MPI-3: Fault Resilience, Asynchronous Collective
- Implementation
  - mpich ANL (Argonne National Laboratory), OpenMPI, MVAPICH
  - H/W vendors
  - C/C++, FOTRAN, Java ; Unix, Linux, Windows, Mac OS

What is MPI ? (2/2)

- “mpich” (free) is widely used
  - supports MPI-2 spec. (partially)
  - MPICH2 after Nov. 2005.
  - http://www.mcs.anl.gov/mpi/
- Why MPI is widely used as *de facto standard* ?
  - Uniform interface through MPI forum
    - Portable, can work on any types of computers
    - Can be called from Fortran, C, etc.
  - mpich
    - free, supports every architecture
- PVM (Parallel Virtual Machine) was also proposed in early 90’s but not so widely used as MPI

References

- W.Gropp et al., Using MPI second edition, MIT Press, 1999.
- M.J.Quinn, Parallel Programming in C with MPI and OpenMP, McGrawhill, 2003.
- W.Gropp et al., MPI：The Complete Reference Vol.I, II, MIT Press, 1998.
- http://www.mcs.anl.gov/mpi/www/
  - API (Application Interface) of MPI

How to learn MPI (1/2)

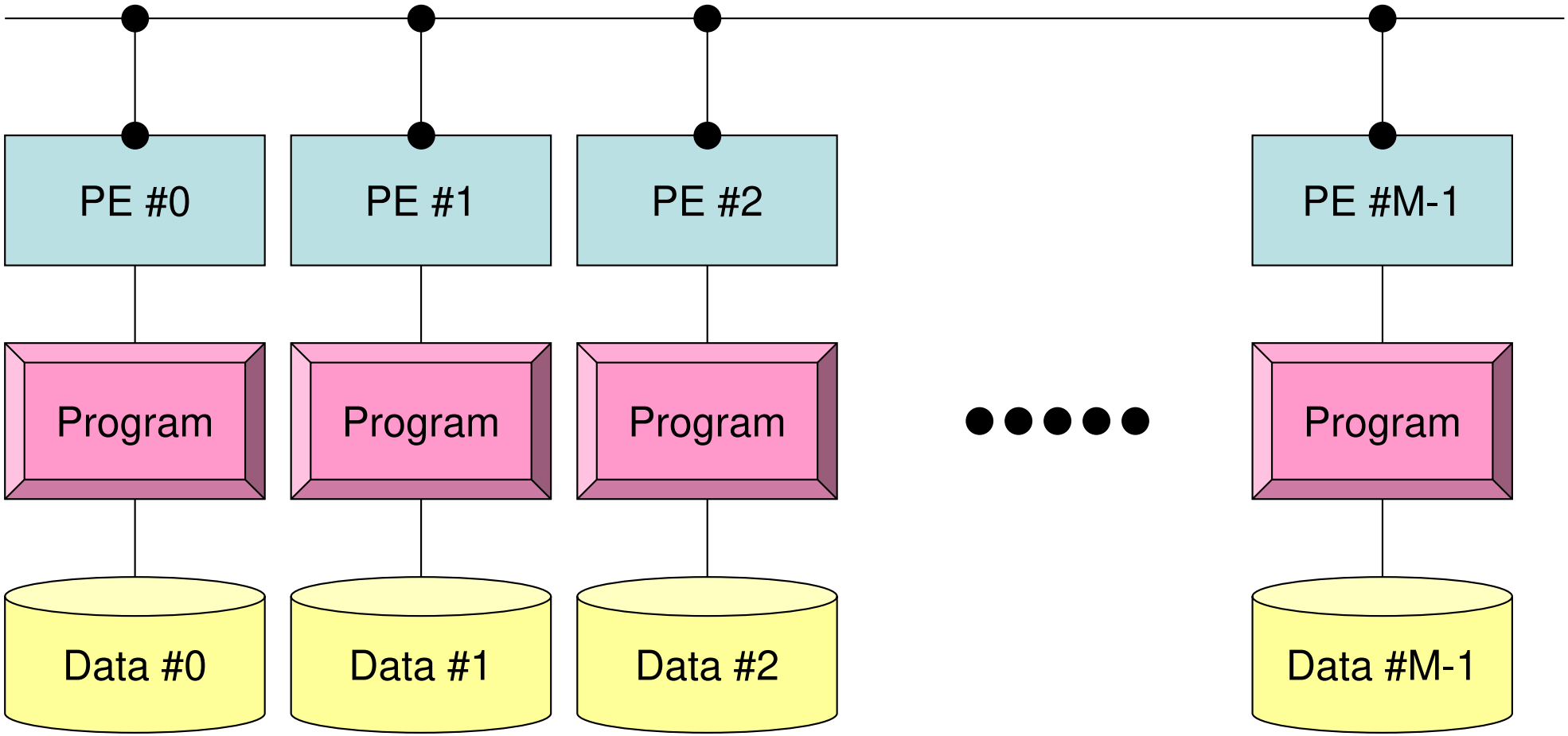
- Grammar
  - 10-20 functions of MPI-1 will be taught in the class
    - although there are many convenient capabilities in MPI-2
  - If you need further information, you can find information from web, books, and MPI experts.
- Practice is important
  - Programming
  - "Running the codes" is the most important
- Be familiar with or "grab" the idea of SPMD/SIMD op's
  - Single Program/Instruction Multiple Data
  - Each process does same operation for different data
    - Large-scale data is decomposed, and each part is computed by each process
  - Global/Local Data, Global/Local Numbering

PE: Processing Element
Processor, Domain, Process

SPMD

You understand 90% MPI, if you understand this figure.

```
mpirun -np M <Program>
```



Each process does same operation for different data

Large-scale data is decomposed, and each part is computed by each process

It is ideal that parallel program is not different from serial one except communication.

Some Technical Terms

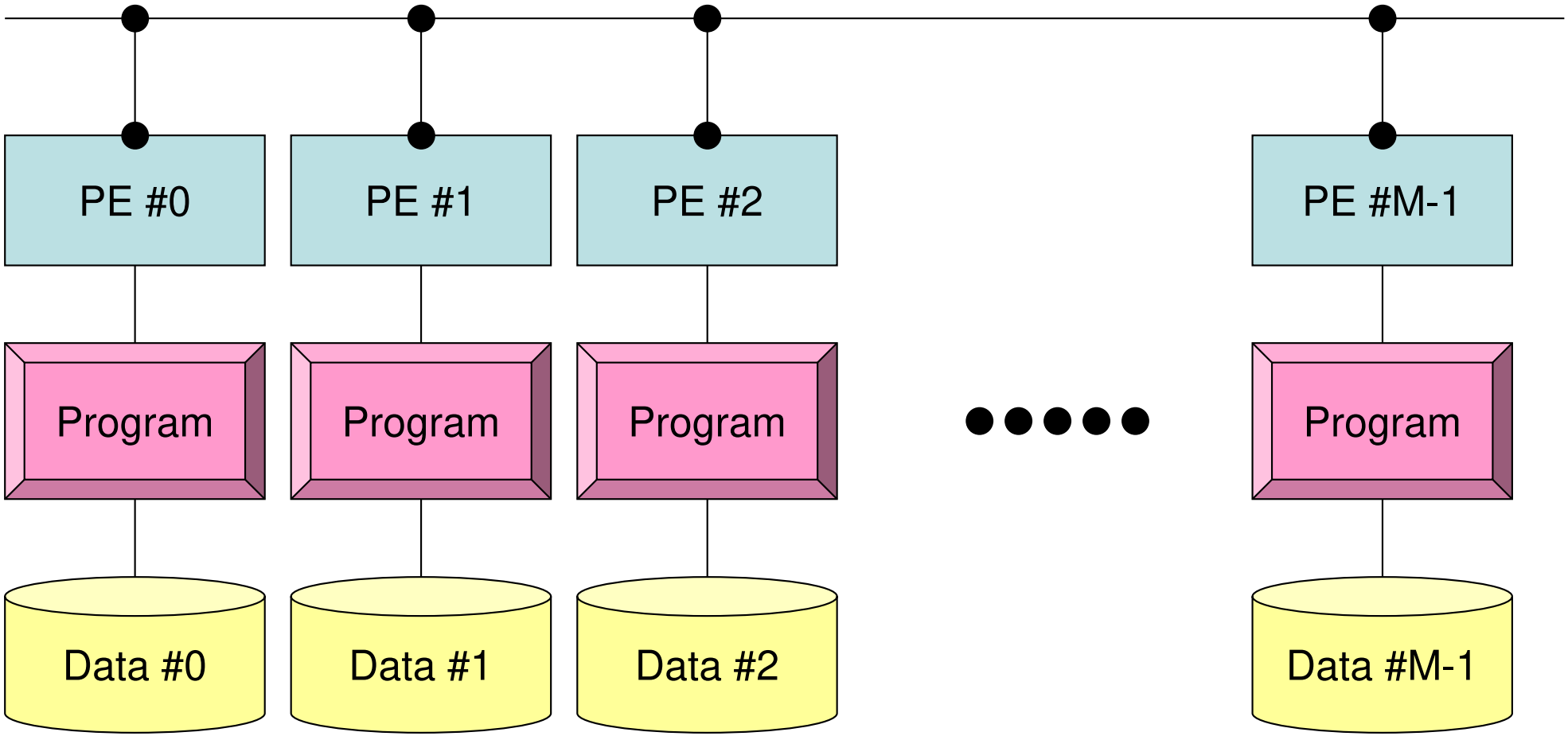
- Processor, Core
  - Processing Unit (H/W), Processor=Core for single-core proc's
- Process
  - Unit for MPI computation, nearly equal to "core"
  - Each core (or processor) can host multiple processes (but not efficient)
- PE (Processing Element)
  - PE originally mean "processor", but it is sometimes used as "process" in this class. Moreover it means "domain" (next)
    - In multicore proc's: PE generally means "core"
- Domain
  - domain=process (=PE), each of "MD" in "SPMD", each data set
- Process ID of MPI (ID of PE, ID of domain) starts from "0"
  - if you have 8 processes (PE's, domains), ID is 0~7

PE: Processing Element
Processor, Domain, Process

SPMD

You understand 90% MPI, if you understand this figure.

```
mpirun -np M <Program>
```



Each process does same operation for different data

Large-scale data is decomposed, and each part is computed by each process

It is ideal that parallel program is not different from serial one except communication.

How to learn MPI (2/2)

- NOT so difficult.
- Therefore, 5-6-hour lectures are enough for just learning grammar of MPI.

- Grab the idea of SPMD !

Schedule

- MPI
  - Basic Functions
  - Collective Communication
  - Point-to-Point (or Peer-to-Peer) Communication

- 105 min. x 3-4 lectures
  - Collective Communication
    - Report S1
  - Point-to-Point Communication
    - Report S2: Parallelization of 1D code
  - At this point, you are almost an expert of MPI programming.

- What is MPI ?
- **Your First MPI Program: Hello World**
- Collective Communication
- Point-to-Point Communication

Login to Oakbridge-CX (OBCX)

```
ssh t73**@obcx.cc.u-tokyo.ac.jp
```

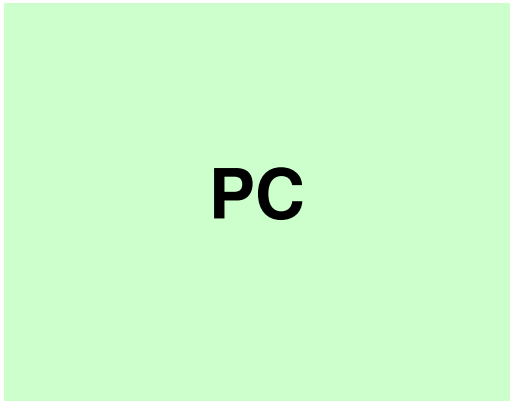
Create directory

```
>$ cd /work/gt73/t73XXX
>$ mkdir pFEM
>$ cd pFEM
```

In this class this top-directory is called `<$O-TOP>`.
Files are copied to this directory.

Under this directory, `S1, S2, S1-ref` are created:

```
<$O-S1> = <$O-TOP>/mpi/S1
<$O-S2> = <$O-TOP>/mpi/S2
```



Copying files on OBCX

Fortan

```
>$ cd /work/gt73/t73XXX/pFEM
>$ cp /work/gt73/z30088/pFEM/F/s1-f.tar .
>$ tar xvf s1-f.tar
```

C

```
>$ cd /work/gt73/t73XXX/pFEM
>$ cp /work/gt73/z30088/pFEM/C/s1-c.tar .
>$ tar xvf s1-c.tar
```

Please include "-no-multibyte-chars" as a option for compiling by C, if you have any "multi-byte" errors.

Confirmation

```
>$ ls
    mpi

>$ cd mpi/S1
```

This directory is called as <$O-S1>.

<$O-S1> = <$O-TOP>/mpi/S1

First Example

hello.f

```
implicit REAL*8 (A-H,O-Z)
include 'mpif.h'
integer :: PETOT, my_rank, ierr

call MPI_INIT      (ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD, PETOT, ierr )
call MPI_COMM_RANK (MPI_COMM_WORLD, my_rank, ierr )

write (*,'(a,2i8)') 'Hello World FORTRAN', my_rank, PETOT

call MPI_FINALIZE (ierr)

stop
end
```

hello.c

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
    int n, myid, numprocs, i;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    printf ("Hello World %d¥n", myid);
    MPI_Finalize();
}
```

Compiling hello.f/c

```
>$ cd /work/gt73/t73XXX/pFEM/mpi/S1
>$ mpiifort -align array64byte -O3 -axCORE-AVX512 hello.f
>$ mpiicc -align -O3 -axCORE-AVX512 hello.c
```

FORTRAN

$> "mpiifort":
required compiler & libraries are included for FORTRAN90+MPI

C

$> "mpiicc":
required compiler & libraries are included for C+MPI

Running Job

- Batch Jobs
  - Only batch jobs are allowed.
  - Interactive executions of jobs are not allowed.
- How to run
  - writing job script
  - submitting job
  - checking job status
  - checking results
- Utilization of computational resources
  - 1-node (56 cores) is occupied by each job.
  - Your node is not shared by other jobs.

Job Script

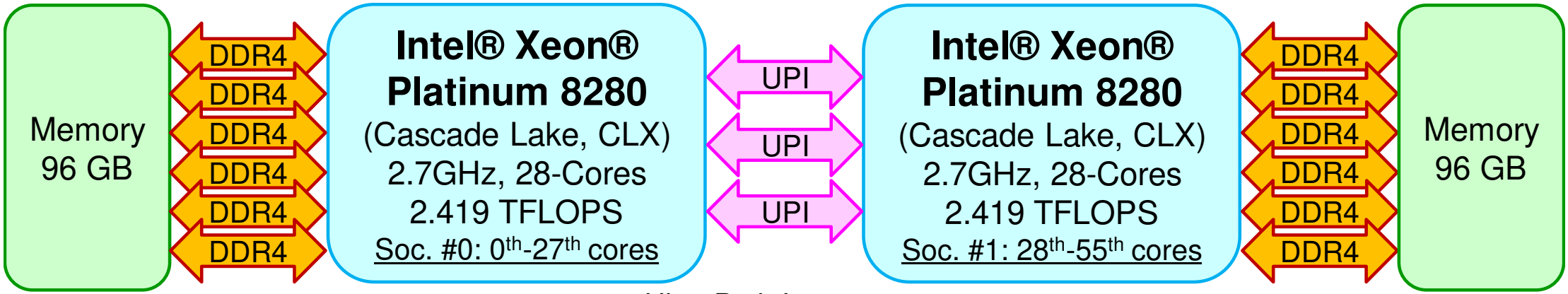
- **<$O-S1>/hello.sh**
- Scheduling + Shell Script

```
#!/bin/sh
#PJM -N "hello"                          Job Name
#PJM -L rscgrp=lecture3                  Name of "Resource Group"
#PJM -L node=1                           Node#
#PJM --mpi proc=4                        Total MPI Process#
#PJM -L elapse=00:15:00                  Computation Time
#PJM -g gt73                             Group Name (Wallet)
#PJM -j
#PJM -e err                              Standard Error
#PJM -o hello.lst                        Standard Output

mpiexec.hydra -n ${PJM_MPI_PROC} ./a.out
```

```
mpiexec.hydra          Command for Running MPI JOB
-n ${PJM_MPI_PROC}     = (--mpi proc=XX), in this case =4
./a.out                name of executable file
```

Process Number

```
#PJM -L node=1;#PJM --mpi proc= 1    1-node, 1-proc, 1-proc/n
#PJM -L node=1;#PJM --mpi proc= 4    1-node, 4-proc, 4-proc/n
#PJM -L node=1;#PJM --mpi proc=16    1-node,16-proc,16-proc/n
#PJM -L node=1;#PJM --mpi proc=28    1-node,28-proc,28-proc/n
#PJM -L node=1;#PJM --mpi proc=56    1-node,56-proc,56-proc/n

#PJM -L node=4;#PJM --mpi proc=128   4-node,128-proc,32-proc/n
#PJM -L node=8;#PJM --mpi proc=256   8-node,256-proc,32-proc/n
#PJM -L node=8;#PJM --mpi proc=448   8-node,448-proc,56-proc/n
```



2933 MHz × 6ch
140.8 GB/sec

Ultra Path Interconnect
10.4 GT/sec × 3
= 124.8 GB/sec

2933 MHz × 6ch
140.8 GB/sec

Job Submission

```
>$ cd /work/gt73/t73XXX/pFEM/mpi/S1

(modify hello.sh)
>$ pjsub hello.sh

>$ cat hello.lst

  Hello World 0
  Hello World 3
  Hello World 2
  Hello World 1
```

Available “Resource Group’s”

- Following 2 resource groups are available.
- 8 nodes can be used
  - **`lecture`**
    - 8 nodes (448 cores), 15 min., valid until the end of March 2022
    - Shared by all “educational” users
  - **`lecture3`**
    - 8 nodes (448 cores), 15 min., active during class time
    - More jobs (compared to **`lecture`**) can be processed up on availability.

Submitting & Checking Jobs

• Submitting Jobs	`pjsub SCRIPT NAME`
• Checking status of jobs	`pjstat`
• Deleting/aborting	`pjdel JOB ID`
• Checking status of queues	`pjstat --rsc`
• Detailed info. of queues	`pjstat --rsc -x`
• Info of running jobs	`pjstat -a`
• History of Submission	`pjstat -H`
• Limitation of submission	`pjstat --limit`

```
[t73XYZ@obcx04 run]$ pjsub go1.sh
[INFO] PJM 0000 pjsub Job 292019 submitted.

[t73XYZ@obcx04 run]$ pjsub go2.sh
[INFO] PJM 0000 pjsub Job 292020 submitted.

[t73XYZ@obcx04 run]$ pjstat
Oakbridge-CX scheduled stop time: 2020/04/24(Fri) 09:00:00 (Remain:  3days 23:09:15)

JOB_ID       JOB_NAME   STATUS  PROJECT  RSCGROUP       START_DATE        ELAPSE          TOKEN NODE
292019       test1      RUNNING gt73     lecture        04/20 09:50:42<   00:00:02            -    1
292020       test2      QUEUED  gt73     lecture        --/-- --:--:--    00:00:00            -    1

[t73XYZ@obcx04 run]$ pjstat
Oakbridge-CX scheduled stop time: 2020/04/24(Fri) 09:00:00 (Remain:  3days 23:09:12)

JOB_ID       JOB_NAME   STATUS  PROJECT  RSCGROUP       START_DATE        ELAPSE          TOKEN NODE
292019       test1      RUNNING gt73     lecture        04/20 09:50:42<   00:00:06            -    1
292020       test2      RUNNING gt73     lecture        04/20 09:50:46<   00:00:02            -    1

[t73XYZ@obcx04 run]$ pjdel 292020
[INFO] PJM 0100 pjdel Job 292020 canceled.

[t73XYZ@obcx04 run]$ pjstat
Oakbridge-CX scheduled stop time: 2020/04/24(Fri) 09:00:00 (Remain:  3days 23:09:04)

JOB_ID       JOB_NAME   STATUS  PROJECT  RSCGROUP       START_DATE        ELAPSE          TOKEN NODE
292019       test1      RUNNING gt73     lecture        04/20 09:50:42<   00:00:14            -    1

[t73XYZ@obcx04 run]$ pjstat
Oakbridge-CX scheduled stop time: 2020/04/24(Fri) 09:00:00 (Remain:  3days 23:07:14)

No unfinished job found.
```

```
[t73XYZ@obcx04 ~]$ pjstat --rsc
RSCGRP                  STATUS              NODE
lecture                 [ENABLE,START]        32
lecture3                [DISABLE,STOP]        64

[t73XYZ@obcx04 ~]$ pjstat --rsc -x
RSCGRP                  STATUS           MIN_NODE MAX_NODE MAX_ELAPSE REMAIN_ELAPSE MEM(GB) PROJECT
lecture                 [ENABLE,START]          1        8   00:15:00      00:15:00     168 gt62
lecture3                [DISABLE,STOP]          1        8   00:15:00      --:--:--     168 gt62

[t73XYZ@obcx04 ~]$ pjstat -a
Oakbridge-CX scheduled stop time: 2020/04/24(Fri) 09:00:00 (Remain:  3days 23:19:29)

JOB_ID       JOB_NAME   STATUS  PROJECT  RSCGROUP       START_DATE        ELAPSE           TOKEN NODE
284147       ********** RUNNING ******** -              04/19 12:58:20    --:--:--             -    -
284149       ********** RUNNING ******** -              04/19 11:50:18    --:--:--             -    -
284159       ********** RUNNING ******** -              04/19 19:16:10    --:--:--             -    -
289904       ********** RUNNING ******** small          04/18 19:59:41    37:40:50             -    2
289909       ********** RUNNING ******** small          04/19 01:02:58    32:37:33             -    2
(...)

[t73XYZ@obcx04 ~]$ pjstat -H
Oakbridge-CX scheduled stop time: 2020/04/24(Fri) 09:00:00 (Remain:  3days 23:19:16)

JOB_ID       JOB_NAME   STATUS  PROJECT  RSCGROUP       START_DATE        ELAPSE           TOKEN NODE
290914       test       END     gt62     lecture        04/18 12:46:24    00:01:44             -    1
290913       test       END     gt62     lecture        04/18 12:46:06    00:02:07             -    1
290915       test       END     gt62     lecture        04/18 12:49:26    00:00:59             -    1
(...)

[t73XYZ@obcx04 ~]$ pjstat --limit
PROJECT     ACCEPT        RUN  BULK_RUN      NODE
gt73        0/  80     0/  20     0/ 256     0/   -
```

Basic/Essential Functions

```
implicit REAL*8 (A-H,O-Z)
include 'mpif.h'
integer :: PETOT, my_rank, ierr

call MPI_INIT      (ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD, PETOT, ierr )
call MPI_COMM_RANK (MPI_COMM_WORLD, my_rank, ierr )

write (*,'(a,2i8)') 'Hello World FORTRAN', my_rank, PETOT

call MPI_FINALIZE (ierr)

stop
end
```

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
    int n, myid, numprocs, i;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    printf ("Hello World %d¥n", myid);
    MPI_Finalize();
}
```

`mpif.h', "mpi.h"

Essential Include file

"use mpi" is possible in F90

MPI_Init

Initialization

MPI_Comm_size

Number of MPI Processes

mpirun -np XX <prog>

MPI_Comm_rank

Process ID starting from 0

MPI_Finalize

Termination of MPI processes

Difference between FORTRAN/C

- (Basically) same interface
  - In C, UPPER/lower cases are considered as different
    - e.g.: **MPI_C**omm_size
      - MPI: UPPER case
      - First character of the function except “MPI_” is in UPPER case.
      - Other characters are in lower case.
- In Fortran, return value `ierr` has to be added at the end of the argument list.
- C needs special types for variables:
  - MPI_Comm, MPI_Datatype, MPI_Op etc.
- **`MPI_INIT`** is different:
  - `call MPI_INIT (ierr)`
  - `MPI_Init (int *argc, char ***argv)`

What's are going on ?

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
    int n, myid, numprocs, i;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    printf ("Hello World %d¥n", myid);
    MPI_Finalize();
}
```

```
#!/bin/sh
#PJM -N "hello"                 Job Name
#PJM -L rscgrp=lecture7         Name of "QUEUE"
#PJM -L node=1                  Node#
#PJM --mpi proc=4               Total MPI Process#
#PJM -L elapse=00:15:00         Computation Time
#PJM -g gt37                    Group Name (Wallet)
#PJM -j
#PJM -e err                     Standard Error
#PJM -o hello.lst               Standard Output

mpiexec.hydra -n ${PJM_MPI_PROC} ./a.out
```

- **`mpiexec.hydra`** starts up 4 MPI processes ("proc=4")
  - A single program runs on four processes.
  - each process writes a value of `myid`
- Four processes do same operations, but values of **`myid`** are different.
- Output of each process is different.
- That is SPMD !

mpi.h, mpif.h

```
implicit REAL*8 (A-H,O-Z)
include 'mpif.h'
integer :: PETOT, my_rank, ierr

call MPI_INIT      (ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD, PETOT, ierr )
call MPI_COMM_RANK (MPI_COMM_WORLD, my_rank, ierr )

write (*,'(a,2i8)') 'Hello World FORTRAN', my_rank, PETOT

call MPI_FINALIZE (ierr)

stop
end
```

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
    int n, myid, numprocs, i;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    printf ("Hello World %d¥n", myid);
    MPI_Finalize();
}
```

- Various types of parameters and variables for MPI & their initial values.
- Name of each var. starts from “MPI_”
- Values of these parameters and variables cannot be changed by users.
- Users do not specify variables starting from “MPI_” in users’ programs.

C

MPI_Init

- Initialize the MPI execution environment (required)
- It is recommended to put this BEFORE all statements in the program.

- **MPI_Init (argc, argv)**

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
    int n, myid, numprocs, i;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    printf ("Hello World %d¥n", myid);
    MPI_Finalize();
}
```

C

MPI_Finalize

- Terminates MPI execution environment (required)
- It is recommended to put this AFTER all statements in the program.
- Please do not forget this.

- **MPI_Finalize ()**

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
    int n, myid, numprocs, i;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    printf ("Hello World %d¥n", myid);
    MPI_Finalize();
}
```

C

MPI_Comm_size

- Determines the size of the group associated with a communicator
- not required, but very convenient function

- **MPI_Comm_size (comm, size)**
  - **comm** MPI_Comm I communicator
  - **size** int O number of processes in the group of communicator

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
    int n, myid, numprocs, i;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    printf ("Hello World %d¥n", myid);
    MPI_Finalize();
}
```

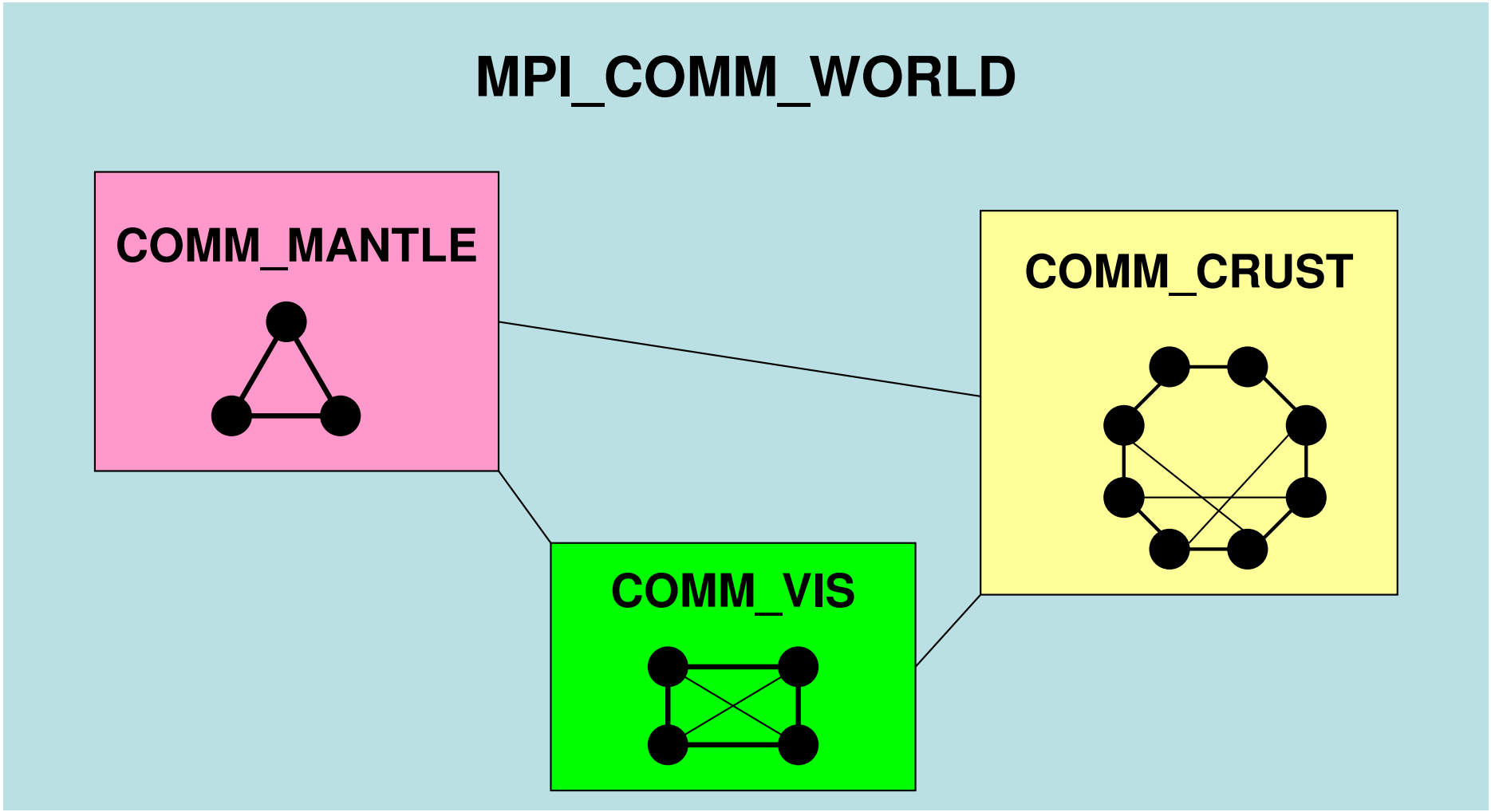
What is Communicator ?

```
MPI_Comm_Size (MPI_COMM_WORLD, PETOT)
```

- Group of processes for communication
- Communicator must be specified in MPI program as a unit of communication
- All processes belong to a group, named "**MPI_COMM_WORLD**" (default)
- Multiple communicators can be created, and complicated operations are possible.
  - Computation, Visualization
- Only "MPI_COMM_WORLD" is needed in this class.

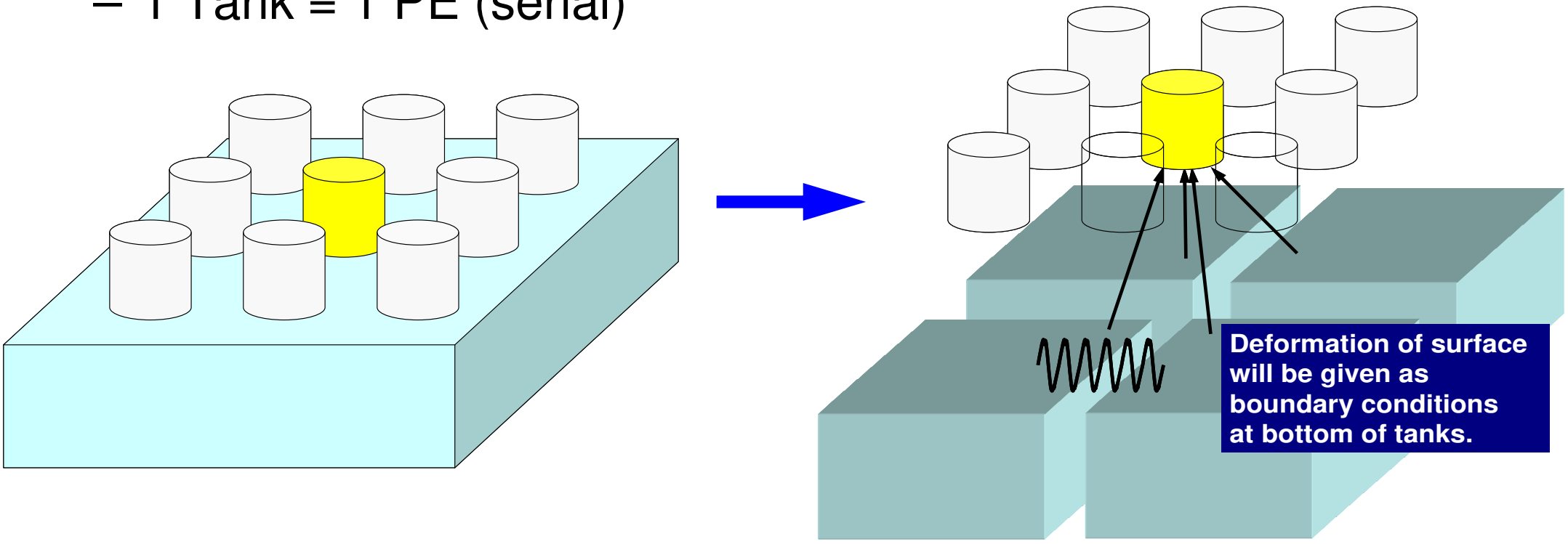
Communicator in MPI

One process can belong to multiple communicators



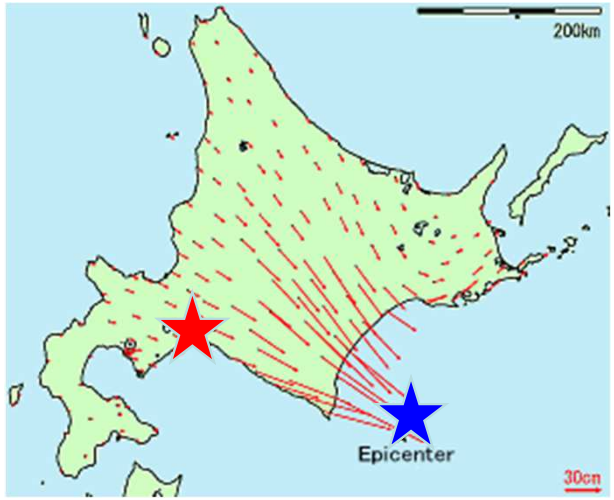
Target Application

- Coupling between “Ground Motion” and “Sloshing of Tanks for Oil-Storage”
  - “One-way” coupling from “Ground Motion” to “Tanks”.
  - Displacement of ground surface is given as forced displacement of bottom surface of tanks.
  - 1 Tank = 1 PE (serial)

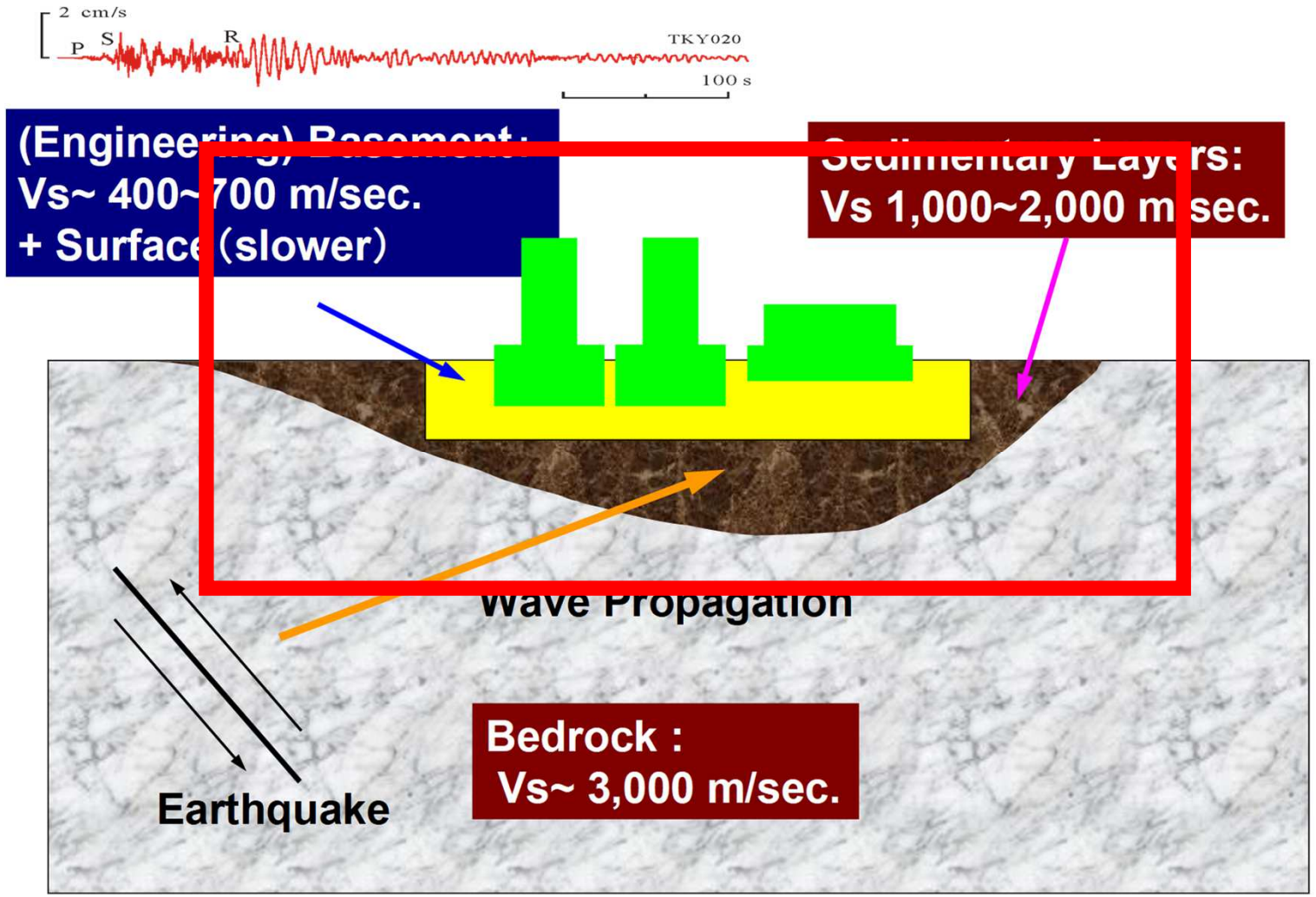


2003 Tokachi Earthquake (M8.0)

Fire accident of oil tanks due to long period
ground motion (surface waves) developed in the basin of Tomakomai

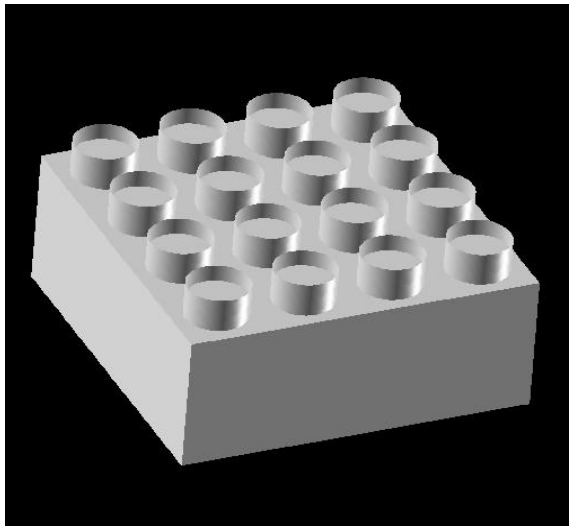


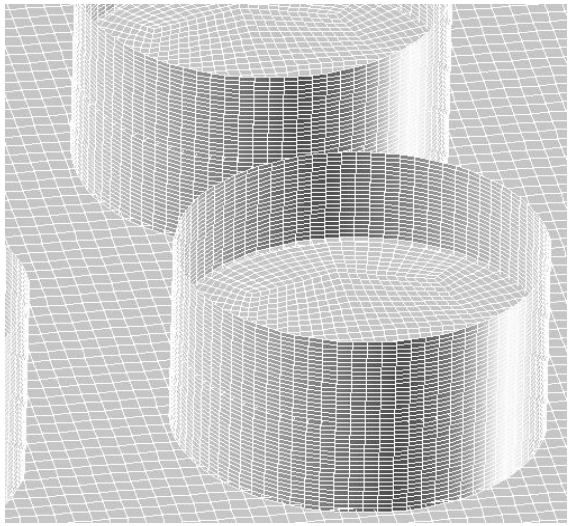
Seismic Wave Propagation, Underground Structure



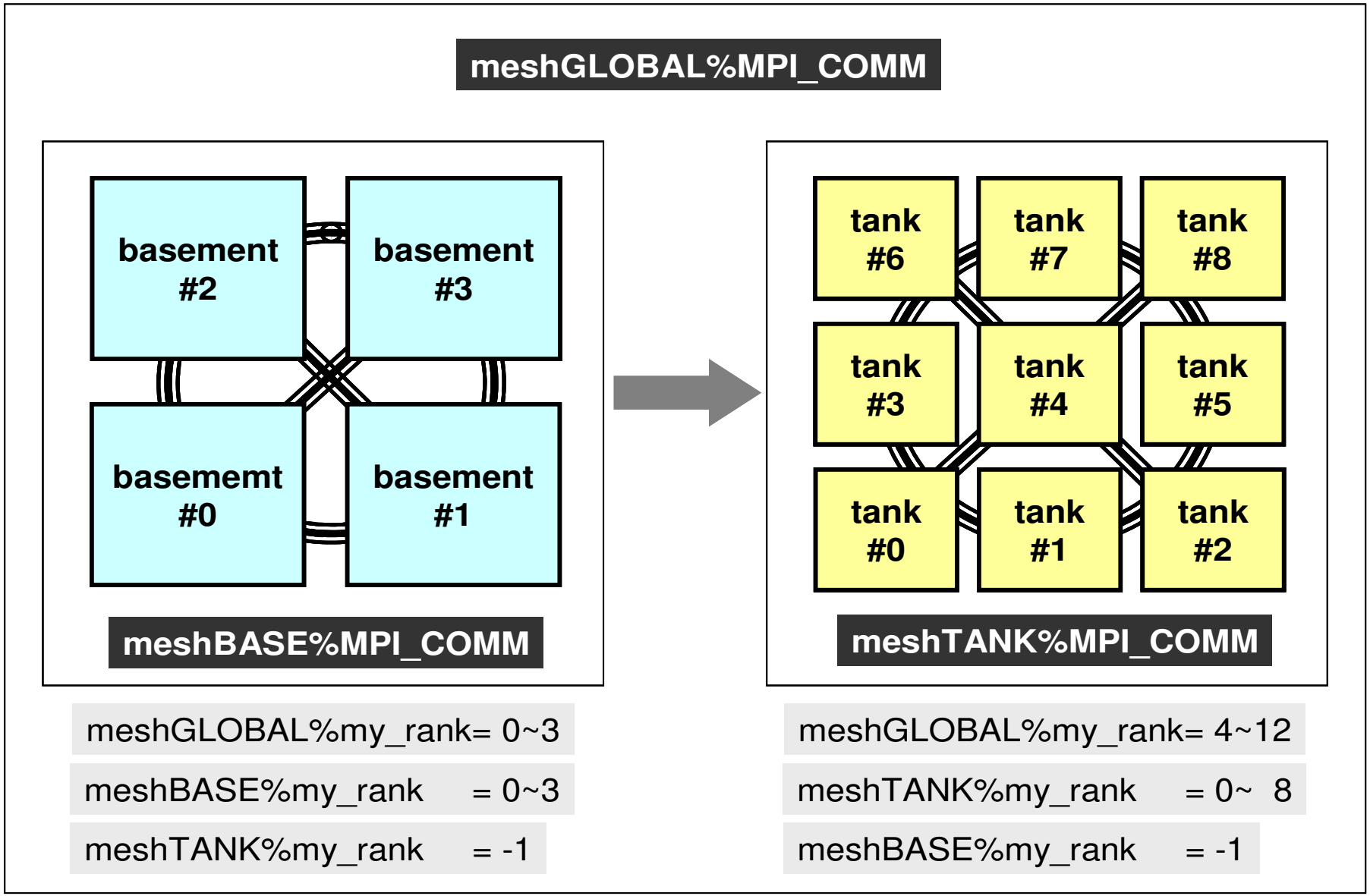
Simulation Codes

- Ground Motion (Ichimura): Fortran
  - Parallel FEM, 3D Elastic/Dynamic
    - Explicit forward Euler scheme
  - Each element: 2m × 2m × 2m cube
  - 240m × 240m × 100m region
- Sloshing of Tanks (Nagashima): C
  - Serial FEM (Embarrassingly Parallel)
    - Implicit backward Euler, Skyline method
    - Shell elements + Inviscid potential flow
  - D: 42.7m, H: 24.9m, T: 20mm,
  - Frequency: 7.6sec.
  - 80 elements in circ., 0.6m mesh in height
  - Tank-to-Tank: 60m, 4 × 4
- Total number of unknowns: 2,918,169

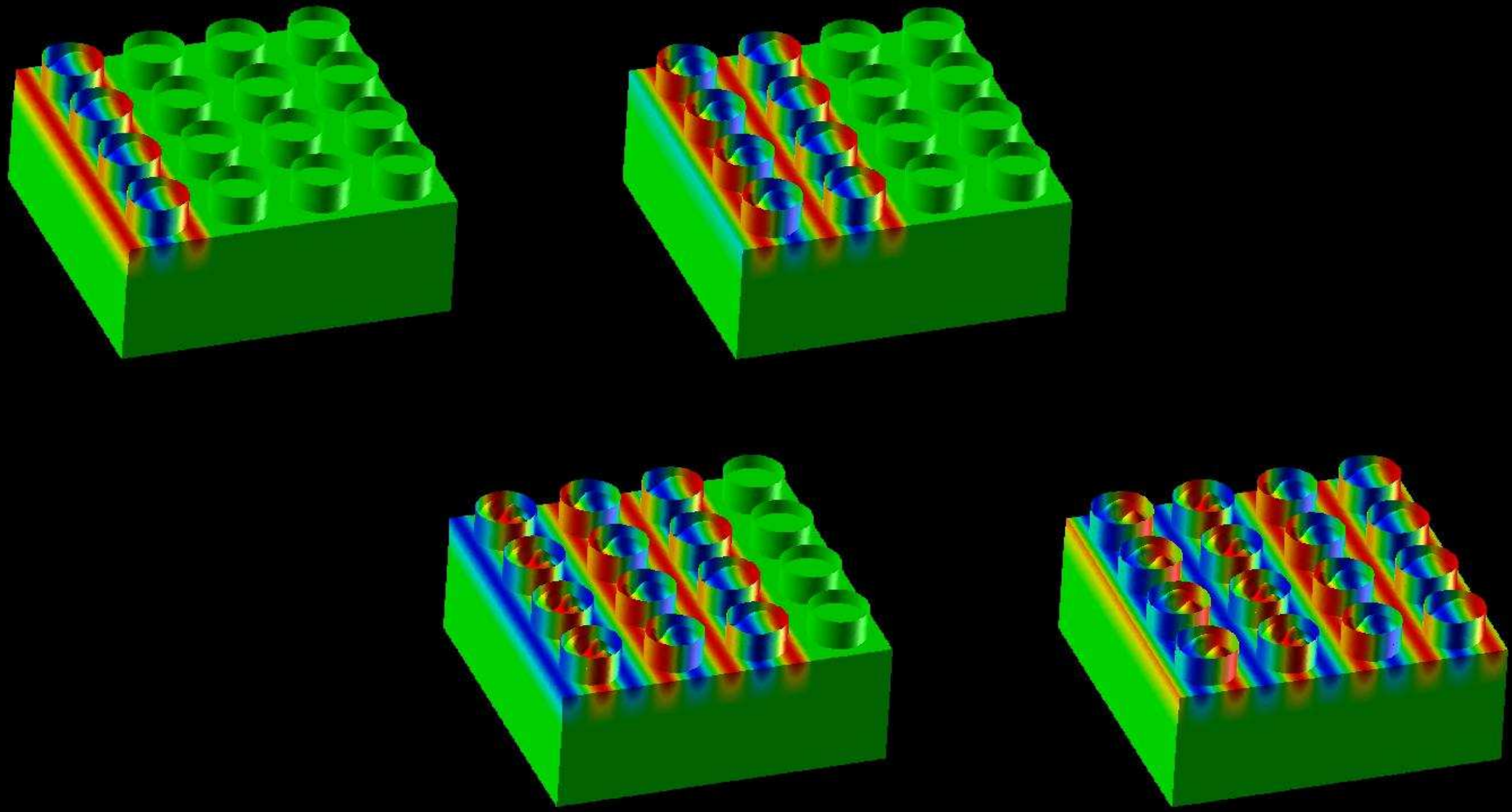


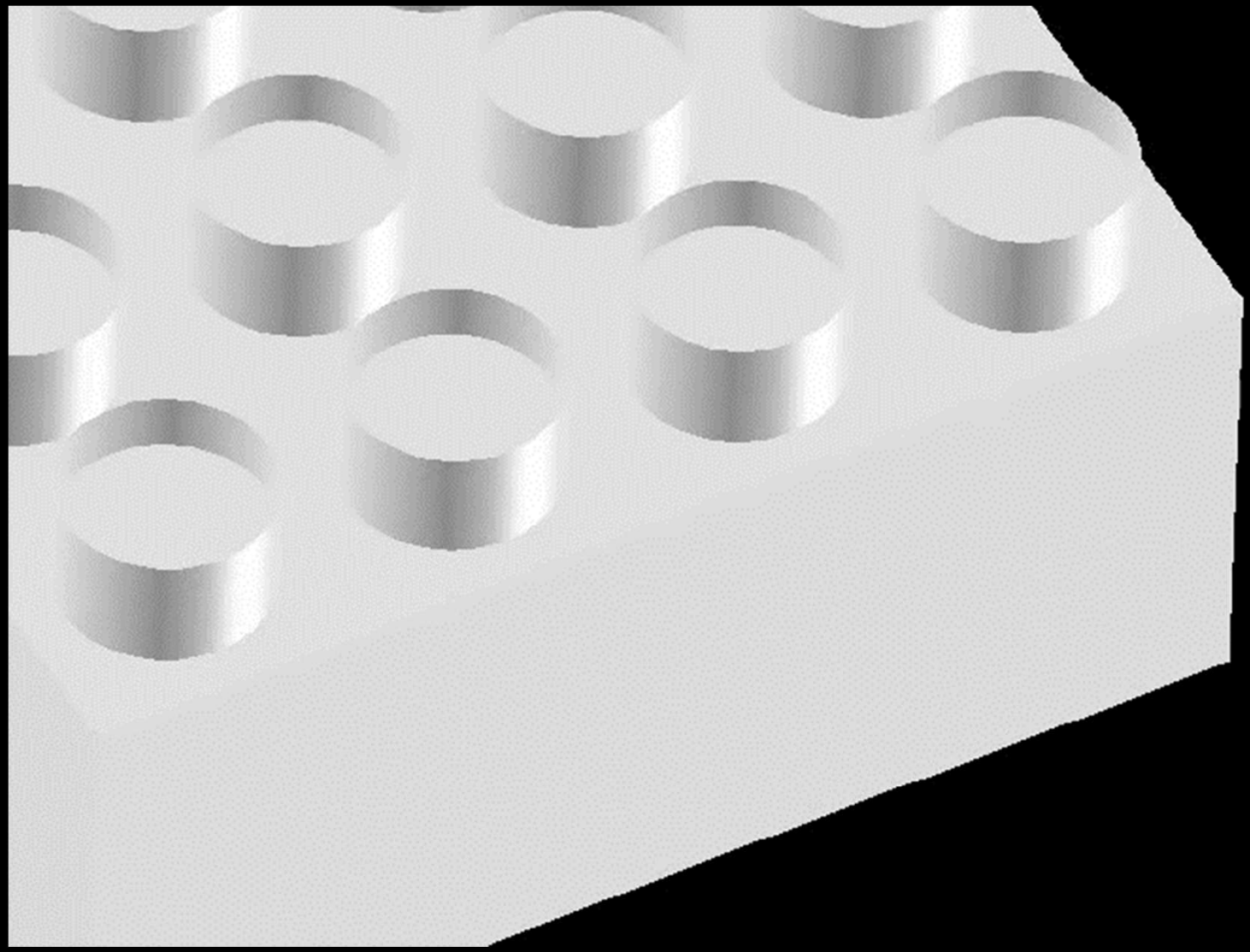


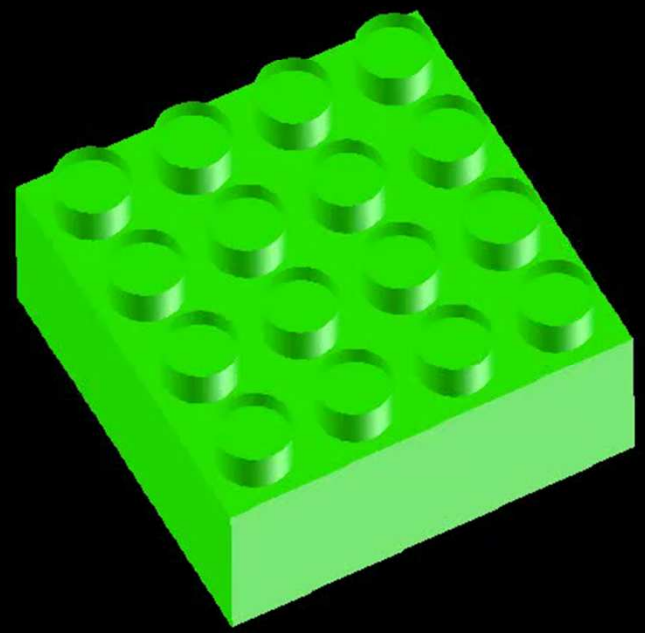
Three Communicators



Coupling between “Ground Motion” and “Sloshing of Tanks for Oil-Storage”







C

MPI_Comm_rank

- Determines the rank of the calling process in the communicator
  - "ID of MPI process" is sometimes called "rank"

- **MPI_Comm_rank (comm, rank)**
  - **comm** MPI_Comm I communicator
  - **rank** int O rank of the calling process in the group of comm
    Starting from "0"

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
    int n, myid, numprocs, i;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    printf ("Hello World %d¥n", myid);
    MPI_Finalize();
}
```

C

MPI_Abort

- Aborts MPI execution environment

- **`MPI_Abort (comm, errcode)`**
  - **`comm`** MPI_Comm I communicator
  - **`errcode`** int O error code

C

MPI_Wtime

- Returns an elapsed time on the calling processor

- **`time= MPI_Wtime ()`**
  - **`time`** `double O` Time in seconds since an arbitrary time in the past.

```
…
double Stime, Etime;

Stime= MPI_Wtime ();

(…)

Etime= MPI_Wtime ();
```

Example of MPI_Wtime

```
$> cd /work/gt73/t73XXX/pFEM/mpi/S1

$> mpiicc -O1 time.c
$> mpiifort -O1 time.f

(modify go4.sh, 4 processes)
$> pjsub go4.sh

     0    1.113281E+00
     3    1.113281E+00
     2    1.117188E+00
     1    1.117188E+00

 Process      Time
   ID
```

go4.sh

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=lecture3
#PJM -L node=1
#PJM --mpi proc=4
#PJM -L elapse=00:15:00
#PJM -g gt73
#PJM -j
#PJM -e err
#PJM -o test.lst

mpiexec.hydra -n ${PJM_MPI_PROC} ./a.out
```

MPI_Wtick

- Returns the resolution of MPI_Wtime
- depends on hardware, and compiler

- **time= MPI_Wtick ()**
  - **time** double O Time in seconds of resolution of MPI_Wtime

```
implicit REAL*8 (A-H,O-Z)
include 'mpif.h'

…
TM= MPI_WTICK ()
write (*,*) TM
…
```

```
double Time;

…
Time = MPI_Wtick();
printf("%5d%16.6E¥n", MyRank, Time);
…
```

Example of MPI_Wtick

```
$> cd /work/gt73/t73XXX/pFEM/mpi/S1

$> mpiicc -O1 wtick.c
$> mpiifort -O1 wtick.f

(modify go1.sh, 1 process)
$> pjsub go1.sh
```

go1.sh

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=lecture3
#PJM -L node=1
#PJM --mpi proc=1
#PJM -L elapse=00:15:00
#PJM -g gt73
#PJM -j
#PJM -e err
#PJM -o test.lst

mpiexec.hydra -n ${PJM_MPI_PROC} ./a.out
```

C

MPI_Barrier

- Blocks until all processes in the communicator have reached this routine.
- Mainly for debugging, huge overhead, not recommended for real code.

- **`MPI_Barrier (comm)`**
  - **`comm`** MPI_Comm I communicator

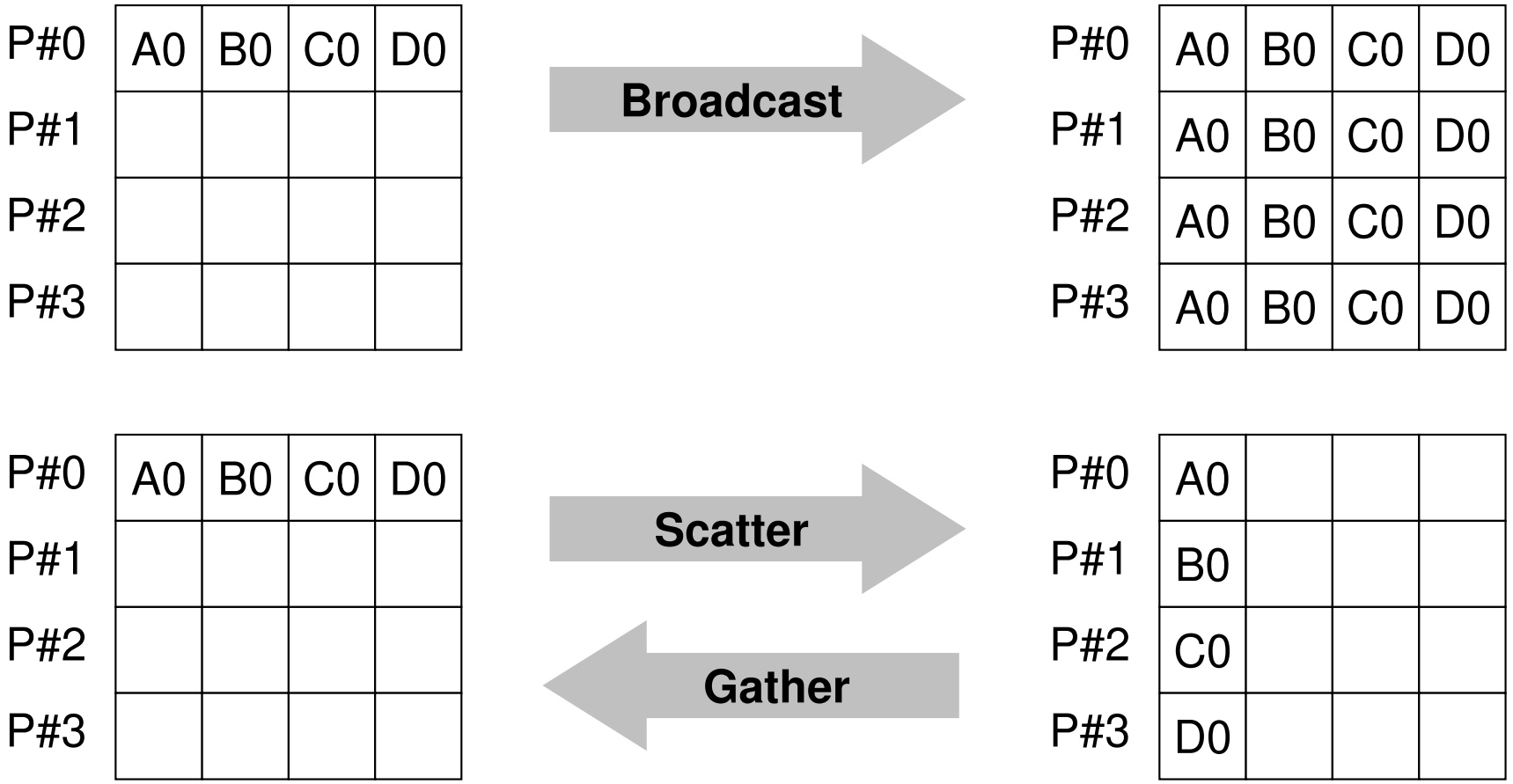
- What is MPI ?
- Your First MPI Program: Hello World
- **Collective Communication**
- Point-to-Point Communication

What is Collective Communication ?
集団通信, グループ通信

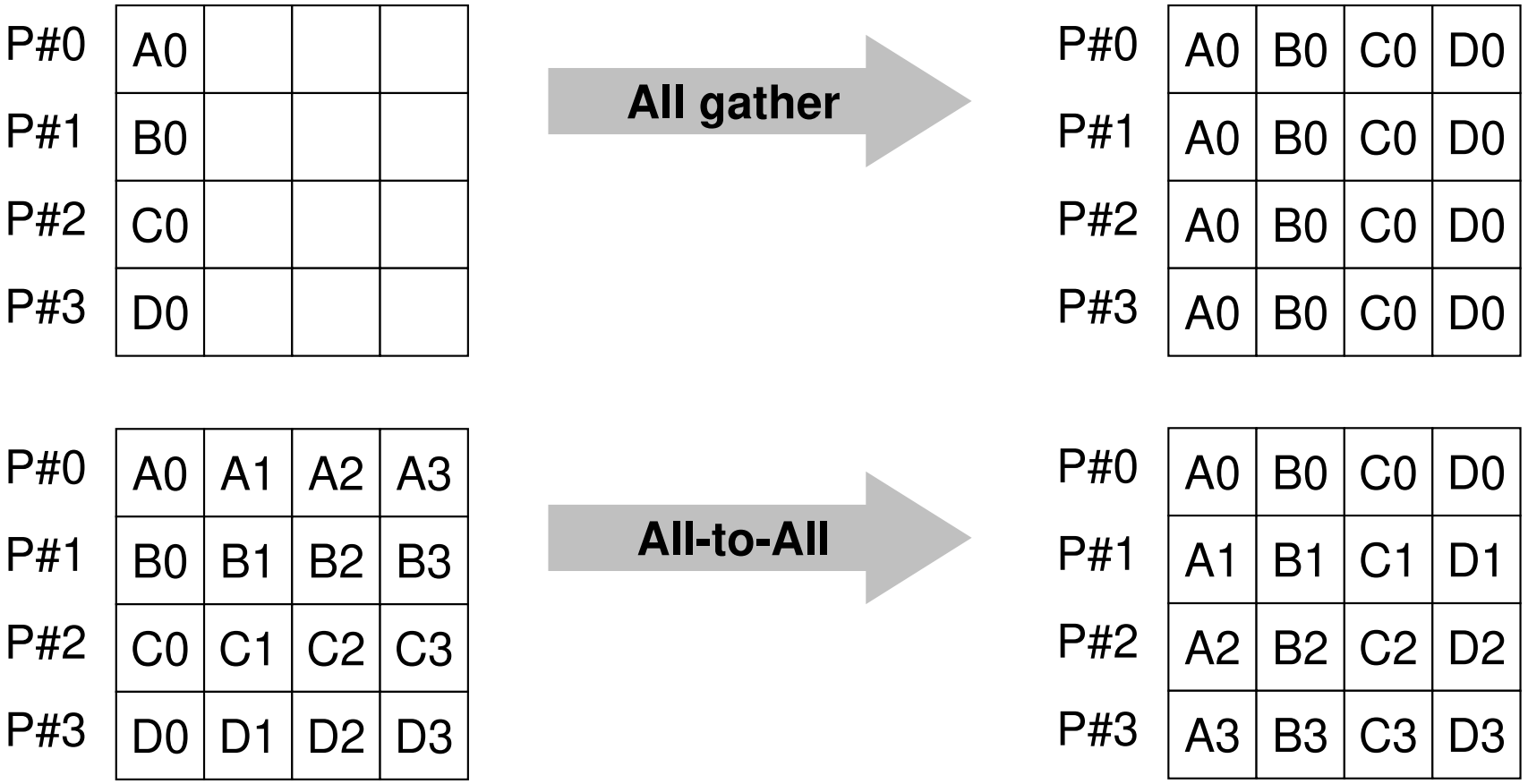
- Collective communication is the process of exchanging information between multiple MPI processes in the communicator: one-to-all or all-to-all communications.

- Examples
  - Broadcasting control data
  - Max, Min
  - Summation
  - Dot products of vectors
  - Transformation of dense matrices

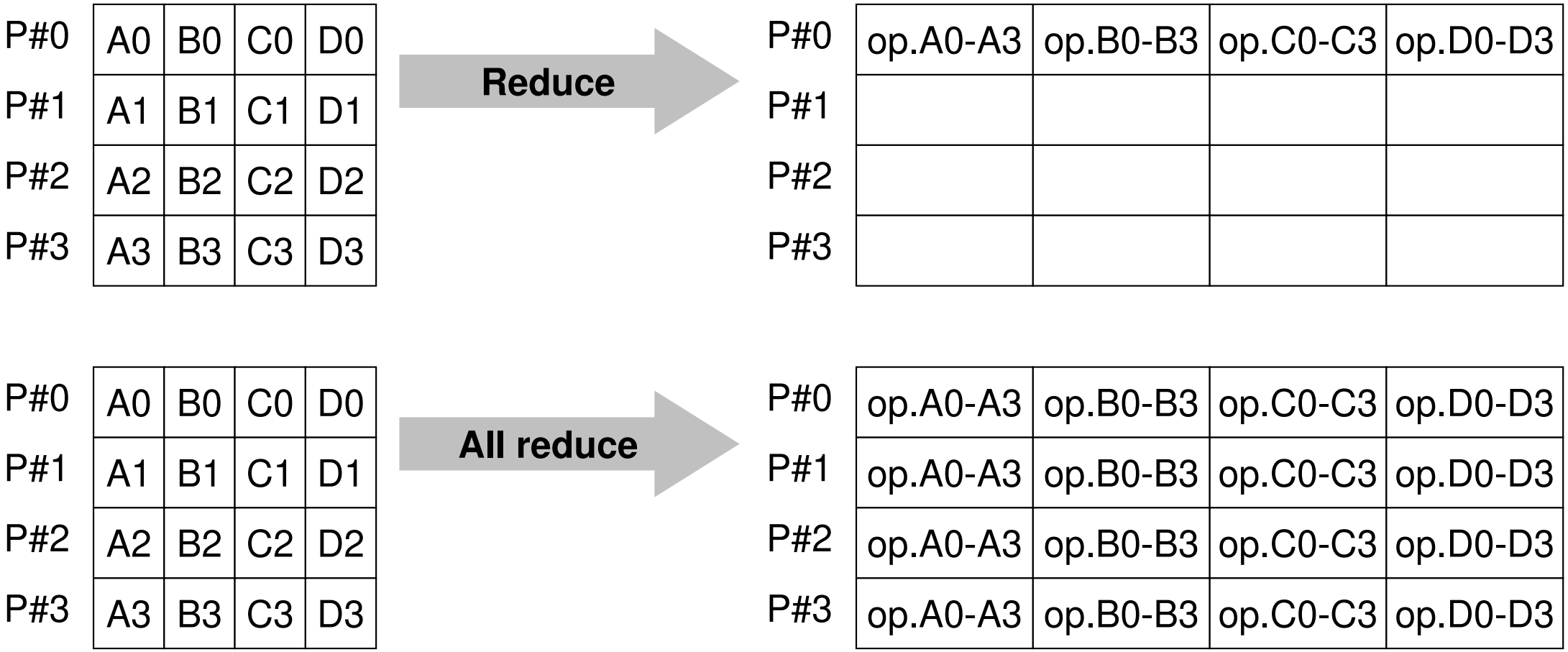
Example of Collective Communications (1/4)



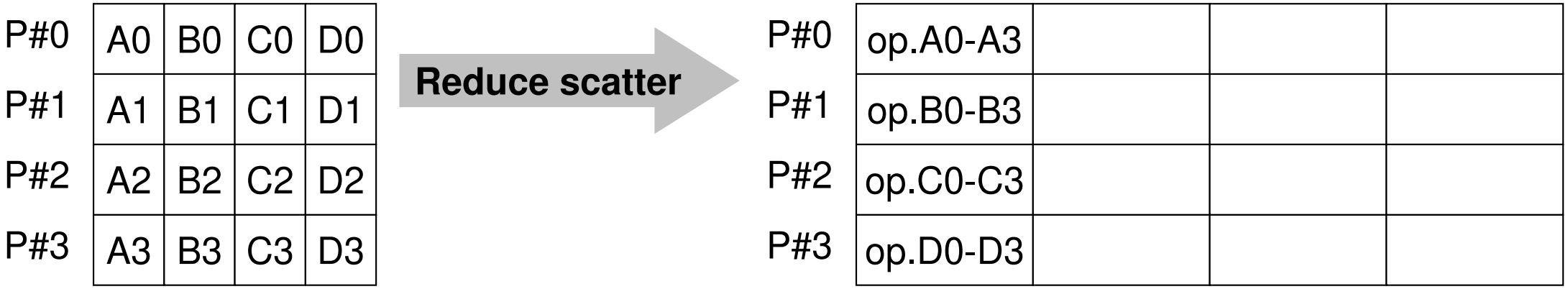
Example of Collective Communications (2/4)



Example of Collective Communications (3/4)



Example of Collective Communications (4/4)



Examples by Collective Comm.

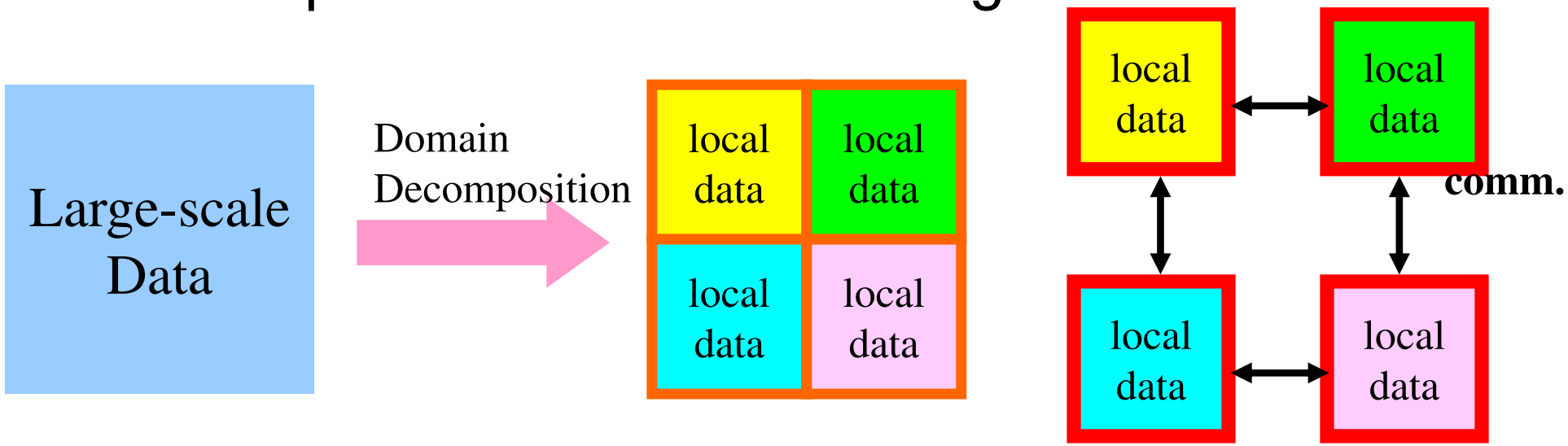
- **Dot Products of Vectors**
- Scatter/Gather
- Reading Distributed Files
- MPI_Allgatherv

Global/Local Data

- Data structure of parallel computing based on SPMD, where large scale “global data” is decomposed to small pieces of “local data”.

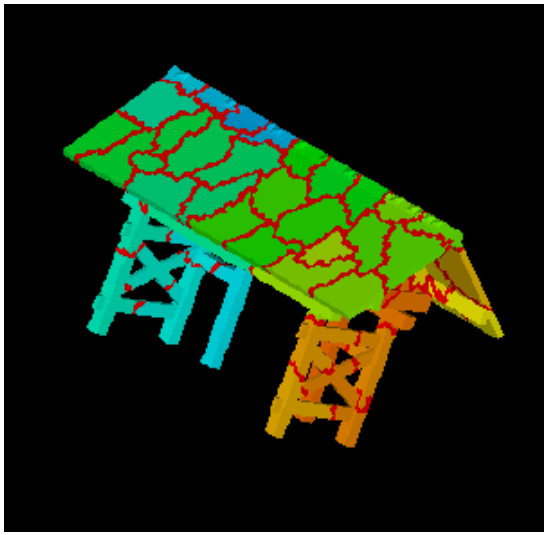
Domain Decomposition/Partitioning

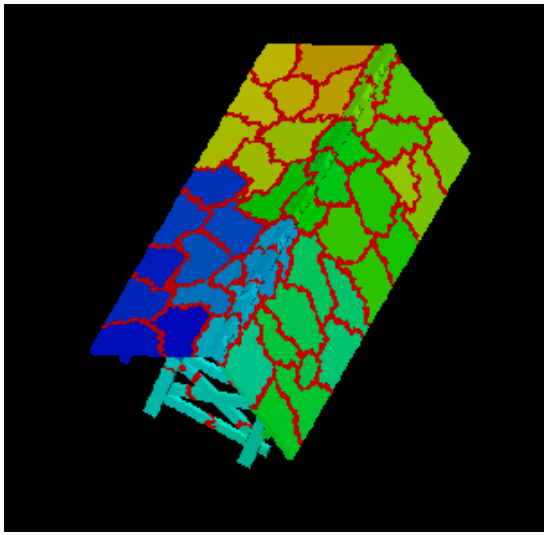
- PC with 1GB RAM: can execute FEM application with up to 10^6 meshes
  - 10^3km × 10^3 km × 10^2 km (SW Japan): 10^8 meshes by 1km cubes
- Large-scale Data: Domain decomposition, parallel & local operations
- Global Computation: Comm. among domains needed



Local Data Structure

- It is important to define proper local data structure for target computation (and its algorithm)
  - Algorithms= Data Structures
- Main objective of this class !





Global/Local Data

- Data structure of parallel computing based on SPMD, where large scale “global data” is decomposed to small pieces of “local data”.

- Consider the dot product of following VECp and VECs with length=20 by parallel computation using 4 processors

```
VECp[ 0]=  2
     [ 1]=  2
     [ 2]=  2
…
     [17]=  2
     [18]=  2
     [19]=  2
```

```
VECs[ 0]=  3
     [ 1]=  3
     [ 2]=  3
…
     [17]=  3
     [18]=  3
     [19]=  3
```

<$O-S1>/dot.f, dot.c

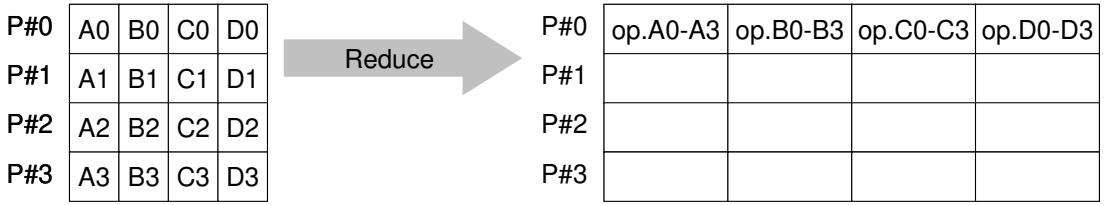
```
implicit REAL*8 (A-H,O-Z)
real(kind=8),dimension(20):: &
    VECp,  VECs

do i= 1, 20
  VECp(i)= 2.0d0
  VECs(i)= 3.0d0
enddo

sum= 0.d0
do ii= 1, 20
  sum= sum + VECp(ii)*VECs(ii)
enddo

stop
end
```

```
#include <stdio.h>
int main(){
   int i;
   double VECp[20], VECs[20]
   double sum;

   for(i=0;i<20;i++){
     VECp[i]= 2.0;
     VECs[i]= 3.0;
   }

   sum = 0.0;
   for(i=0;i<20;i++){
     sum += VECp[i] * VECs[i];
   }
   return 0;
}
```

<$O-S1>/dot.f, dot.c

```
>$ cd /work/gt73/t73XXX/pFEM/mpi/S1

>$ icc dot.c
>$ ifort dot.f

>$ ./a.out

   1          2.          3.
   2          2.          3.
   3          2.          3.
…
  18          2.          3.
  19          2.          3.
  20          2.          3.

dot product       120.
```

MPI_Reduce



- Reduces values on all processes to a single value
  - Summation, Product, Max, Min etc.

- **MPI_Reduce (sendbuf,recvbuf,count,datatype,op,root,comm)**
  - **sendbuf** choice I starting address of send buffer
  - **recvbuf** choice O starting address receive buffer
    type is defined by "**datatype**"
  - **count** int I number of elements in send/receive buffer
  - **datatype** MPI_Datatype I data type of elements of send/recive buffer
    FORTRAN MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_CHARACTER etc.
    C MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR etc
  - **op** MPI_Op I reduce operation
    MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_BAND etc
    Users can define operations by **MPI_OP_CREATE**
  - **root** int I rank of root process
  - **comm** MPI_Comm I communicator

C

Send/Receive Buffer (Sending/Receiving)

- Arrays of “send (sending) buffer” and “receive (receiving) buffer” often appear in MPI.
- Addresses of “send (sending) buffer” and “receive (receiving) buffer” must be different.

Send/Receive Buffer (1/3)
A: Scalar

```
call MPI_REDUCE
(A,recvbuf, 1,datatype,op,root,comm,ierr)

MPI_Reduce
(A,recvbuf,1,datatype,op,root,comm)
```

Send/Receive Buffer (2/3)
A: Array

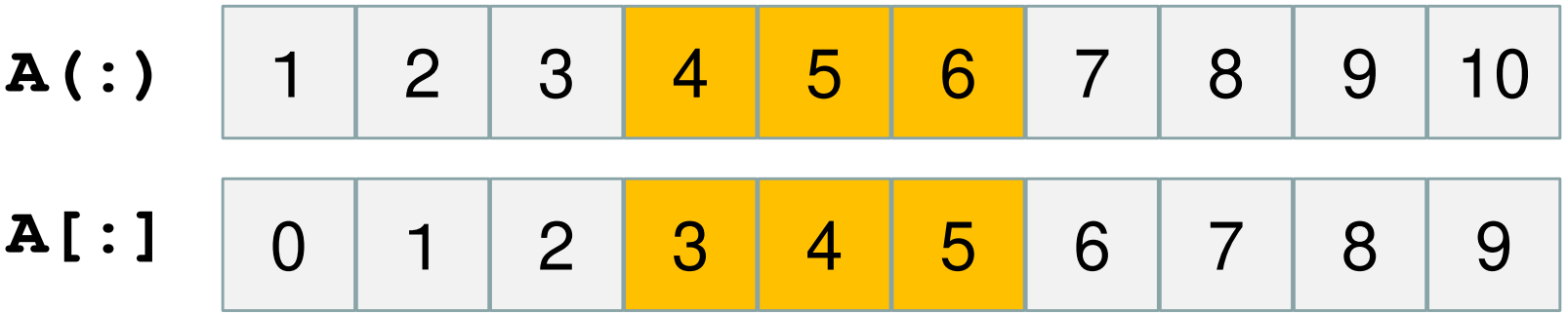
```
call MPI_REDUCE
(A,recvbuf, 3,datatype,op,root,comm,ierr)

MPI_Reduce
(A,recvbuf,3,datatype,op,root,comm)
```

- Starting Address of Send Buffer
  - `A(1)`: Fortran, `A[0]`: C
  - 3 (continuous) components of `A (A(1)-A(3), A[0]-A[2])` are sent

Send/Receive Buffer (3/3) A: Array

```
call MPI_REDUCE
(A(4),recvbuf, 3,datatype,op,root,comm,ierr)

MPI_Reduce
(A[3],recvbuf,3,datatype,op,root,comm)
```

- Starting Address of Send Buffer
  - `A(4)`: Fortran, `A[3]`: C
  - 3 (continuous) components of `A (A(4)-A(6), A[3]-A[5])` are sent



C

Example of MPI_Reduce (1/2)

```
MPI_Reduce
(sendbuf,recvbuf,count,datatype,op,root,comm)
```

```
double X0, X1;

MPI_Reduce
(&X0, &X1, 1, MPI_DOUBLE, MPI_MAX, 0, <comm>);
```

```
double X0[4], XMAX[4];

MPI_Reduce
(X0, XMAX, 4, MPI_DOUBLE, MPI_MAX, 0, <comm>);
```

Global Max values of X0[i] go to XMAX[i] on #0 process（i=0~3）

C

Example of MPI_Reduce (2/2)

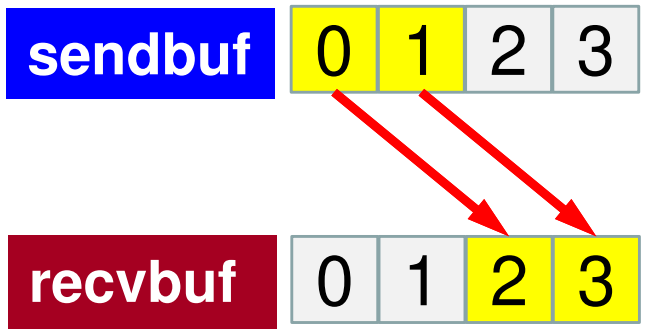
```
MPI_Reduce
(sendbuf,recvbuf,count,datatype,op,root,comm)
```

```
double X0, XSUM;

MPI_Reduce
(&X0, &XSUM, 1, MPI_DOUBLE, MPI_SUM, 0, <comm>)
```

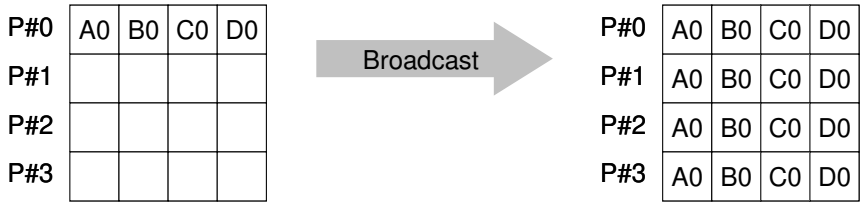
Global summation of X0 goes to XSUM on #0 process.

```
double X0[4];

MPI_Reduce
(&X0[0], &X0[2], 2, MPI_DOUBLE_PRECISION, MPI_SUM, 0, <comm>)
```

- Global summation of X0[0] goes to X0[2] on #0 process.
- Global summation of X0[1] goes to X0[3] on #0 process.



MPI_Bcast



- Broadcasts a message from the process with rank "root" to all other processes of the communicator

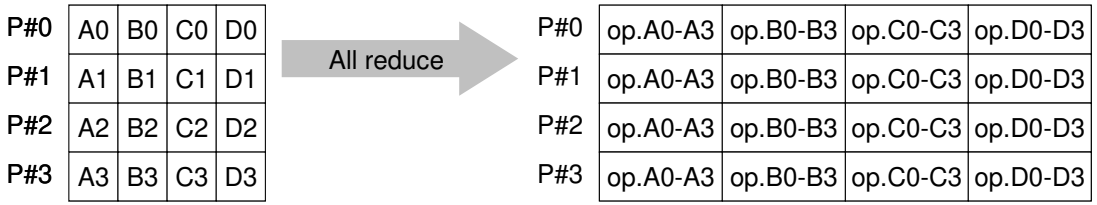
- **MPI_Bcast (buffer,count,datatype,root,comm)**
  - **buffer** choice I/O starting address of buffer
    type is defined by "**datatype**"
  - **count** int I number of elements in send/recv buffer
  - **datatype** MPI_Datatype I data type of elements of send/recv buffer

    FORTRAN MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_CHARACTER etc.

    C MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR etc.
  - **root** int I rank of root process
  - **comm** MPI_Comm I communicator

C

MPI_Allreduce



- MPI_Reduce + MPI_Bcast
- Summation (of dot products) and MAX/MIN values are likely to utilized in each process

- **`call MPI_Allreduce`**
  `(sendbuf,recvbuf,count,datatype,op, comm)`
  - **`sendbuf`** choice I starting address of send buffer
  - **`recvbuf`** choice O starting address receive buffer
    type is defined by **"`datatype`"**
  - **`count`** int I number of elements in send/recv buffer
  - **`datatype`** MPI_Datatype I data type of elements of send/recv buffer
  - **`op`** MPI_Op I reduce operation
  - **`comm`** MPI_Comm I communicator

C

“op” of MPI_Reduce/Allreduce

C

```
MPI_Reduce
(sendbuf,recvbuf,count,datatype,op,root,comm)
```

- **MPI_MAX**, **MPI_MIN** Max, Min
- **MPI_SUM**, **MPI_PROD** Summation, Product
- **MPI_LAND** Logical AND

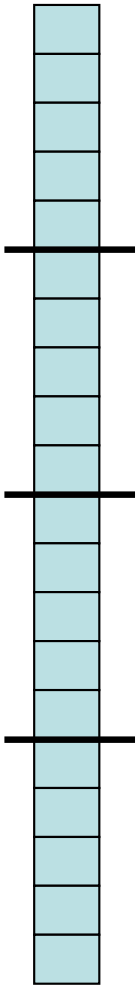
C

Local Data (1/2)

- Decompose vector with length=20 into 4 domains (processes)
- Each process handles a vector with length= 5

```
VECp[ 0]=  2
     [ 1]=  2
     [ 2]=  2
…
     [17]=  2
     [18]=  2
     [19]=  2
```

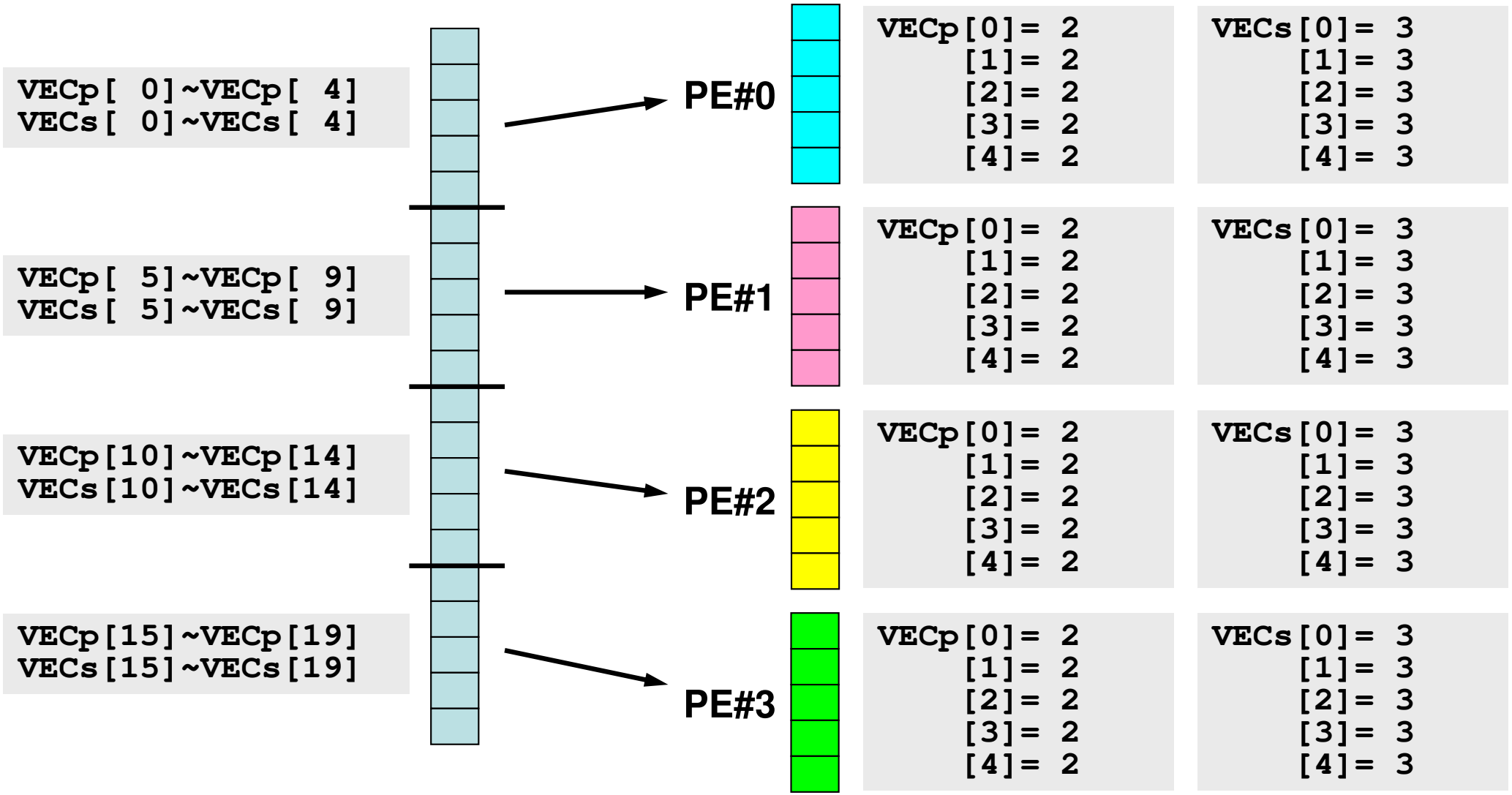
```
VECs[ 0]=  3
     [ 1]=  3
     [ 2]=  3
…
     [17]=  3
     [18]=  3
     [19]=  3
```



C

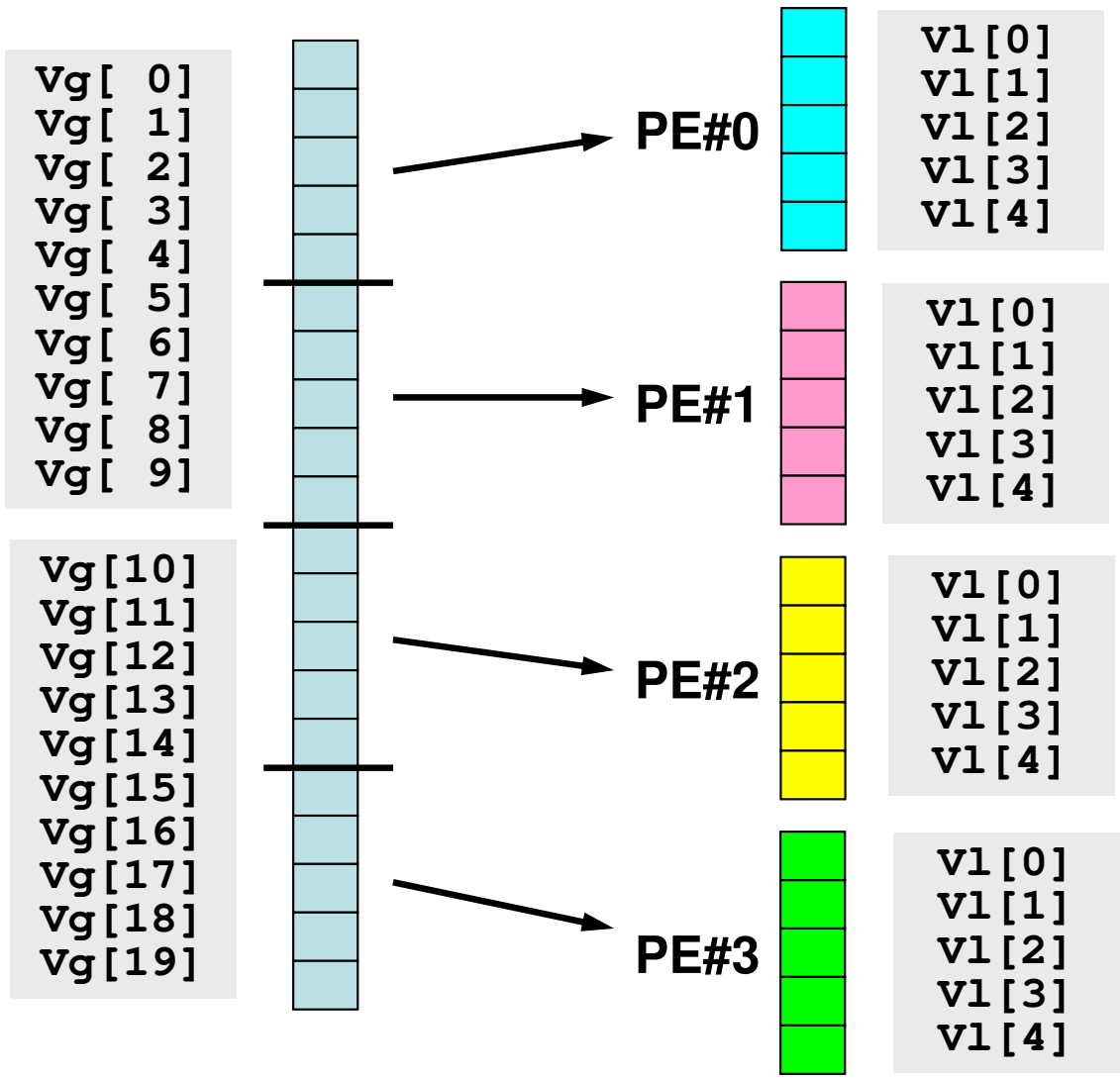
Local Data (2/2)

- 1th-5th components of original global vector go to 1th-5th components of PE#0, 6th-10th -> PE#1, 11th-15th -> PE#2, 16th-20th -> PE#3.



But ...

- It is too easy !! Just decomposing and renumbering from 1 (or 0).
- Of course, this is not enough. Further examples will be shown in the latter part.



Example: Dot Product (1/3)

<$O-S1>/allreduce.c

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int main(int argc, char **argv){
        int i,N;
        int PeTot, MyRank;
        double VECp[5], VECs[5];
        double sumA, sumR, sum0;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &PeTot);
        MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);

        sumA= 0.0;
        sumR= 0.0;

        N=5;
        for(i=0;i<N;i++){
          VECp[i] = 2.0;
          VECs[i] = 3.0;
        }

        sum0 = 0.0;
        for(i=0;i<N;i++){
                sum0 += VECp[i] * VECs[i];
        }
```

Local vector is generated at each local process.

Example: Dot Product (2/3)

<$O-S1>/allreduce.c

```
        MPI_Reduce(&sum0, &sumR, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
        MPI_Allreduce(&sum0, &sumA, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
        printf("before BCAST %5d %15.0F %15.0F¥n", MyRank, sumA, sumR);

        MPI_Bcast(&sumR, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        printf("after  BCAST %5d %15.0F %15.0F¥n", MyRank, sumA, sumR);

        MPI_Finalize();

        return 0;
}
```

Example: Dot Product (3/3)

<$O-S1>/allreduce.c

```
MPI_Reduce(&sum0, &sumR, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

MPI_Allreduce(&sum0, &sumA, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

Dot Product
Summation of results of each process (sum0)
"sumR" has value only on PE#0.

"sumA" has value on all processes by MPI_Allreduce

```
MPI_Bcast(&sumR, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

"sumR" has value on PE#1-#3 by MPI_Bcast

Execute <$O-S1>/allreduce.f/c

```
$> cd /work/gt73/t73XXX/pFEM/mpi/S1
$> mpiifort -align array64byte -O3 -axCORE-AVX512 allreduce.f
$> mpiicc -align -O3 -axCORE-AVX512 allreduce.c

(modify go4.sh, 4-processes)
$> pjsub go4.sh

(my_rank, sumALLREDUCE,sumREDUCE)
before BCAST    0    1.200000E+02    1.200000E+02
after  BCAST    0    1.200000E+02    1.200000E+02

before BCAST    1    1.200000E+02    0.000000E+00
after  BCAST    1    1.200000E+02    1.200000E+02

before BCAST    3    1.200000E+02    0.000000E+00
after  BCAST    3    1.200000E+02    1.200000E+02

before BCAST    2    1.200000E+02    0.000000E+00
after  BCAST    2    1.200000E+02    1.200000E+02
```

Examples by Collective Comm.

- Dot Products of Vectors
- **Scatter/Gather**
- Reading Distributed Files
- MPI_Allgatherv

Global/Local Data (1/3)

- Parallelization of an easy process where a real number α is added to each component of real vector **VECg**:

```
      do i= 1, NG
        VECg(i)= VECg(i) + ALPHA
      enddo
```

```
for (i=0; i<NG; i++{
    VECg[i]= VECg[i] + ALPHA
}
```

Global/Local Data (2/3)

- Configuration
  - **NG= 32 (length of the vector)**
  - **ALPHA=1000.**
  - Process # of MPI= 4

- Vector VECg has following 32 components (<$O-S1>/a1x.all):

```
(101.0,  103.0,  105.0,  106.0,  109.0,  111.0,  121.0,  151.0,
 201.0,  203.0,  205.0,  206.0,  209.0,  211.0,  221.0,  251.0,
 301.0,  303.0,  305.0,  306.0,  309.0,  311.0,  321.0,  351.0,
 401.0,  403.0,  405.0,  406.0,  409.0,  411.0,  421.0,  451.0)
```

Global/Local Data (3/3)

- Procedure
  - ① Reading vector **VECg** with length=32 from one process (*e.g.* 0th process)
    - Global Data
  - ② Distributing vector components to 4 MPI processes equally (*i.e.* length= 8 for each processes)
    - Local Data, Local ID/Numbering
  - ③ Adding **ALPHA** to each component of the local vector (with length= 8) on each process.
  - ④ Merging the results to global vector with length= 32.

- Actually, we do not need parallel computers for such a kind of small computation.

Operations of Scatter/Gather (1/8)

Reading **VECg** (length=32) from a process (*e.g.* #0)

- Reading global data from #0 process

```
include   'mpif.h'
integer,  parameter :: NG= 32
real(kind=8), dimension(NG):: VECg

call MPI_INIT (ierr)
call MPI_COMM_SIZE (<comm>, PETOT  , ierr)
call MPI_COMM_RANK (<comm>, my_rank, ierr)

if (my_rank.eq.0) then
  open (21, file= 'a1x.all', status= 'unknown')
  do i= 1, NG
    read (21,*) VECg(i)
  enddo
  close (21)
endif
```

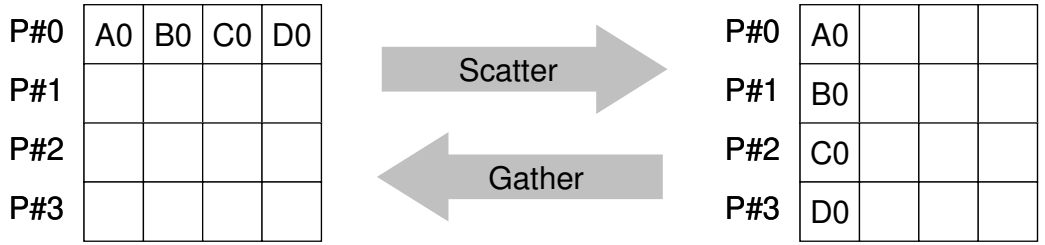
```
#include <mpi.h>
#include <stdio.h>
#include <math.h>
#include <assert.h>

int main(int argc, char **argv){
  int i, NG=32;
  int PeTot, MyRank, MPI_Comm;
  double VECg[32];
  char filename[80];
  FILE *fp;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(<comm>, &PeTot);
  MPI_Comm_rank(<comm>, &MyRank);

        fp = fopen("a1x.all", "r");
        if(!MyRank) for(i=0;i<NG;i++){
                fscanf(fp, "%lf", &VECg[i]);
        }
```

Operations of Scatter/Gather (2/8)

Distributing global data to 4 process equally (*i.e.* length=8 for each process)

- MPI_Scatter

MPI_Scatter



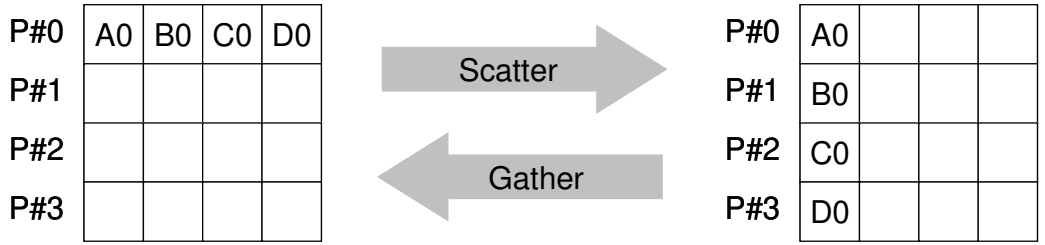
- Sends data from one process to all other processes in a communicator
  - `scount`-size messages are sent to each process

- **`MPI_Scatter (sendbuf, scount, sendtype, recvbuf, rcount, recvtype, root, comm)`**
  - **`sendbuf`** choice I — starting address of sending buffer
    type is defined by "**`datatype`**"
  - **`scount`** int I — number of elements sent to each process
  - **`sendtype`** MPI_Datatype I — data type of elements of sending buffer

    ```
    FORTRAN  MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_CHARACTER etc.
    C        MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR etc.
    ```
  - **`recvbuf`** choice O — starting address of receiving buffer
  - **`rcount`** int I — number of elements received from the root process
  - **`recvtype`** MPI_Datatype I — data type of elements of receiving buffer
  - **`root`** int I — rank of root process
  - **`comm`** MPI_Comm I — communicator

C

MPI_Scatter (cont.)



- **MPI_Scatter (sendbuf, scount, sendtype, recvbuf, rcount, recvtype, root, comm)**
  - **sendbuf** choice I starting address of sending buffer
  - **scount** int I number of elements sent to each process
  - **sendtype** MPI_Datatype I data type of elements of sending buffer
  - **recvbuf** choice O starting address of receiving buffer
  - **rcount** int I number of elements received from the root process
  - **recvtype** MPI_Datatype I data type of elements of receiving buffer
  - **root** int I rank of root process
  - **comm** MPI_Comm I communicator

- Usually
  - **scount = rcount**
  - **sendtype= recvtype**
- This function sends **scount** components starting from **sendbuf** (sending buffer) at process **#root** to each process in **comm**. Each process receives **rcount** components starting from **recvbuf** (receiving buffer).

C

Operations of Scatter/Gather (3/8)

Distributing global data to 4 processes equally

- Allocating receiving buffer **VEC** (length=8) at each process.
- 8 components sent from sending buffer **VECg** of process #0 are received at each process #0-#3 as 1st-8th components of receiving buffer **VEC**.

```
integer, parameter :: N =  8
real(kind=8), dimension(N ) :: VEC
...
call MPI_Scatter                              &
     (VECg, N, MPI_DOUBLE_PRECISION, &
      VEC , N, MPI_DOUBLE_PRECISION, &
      0, <comm>, ierr)
```

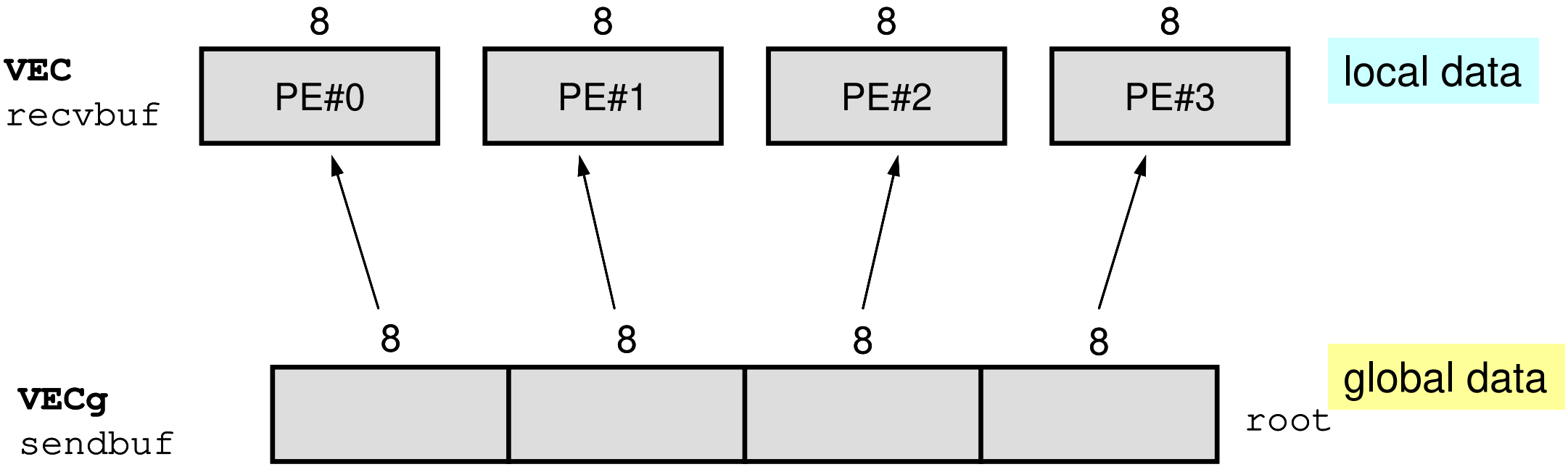
```
int  N=8;
double VEC [8];
...
MPI_Scatter  (VECg,  N,  MPI_DOUBLE,  VEC,  N,
MPI_DOUBLE, 0, <comm>);
```

```
call MPI_SCATTER
(sendbuf, scount, sendtype, recvbuf, rcount,
 recvtype, root, comm, ierr)
```

Operations of Scatter/Gather (4/8)

Distributing global data to 4 processes equally

- 8 components are *scattered* to each process from root (#0)
- 1st-8th components of **VECg** are stored as 1st-8th ones of **VEC** at **#0**, 9th-16th components of **VECg** are stored as 1st-8th ones of **VEC** at **#1**, etc.
  - **VECg**: Global Data, **VEC**: Local Data



Operations of Scatter/Gather (5/8)

Distributing global data to 4 processes equally

- Global Data: 1st-32nd components of **VECg** at **#0**
- Local Data: 1st-8th components of **VEC** at each process
- Each component of **VEC** can be written from each process in the following way:

```
do i= 1, N
  write (*,'(a, 2i8,f10.0)') 'before',  my_rank, i, VEC(i)
enddo
```

```
for(i=0;i<N;i++){
printf("before %5d %5d %10.0F¥n", MyRank, i+1, VEC[i]);}
```

Operations of Scatter/Gather (5/8)

Distributing global data to 4 processes equally

- Global Data: 1st-32nd components of **VECg** at **#0**
- Local Data: 1st-8th components of **VEC** at each process
- Each component of **VEC** can be written from each process in the following way:

PE#0
```
before 0 1   101.
before 0 2   103.
before 0 3   105.
before 0 4   106.
before 0 5   109.
before 0 6   111.
before 0 7   121.
before 0 8   151.
```

PE#1
```
before 1 1   201.
before 1 2   203.
before 1 3   205.
before 1 4   206.
before 1 5   209.
before 1 6   211.
before 1 7   221.
before 1 8   251.
```

PE#2
```
before 2 1   301.
before 2 2   303.
before 2 3   305.
before 2 4   306.
before 2 5   309.
before 2 6   311.
before 2 7   321.
before 2 8   351.
```

PE#3
```
before 3 1   401.
before 3 2   403.
before 3 3   405.
before 3 4   406.
before 3 5   409.
before 3 6   411.
before 3 7   421.
before 3 8   451.
```

Operations of Scatter/Gather (6/8)

On each process, **ALPHA** is added to each of 8 components of **VEC**

- On each process, computation is in the following way

```
real(kind=8), parameter :: ALPHA= 1000.
do i= 1, N
  VEC(i)= VEC(i) + ALPHA
enddo
```

```
double ALPHA=1000.;
...
for(i=0;i<N;i++){
  VEC[i]= VEC[i] + ALPHA;}
```

- Results:

PE#0
```
after 0 1  1101.
after 0 2  1103.
after 0 3  1105.
after 0 4  1106.
after 0 5  1109.
after 0 6  1111.
after 0 7  1121.
after 0 8  1151.
```

PE#1
```
after 1 1  1201.
after 1 2  1203.
after 1 3  1205.
after 1 4  1206.
after 1 5  1209.
after 1 6  1211.
after 1 7  1221.
after 1 8  1251.
```

PE#2
```
after 2 1  1301.
after 2 2  1303.
after 2 3  1305.
after 2 4  1306.
after 2 5  1309.
after 2 6  1311.
after 2 7  1321.
after 2 8  1351.
```

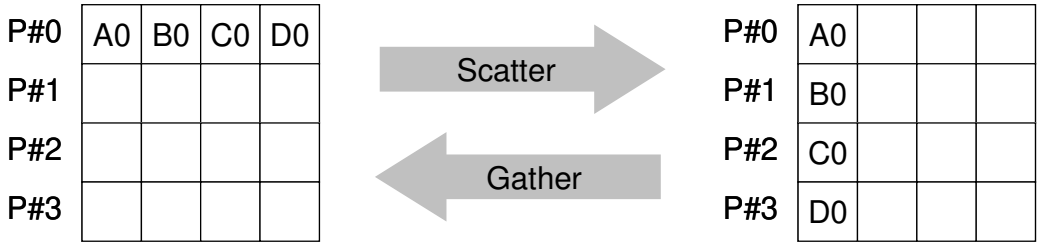
PE#3
```
after 3 1  1401.
after 3 2  1403.
after 3 3  1405.
after 3 4  1406.
after 3 5  1409.
after 3 6  1411.
after 3 7  1421.
after 3 8  1451.
```

Operations of Scatter/Gather (7/8)

Merging the results to global vector with length= 32

- Using MPI_Gather (inverse operation to MPI_Scatter)

MPI_Gather



- Gathers together values from a group of processes, inverse operation to MPI_Scatter

- **`MPI_Gather (sendbuf, scount, sendtype, recvbuf, rcount, recvtype, root, comm)`**
  - **`sendbuf`** choice I starting address of sending buffer
  - **`scount`** int I number of elements sent to each process
  - **`sendtype`** MPI_Datatype I data type of elements of sending buffer
  - **`recvbuf`** choice O starting address of receiving buffer
  - **`rcount`** int I number of elements received from the root process
  - **`recvtype`** MPI_Datatype I data type of elements of receiving buffer
  - **`root`** int I rank of root process
  - **`comm`** MPI_Comm I communicator

- **`recvbuf`** is on **`root`** process.

C

Operations of Scatter/Gather (8/8)

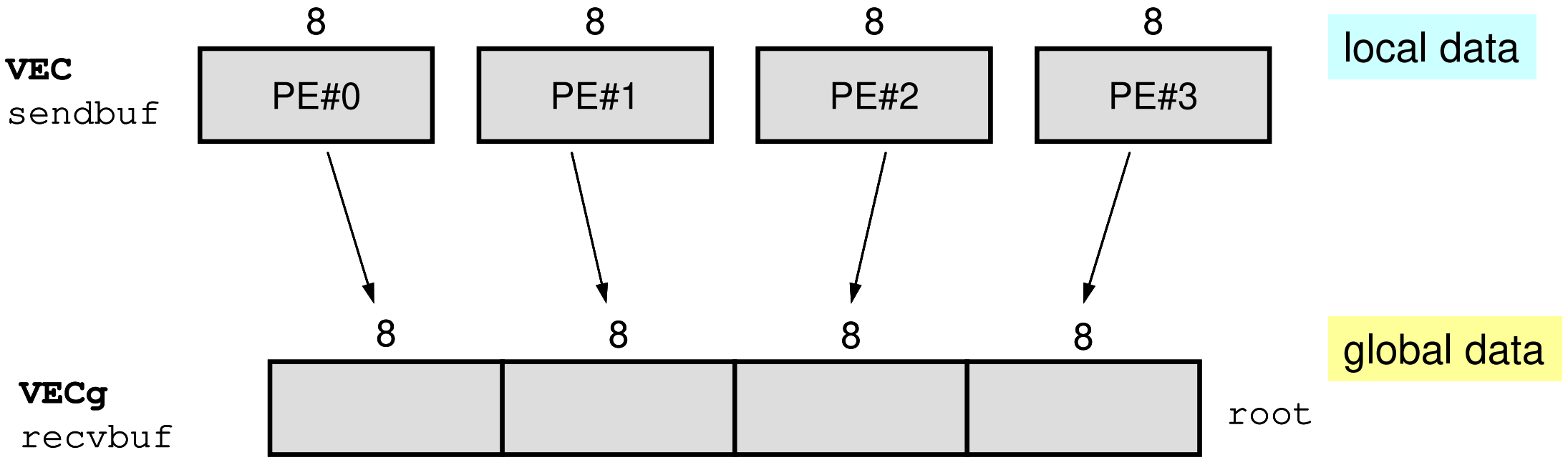
Merging the results to global vector with length= 32

- Each process components of **VEC** to **VECg** on root (#0 in this case).

```
call MPI_Gather                                    &
     (VEC , N, MPI_DOUBLE_PRECISION, &
      VECg, N, MPI_DOUBLE_PRECISION, &
      0, <comm>, ierr)
```

```
MPI_Gather (VEC, N, MPI_DOUBLE, VECg, N,
MPI_DOUBLE, 0, <comm>);
```

- 8 components are gathered from each process to the root process.



<$O-S1>/scatter-gather.f/c

```
$> cd /work/gt73/t73XXX/pFEM/mpi/S1
$> mpiifort -align array64byte -O3 -axCORE-AVX512 scatter-gather.f
$> mpiicc -align -O3 -axCORE-AVX512 scatter-gather.c

(modify go4.sh, 4-processes)
$> pjsub go4.sh
```

```
PE#0
before 0 1   101.
before 0 2   103.
before 0 3   105.
before 0 4   106.
before 0 5   109.
before 0 6   111.
before 0 7   121.
before 0 8   151.
```

```
PE#1
before 1 1   201.
before 1 2   203.
before 1 3   205.
before 1 4   206.
before 1 5   209.
before 1 6   211.
before 1 7   221.
before 1 8   251.
```

```
PE#2
before 2 1   301.
before 2 2   303.
before 2 3   305.
before 2 4   306.
before 2 5   309.
before 2 6   311.
before 2 7   321.
before 2 8   351.
```

```
PE#3
before 3 1   401.
before 3 2   403.
before 3 3   405.
before 3 4   406.
before 3 5   409.
before 3 6   411.
before 3 7   421.
before 3 8   451.
```

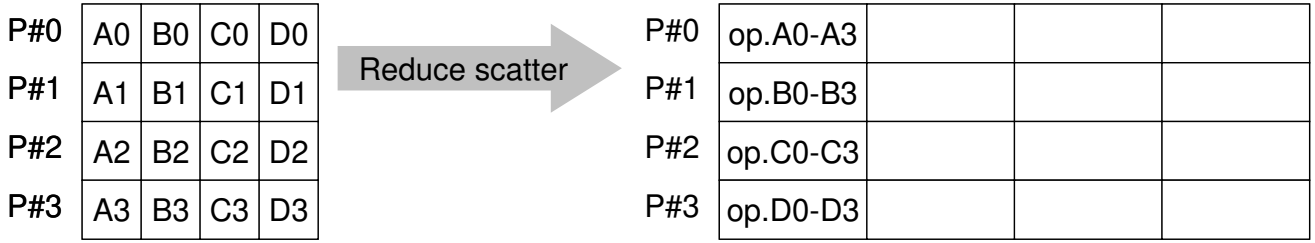
```
PE#0
after 0 1   1101.
after 0 2   1103.
after 0 3   1105.
after 0 4   1106.
after 0 5   1109.
after 0 6   1111.
after 0 7   1121.
after 0 8   1151.
```

```
PE#1
after 1 1   1201.
after 1 2   1203.
after 1 3   1205.
after 1 4   1206.
after 1 5   1209.
after 1 6   1211.
after 1 7   1221.
after 1 8   1251.
```

```
PE#2
after 2 1   1301.
after 2 2   1303.
after 2 3   1305.
after 2 4   1306.
after 2 5   1309.
after 2 6   1311.
after 2 7   1321.
after 2 8   1351.
```

```
PE#3
after 3 1   1401.
after 3 2   1403.
after 3 3   1405.
after 3 4   1406.
after 3 5   1409.
after 3 6   1411.
after 3 7   1421.
after 3 8   1451.
```

MPI_Reduce_scatter



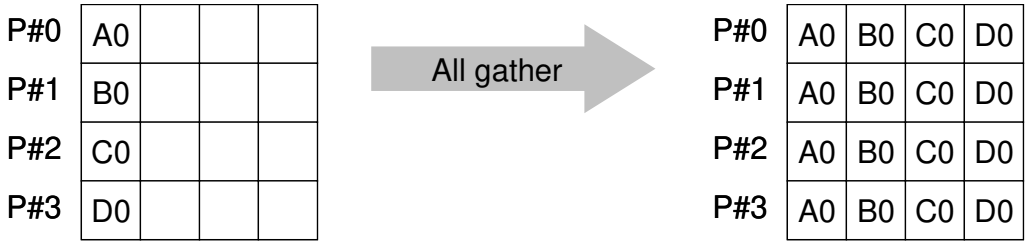
- MPI_Reduce + MPI_Scatter

- **MPI_Reduce_Scatter (sendbuf, recvbuf, rcount, datatype, op, comm)**
  - **sendbuf** choice I starting address of sending buffer
  - **recvbuf** choice O starting address of receiving buffer
  - **rcount** int I integer array specifying the number of elements in result distributed to each process. Array must be identical on all calling processes.
  - **datatype** MPI_Datatype I data type of elements of sending/receiving buffer
  - **op** MPI_Op I reduce operation

    MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_BAND etc
  - **comm** MPI_Comm I communicator

C

MPI_Allgather

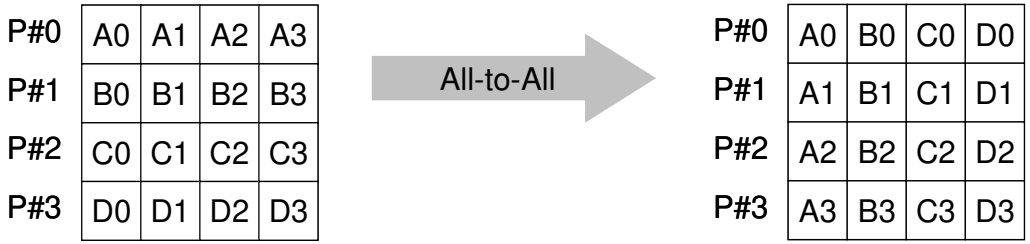


- MPI_Gather+MPI_Bcast
  - Gathers data from all tasks and distribute the combined data to all tasks

- **`MPI_Allgather (sendbuf, scount, sendtype, recvbuf, rcount, recvtype, comm)`**
  - **`sendbuf`** choice I — starting address of sending buffer
  - **`scount`** int I — number of elements sent to each process
  - **`sendtype`** MPI_Datatype I — data type of elements of sending buffer
  - **`recvbuf`** choice O — starting address of receiving buffer
  - **`rcount`** int I — number of elements received from each process
  - **`recvtype`** MPI_Datatype I — data type of elements of receiving buffer
  - **`comm`** MPI_Comm I — communicator

C

MPI_Alltoall



- Sends data from all to all processes: transformation of dense matrix

- **MPI_Alltoall (sendbuf, scount, sendtype, recvbuf, rcount, recvtype, comm)**
  - **sendbuf** choice I starting address of sending buffer
  - **scount** int I number of elements sent to each process
  - **sendtype** MPI_Datatype I data type of elements of sending buffer
  - **recvbuf** choice O starting address of receiving buffer
  - **rcount** int I number of elements received from the root process
  - **recvtype** MPI_Datatype I data type of elements of receiving buffer
  - **comm** MPI_Comm I communicator

C

Examples by Collective Comm.

- Dot Products of Vectors
- Scatter/Gather
- **Reading Distributed Files**
- MPI_Allgatherv

Operations of Distributed Local Files

- In Scatter/Gather example, PE#0 reads global data, that is *scattered* to each processer, then parallel operations are done.

- If the problem size is very large, a single processor may not read entire global data.
  - If the entire global data is decomposed to distributed local data sets, each process can read the local data.
  - If global operations are needed to a certain sets of vectors, MPI functions, such as MPI_Gather etc. are available.

Reading Distributed Local Files: Uniform Vec. Length (1/2)

```
>$ cd /work/gt73/t73XXX/pFEM/mpi/S1
>$ ls a1.*
    a1.0 a1.1 a1.2 a1.3
>$ mpiicc -O3 file.c
>$ mpiifort -O3 file.f
(modify go4.sh for 4 processes)
>$ pjsub go4.sh
```

a1x.all is decomposed to 4 files.

a1.0
101.0
103.0
105.0
106.0
109.0
111.0
121.0
151.0

a1.1
201.0
203.0
205.0
206.0
209.0
211.0
221.0
251.0

a1.2
301.0
303.0
305.0
306.0
309.0
311.0
321.0
351.0

a1.3
401.0
403.0
405.0
406.0
409.0
411.0
421.0
451.0

go4.sh

```
#!/bin/sh
#PJM -N “test“                          Job Name
#PJM -L rscgrp=lecture3                 Name of “QUEUE”
#PJM -L node=1                          Node#
#PJM --mpi proc=4                       Total MPI Process#
#PJM -L elapse=00:15:00                 Computation Time
#PJM -g gt73                            Group Name (Wallet)
#PJM -j
#PJM -e err                             Standard Error
#PJM -o test.lst                        Standard Output

mpiexec.hydra -n ${PJM_MPI_PROC} ./a.out
```

Reading Distributed Local Files: Uniform Vec. Length (2/2)

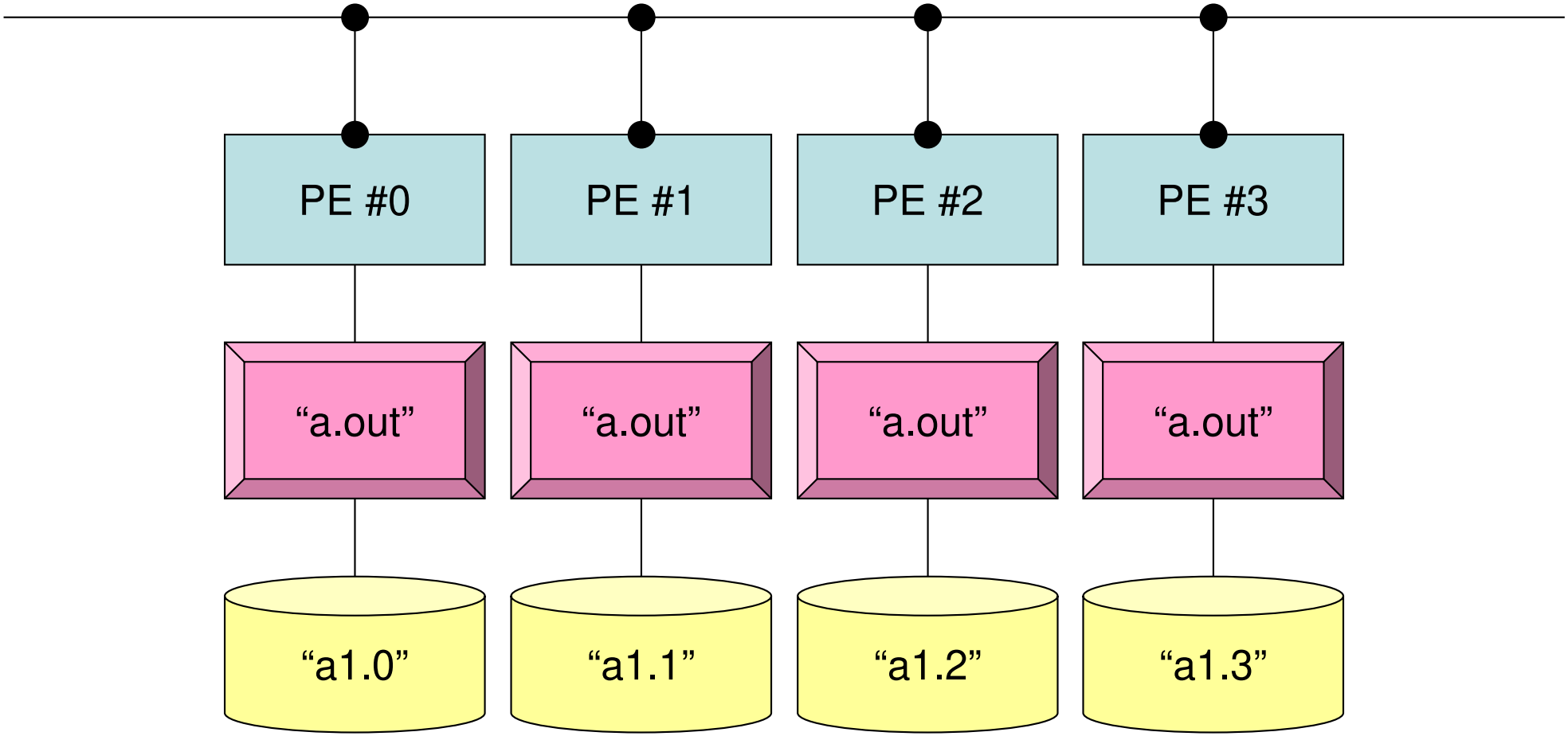
<$O-S1>/file.c

```
int main(int argc, char **argv){
        int i;
        int PeTot, MyRank;
        MPI_Comm SolverComm;
        double vec[8];
        char FileName[80];
        FILE *fp;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &PeTot);
        MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);

        sprintf(FileName, "a1.%d", MyRank);

        fp = fopen(FileName, "r");
        if(fp == NULL) MPI_Abort(MPI_COMM_WORLD, -1)
        for(i=0;i<8;i++){
                fscanf(fp, "%lf", &vec[i]);        }

        for(i=0;i<8;i++){
                printf("%5d%5d%10.0f¥n", MyRank, i+1, vec[i]);
        }
        MPI_Finalize();
        return 0;
}
```

Similar to "Hello"

Local ID is 0-7

Typical SPMD Operation




```
mpirun -np 4 a.out
```

Non-Uniform Vector Length (1/2)

```
>$ cd /work/gt73/t73XXX/pFEM/mpi/S1
>$ ls a2.*
   a2.0 a2.1 a2.2 a2.3
>$ cat a2.1
  5          Number of Components at each Process
  201.0      Components
  203.0
  205.0
  206.0
  209.0

>$ mpiicc -O3 file2.c
>$ mpiifort -O3 file2.f

(modify go4.sh for 4 processes)
>$ pjsub go4.sh
```

a2.0~a2.3

PE#0

```
   8
101.0
103.0
105.0
106.0
109.0
111.0
121.0
151.0
```

PE#1

```
   5
201.0
203.0
205.0
206.0
209.0
```

PE#2

```
   7
301.0
303.0
305.0
306.0
311.0
321.0
351.0
```

PE#3

```
   3
401.0
403.0
405.0
```

Non-Uniform Vector Length (2/2)

<$O-S1>/file2.c

```
int main(int argc, char **argv){
        int i, int PeTot, MyRank;
        MPI_Comm SolverComm;
        double *vec, *vec2, *vecg;
        int num;
        double sum0, sum;
        char filename[80];
        FILE *fp;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &PeTot);
        MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);

        sprintf(filename, "a2.%d", MyRank);
        fp = fopen(filename, "r");
        assert(fp != NULL);

        fscanf(fp, "%d", &num);
        vec = malloc(num * sizeof(double));
        for(i=0;i<num;i++){fscanf(fp, "%lf", &vec[i]);}

        for(i=0;i<num;i++){
                printf(" %5d%5d%5d%10.0f¥n", MyRank, i+1, num, vec[i]);}

        MPI_Finalize();
}
```

"num" is different at each process

How to generate local data

- Reading global data (N=NG)
  - Scattering to each process
  - Parallel processing on each process
  - (If needed) reconstruction of global data by gathering local data
- Generating local data (N=NL), or reading distributed local data
  - Generating or reading local data on each process
  - Parallel processing on each process
  - (If needed) reconstruction of global data by gathering local data

- In future, latter case is more important, but former case is also introduced in this class for understanding of operations of global/local data.

Examples by Collective Comm.

- Dot Products of Vectors
- Scatter/Gather
- Reading Distributed Files
- **MPI_Allgatherv**

MPI_Gatherv, MPI_Scatterv

- MPI_Gather, MPI_Scatter
  - Length of message from/to each process is uniform

- MPI_XXXv extends functionality of MPI_XXX by allowing a varying count of data from each process:
  - MPI_Gatherv
  - MPI_Scatterv
  - MPI_Allgatherv
  - MPI_Alltoallv

C

MPI_Allgatherv

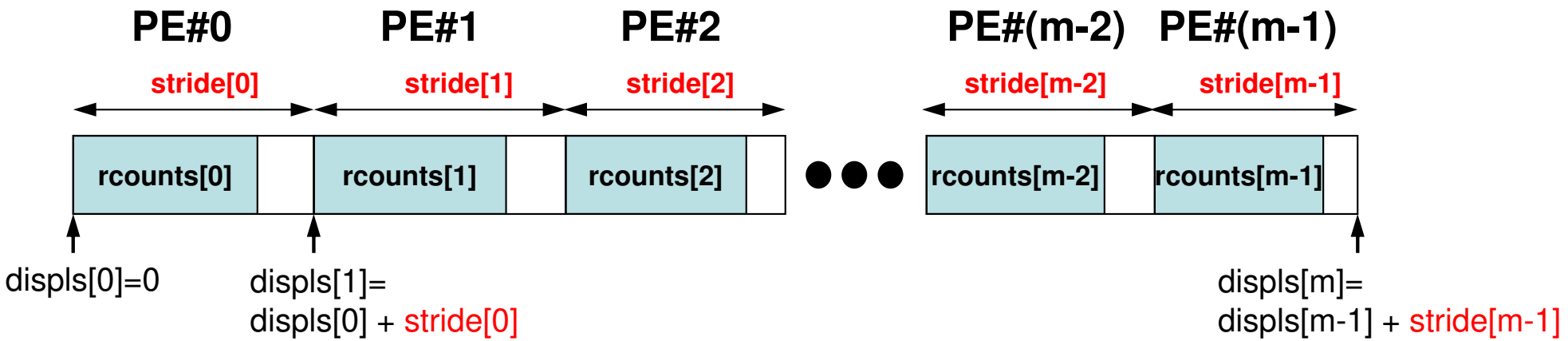
- Variable count version of MPI_Allgather
  - creates “global data” from “local data”

- **MPI_Allgatherv (sendbuf, scount, sendtype, recvbuf, rcounts, displs, recvtype, comm)**
  - **sendbuf** choice I starting address of sending buffer
  - **scount** int I number of elements sent to each process
  - **sendtype** MPI_Datatype I data type of elements of sending buffer
  - **recvbuf** choice O starting address of receiving buffer
  - **rcounts** int I integer array (of length *groupsize*) containing the number of elements that are to be received from each process (array: size= PETOT)
  - **displs** int I integer array (of length *groupsize*). Entry *i* specifies the displacement (relative to recvbuf) at which to place the incoming data from process *i* (array: size= PETOT+1)
  - **recvtype** MPI_Datatype I data type of elements of receiving buffer
  - **comm** MPI_Comm I communicator

C

MPI_Allgatherv (cont.)

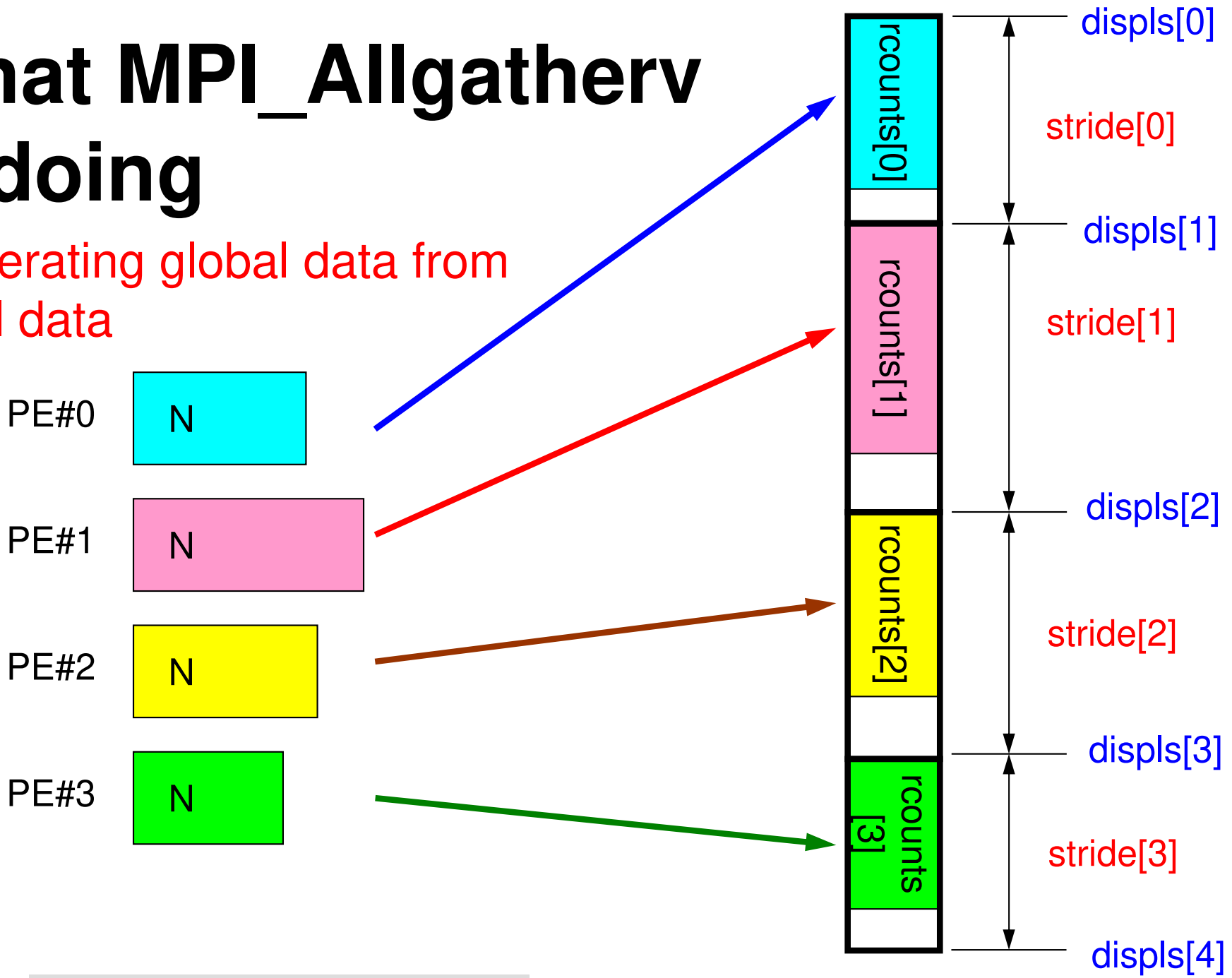
- **`MPI_Allgatherv (sendbuf, scount, sendtype, recvbuf, rcounts, displs, recvtype, comm)`**
  - **`rcounts`** int I integer array (of length *groupsize*) containing the number of elements that are to be received from each process (array: size= `PETOT`)
  - **`displs`** int I integer array (of length *groupsize*). Entry *`i`* specifies the displacement (relative to recvbuf) at which to place the incoming data from process *`i`* (array: size= `PETOT+1`)
  - These two arrays are related to size of final "global data", therefore each process requires information of these arrays (`rcounts, displs`)
    - Each process must have same values for all components of both vectors
  - Usually, **`stride(i)=rcounts(i)`**



size[recvbuf]= displs[PETOT]= sum[stride]

What MPI_Allgatherv is doing

Generating global data from local data

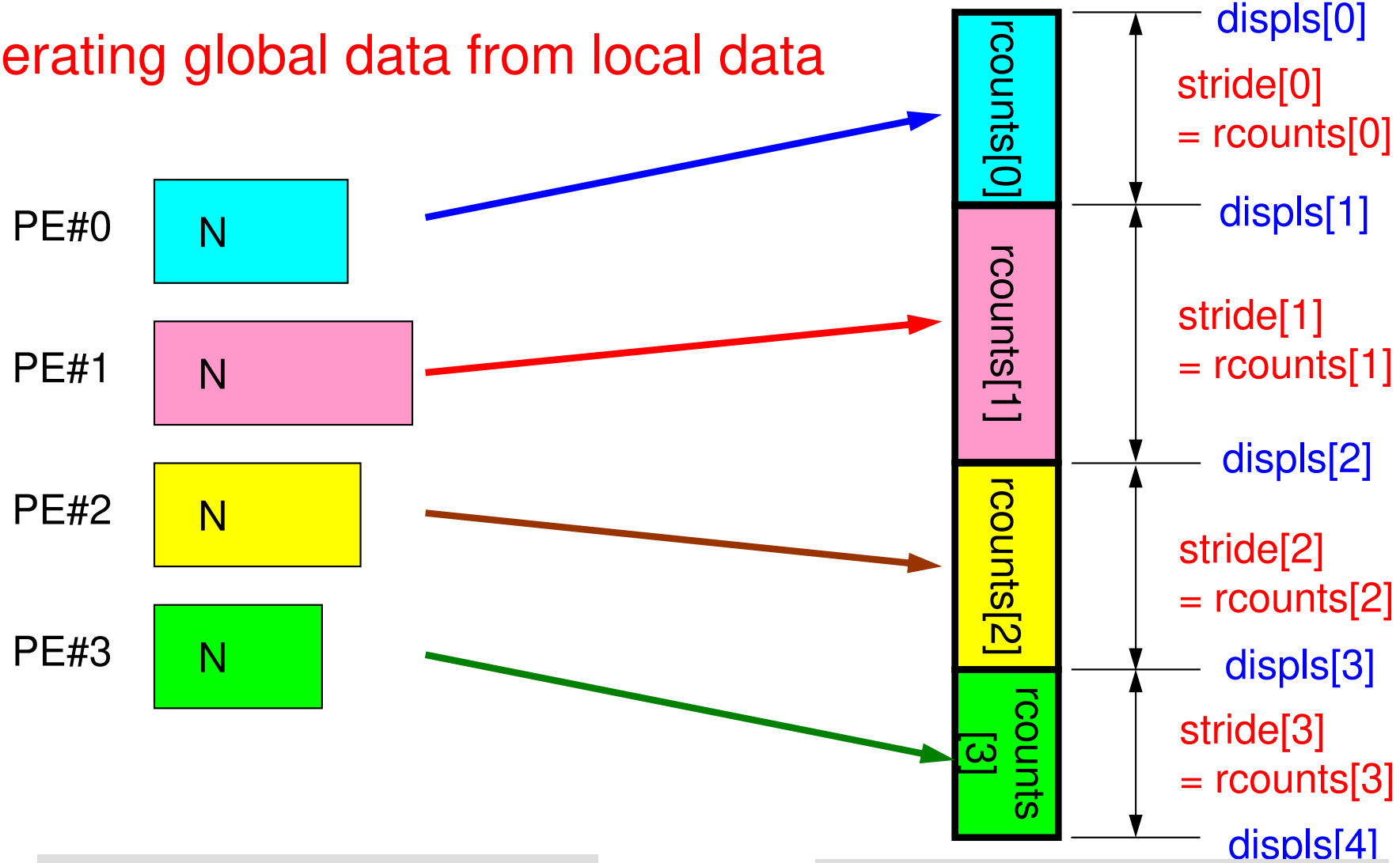


Local Data: sendbuf

Global Data: recvbuf

What MPI_Allgatherv is doing

Generating global data from local data



Local Data: sendbuf

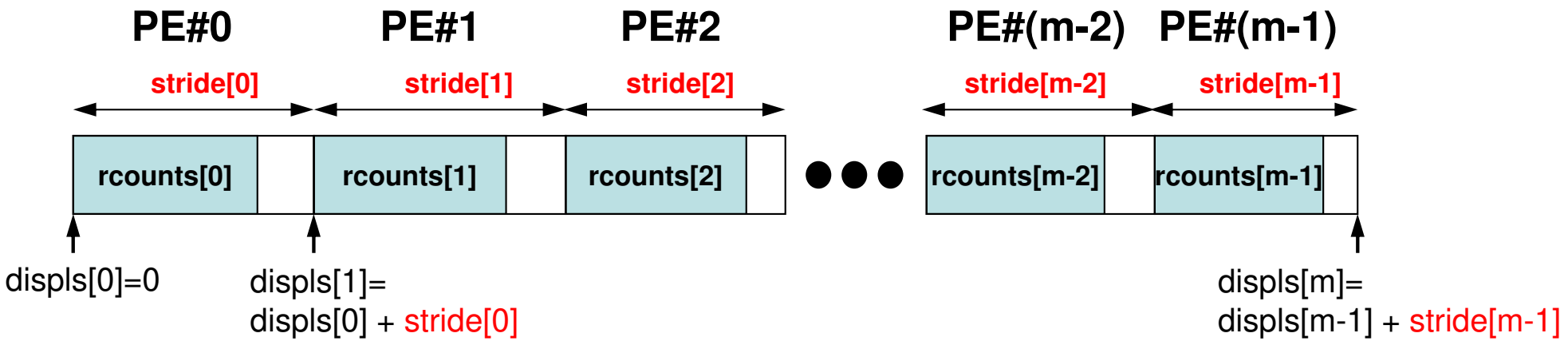
Global Data: recvbuf

MPI_Allgatherv in detail (1/2)

C

- **`MPI_Allgatherv (sendbuf, scount, sendtype, recvbuf, rcounts, displs, recvtype, comm)`**

- **`rcounts`**
  - Size of message from each PE: Size of Local Data (Length of Local Vector)
- **`displs`**
  - Address/index of each local data in the vector of global data
  - **`displs(PETOT+1)`** = Size of Entire Global Data (Global Vector)

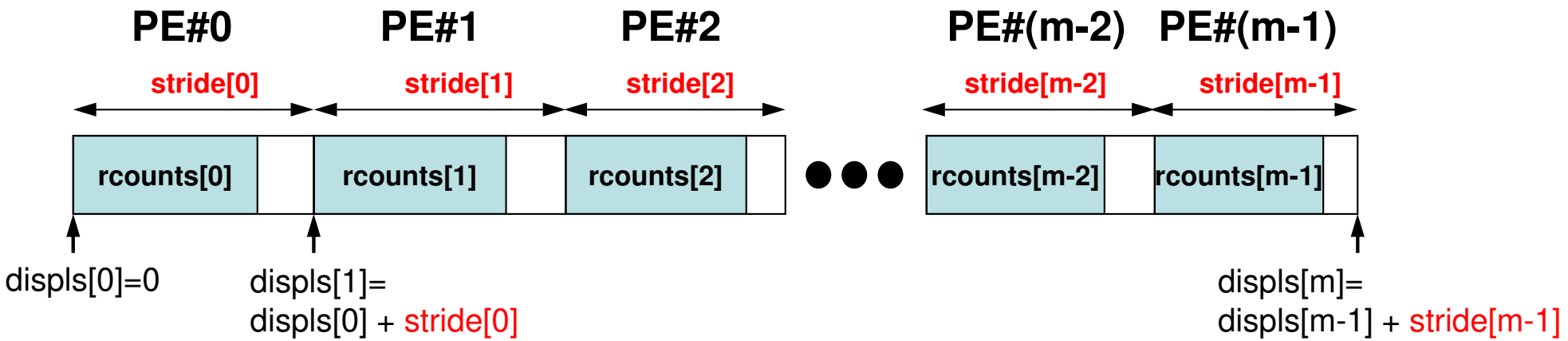


size[recvbuf]= displs[PETOT]= sum[stride]

MPI_Allgatherv in detail (2/2)

C

- Each process needs information of **rcounts** & **displs**
  - **"rcounts"** can be created by gathering local vector length **"N"** from each process.
  - On each process, **"displs"** can be generated from **"rcounts"** on each process.
    - **stride[i]= rcounts[i]**
  - Size of **"recvbuf"** is calculated by summation of **"rcounts"**.



size[recvbuf]= displs[PETOT]= sum[stride]

Preparation for MPI_Allgatherv <$O-S1>/agv.c

- Generating global vector from “a2.0”~”a2.3”.
- Length of the each vector is 8, 5, 7, and 3, respectively. Therefore, size of final global vector is 23 (= 8+5+7+3).

a2.0~a2.3

PE#0

```
   8
101.0
103.0
105.0
106.0
109.0
111.0
121.0
151.0
```

PE#1

```
   5
201.0
203.0
205.0
206.0
209.0
```

PE#2

```
   7
301.0
303.0
305.0
306.0
311.0
321.0
351.0
```

PE#3

```
   3
401.0
403.0
405.0
```

Preparation: MPI_Allgatherv (1/4)

C

<$O-S1>/agv.c

```
int main(int argc, char **argv){
        int i;
        int PeTot, MyRank;
        MPI_Comm SolverComm;
        double *vec, *vec2, *vecg;
        int *Rcounts, *Displs;
        int n;
        double sum0, sum;
        char filename[80];
        FILE *fp;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &PeTot);
        MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);

        sprintf(filename, "a2.%d", MyRank);
        fp = fopen(filename, "r");
        assert(fp != NULL);

        fscanf(fp, "%d", &n);
        vec = malloc(n * sizeof(double));
        for(i=0;i<n;i++){
                fscanf(fp, "%lf", &vec[i]);
        }
```

n(NL) is different at each process

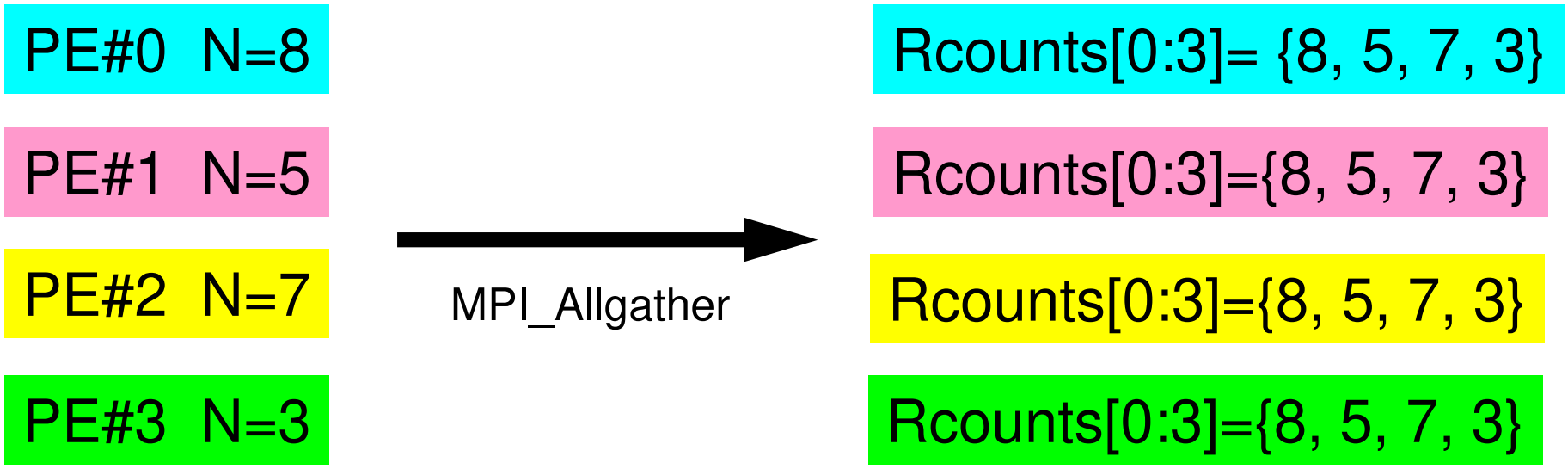
Preparation: MPI_Allgatherv (2/4)

C

<$O-S1>/agv.c

```
Rcounts= calloc(PeTot, sizeof(int));
Displs = calloc(PeTot+1, sizeof(int));

printf("before %d %d", MyRank, n);
for(i=0;i<PeTot;i++){printf(" %d", Rcounts[i]);}

MPI_Allgather(&n, 1, MPI_INT, Rcounts, 1, MPI_INT, MPI_COMM_WORLD);

printf("after  %d %d", MyRank, n);
for(i=0;i<PeTot;i++){printf(" %d", Rcounts[i]);}

Displs[0] = 0;
```

Rcounts on each PE



Preparation: MPI_Allgatherv (2/4)

C

<$O-S1>/agv.c

```
 Rcounts= calloc(PeTot, sizeof(int));
 Displs = calloc(PeTot+1, sizeof(int));

 printf("before %d %d", MyRank, n);
 for(i=0;i<PeTot;i++){printf(" %d", Rcounts[i]);}

 MPI_Allgather(&n, 1, MPI_INT, Rcounts, 1, MPI_INT, MPI_COMM_WORLD);

 printf("after  %d %d", MyRank, n);
 for(i=0;i<PeTot;i++){printf(" %d", Rcounts[i]);}

 Displs[0] = 0;
 for(i=0;i<PeTot;i++){
                 Displs[i+1] = Displs[i] + Rcounts[i];}

         printf("CoundIndex  %d ", MyRank);
         for(i=0;i<PeTot+1;i++){
                 printf(" %d", Displs[i]);
         }
         MPI_Finalize();
         return 0;
}
```

Rcounts on each PE

Displs on each PE

Preparation: MPI_Allgatherv (3/4)

```
> cd /work/gt73/t73XXX/pFEM/mpi/S1
> mpiicc -O3 agv.c

(modify go4.sh for 4 processes)
> pjsub go4.sh


before       0       8       0       0       0       0
after        0       8       8       5       7       3
Displs       0       0       8      13      20      23

before       1       5       0       0       0       0
after        1       5       8       5       7       3
Displs       1       0       8      13      20      23

before       3       3       0       0       0       0
after        3       3       8       5       7       3
Displs       3       0       8      13      20      23

before       2       7       0       0       0       0
after        2       7       8       5       7       3
Displs       2       0       8      13      20      23
```

```
write (*,‘(a,10i8)’) “before”, my_rank, N, rcounts
write (*,'(a,10i8)') "after ", my_rank, N, rcounts
write (*,'(a,10i8)') "displs", my_rank, displs
```

Preparation: MPI_Allgatherv (4/4)

- Only "recvbuf" is not defined yet.
- Size of "recvbuf" = "Displs[PETOT]"

```
MPI_Allgatherv
      ( VEC , N, MPI_DOUBLE,
        recvbuf, rcounts, displs, MPI_DOUBLE,
        MPI_COMM_WORLD);
```

Report S1 (1/2)

- Deadline: January 26th (Wed), 2022, 17:00@ITC-LMS

- Problem S1-1
  - Read local files <$O-S1>/a1.0~a1.3, <$O-S1>/a2.0~a2.3.
  - Develop codes which calculate norm $||x||_2$ of global vector for each case.
    - <$O-S1>file.c, <$O-S1>file2.c
- Problem S1-2
  - Read local files <$O-S1>/a2.0~a2.3.
  - Develop a code which constructs “global vector” using MPI_Allgatherv.

Report S1 (2/2)

- Problem S1-3
  - Develop parallel program which calculates the following numerical integration using “trapezoidal rule” by MPI_Reduce, MPI_Bcast etc.
  - Measure computation time, and parallel performance

$$\int_0^1 \frac{4}{1+x^2}\,dx$$

- Report
  - Cover Page: Name, ID, and Problem ID (S1) must be written.
  - Less than two pages including figures and tables (A4) for each of three sub-problems
    - Strategy, Structure of the Program, Remarks
  - Source list of the program (if you have bugs)
  - Output list (as small as possible)

Options for Optimization

General Option (-O3)

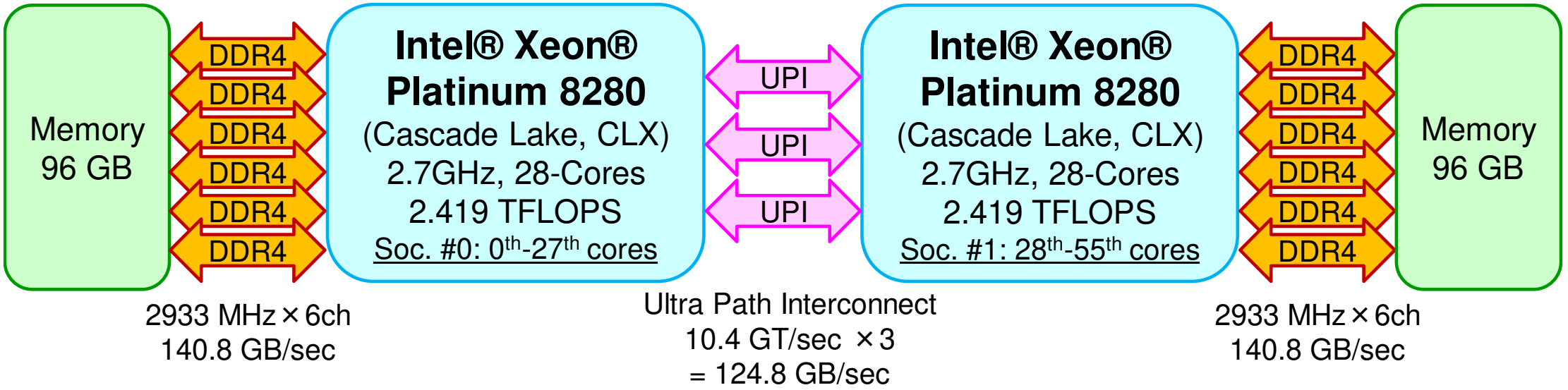
```
$ mpiifort -O3 test.f
$ mpiicc -O3 test.c
```

Special Options for AVX512, NOT Necessarily Fast …

```
$ mpiifort -align array64byte -O3 -axCORE-AVX512 test.f
$ mpiicc -align -O3 -axCORE-AVX512 test.c
```

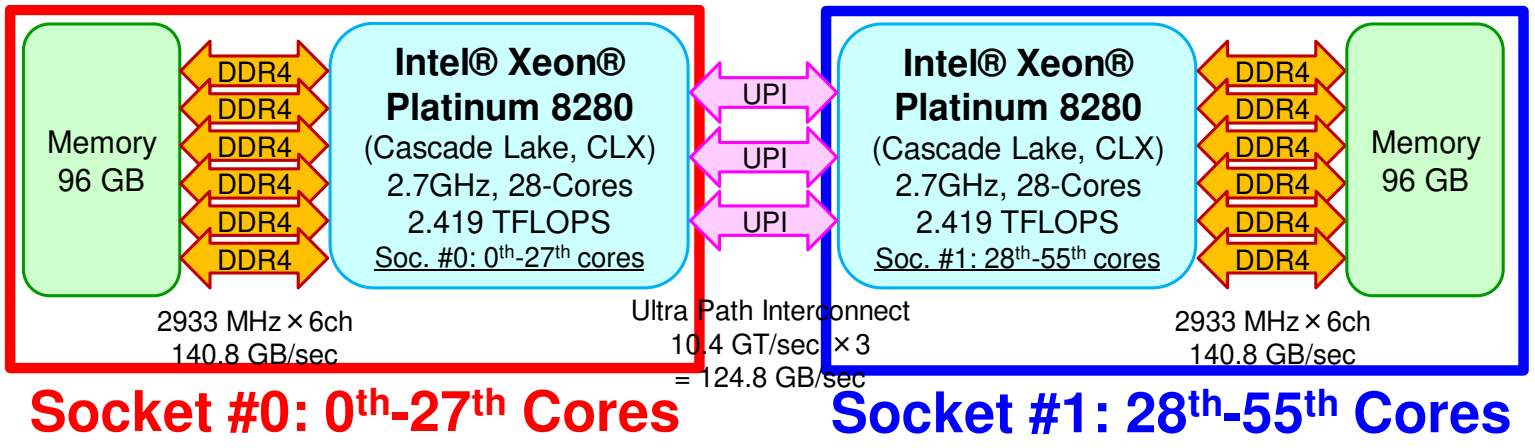
Process Number

```
#PJM -L node=1;#PJM --mpi proc= 1    1-node, 1-proc, 1-proc/n
#PJM -L node=1;#PJM --mpi proc= 4    1-node, 4-proc, 4-proc/n
#PJM -L node=1;#PJM --mpi proc=16    1-node,16-proc,16-proc/n
#PJM -L node=1;#PJM --mpi proc=28    1-node,28-proc,28-proc/n
#PJM -L node=1;#PJM --mpi proc=56    1-node,56-proc,56-proc/n

#PJM -L node=4;#PJM --mpi proc=128   4-node,128-proc,32-proc/n
#PJM -L node=8;#PJM --mpi proc=256   8-node,256-proc,32-proc/n
#PJM -L node=8;#PJM --mpi proc=448   8-node,448-proc,56-proc/n
```



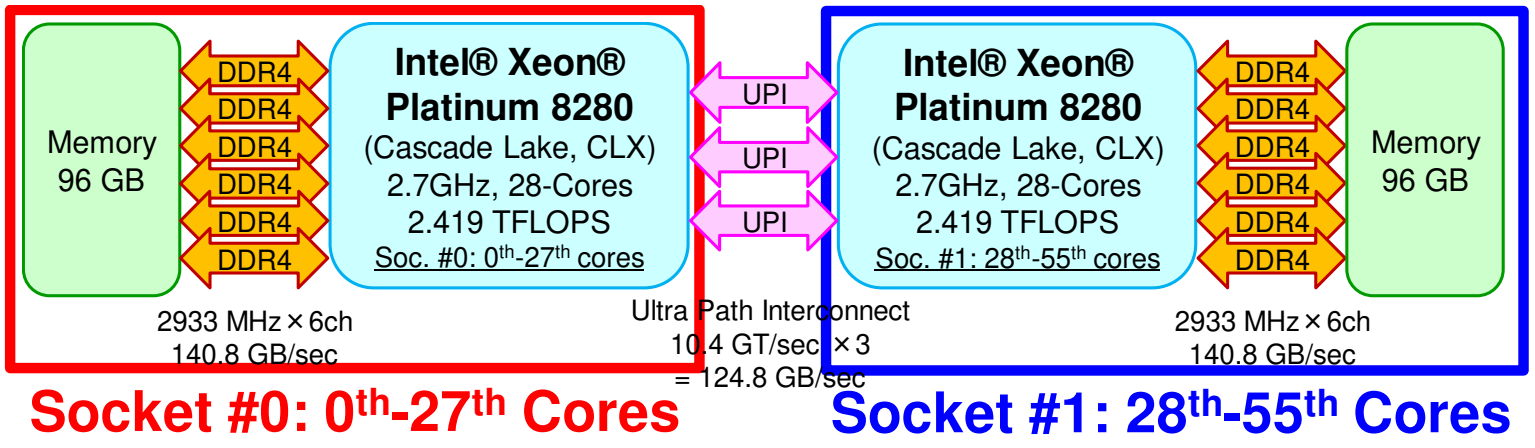
a01.sh: Use 1-core (0th)

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=lecture3
#PJM -L node=1
#PJM --mpi proc=1
#PJM -L elapse=00:15:00
#PJM -g gt73
#PJM -j
#PJM -e err
#PJM -o test.lst

export I_MPI_PIN_PROCESSOR_LIST=0

mpiexec.hydra -n ${PJM_MPI_PROC} ./a.out
```



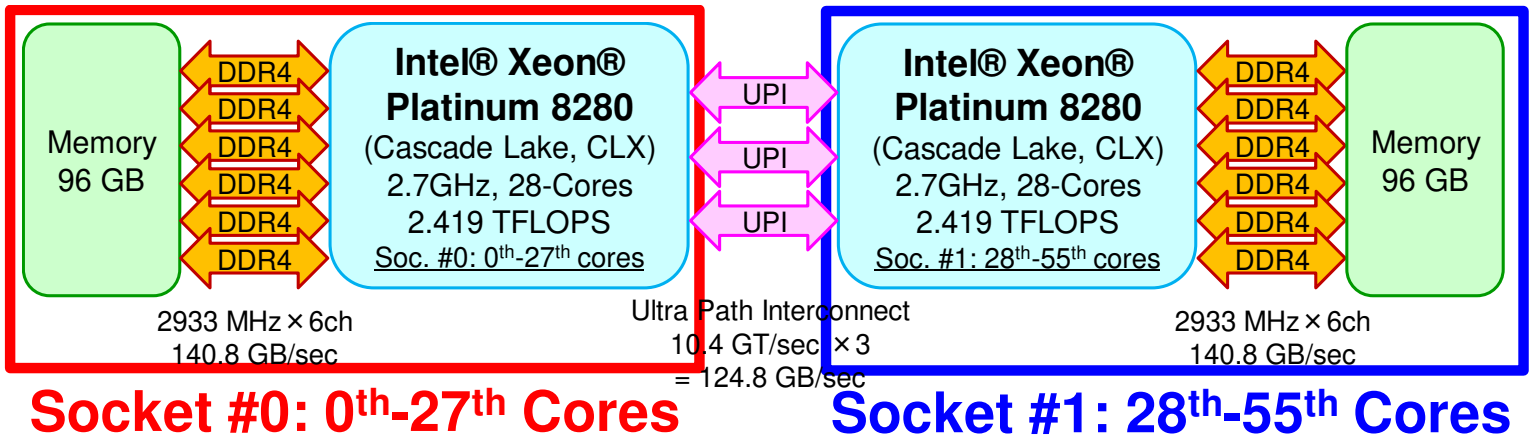
a24.sh: Use 24-cores (0th-23rd)

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=lecture3
#PJM -L node=1
#PJM --mpi proc=24
#PJM -L elapse=00:15:00
#PJM -g gt73
#PJM -j
#PJM -e err
#PJM -o test.lst

export I_MPI_PIN_PROCESSOR_LIST=0-23

mpiexec.hydra -n ${PJM_MPI_PROC} ./a.out
```



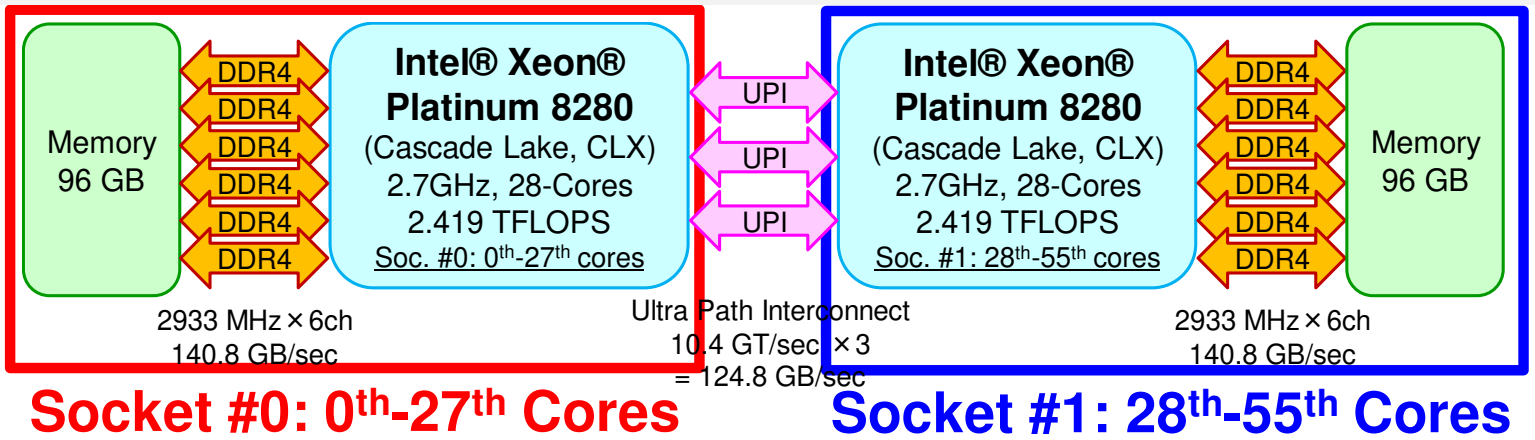
a48.sh: Use 48-cores (0th-23rd, 28th-51st)

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=lecture3
#PJM -L node=1
#PJM --mpi proc=48
#PJM -L elapse=00:15:00
#PJM -g gt73
#PJM -j
#PJM -e err
#PJM -o test.lst

export I_MPI_PIN_PROCESSOR_LIST=0-23,28-51

mpiexec.hydra -n ${PJM_MPI_PROC} ./a.out
```



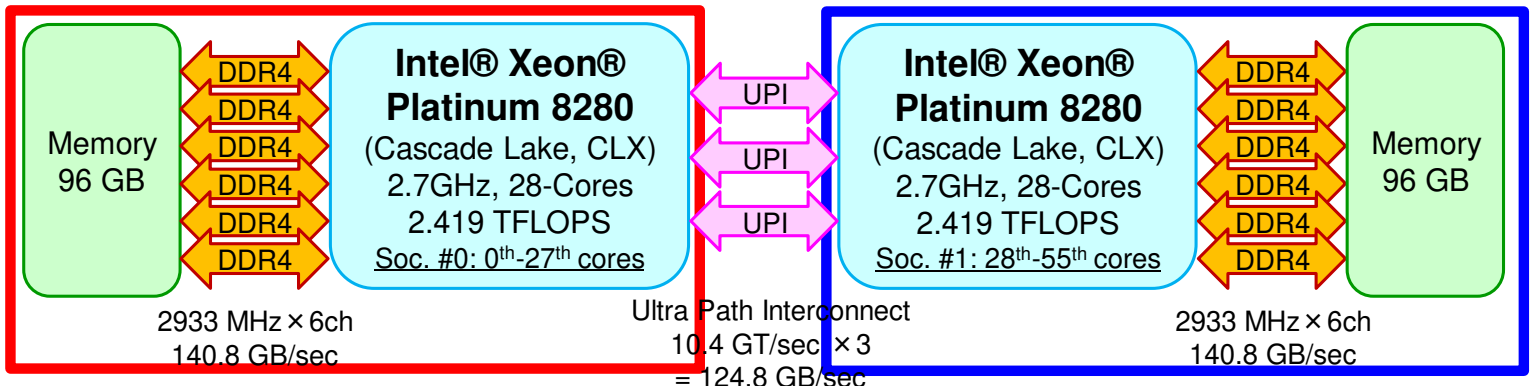
b48.sh: Use 8x48-cores (0th-23rd, 28th-51st)

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=lecture3
#PJM -L node=8
#PJM --mpi proc=384                          384÷8= 48-cores/node
#PJM -L elapse=00:15:00
#PJM -g gt73
#PJM -j
#PJM -e err
#PJM -o test.lst

export I_MPI_PIN_PROCESSOR_LIST=0-23,28-51

mpiexec.hydra -n ${PJM_MPI_PROC} ./a.out
```

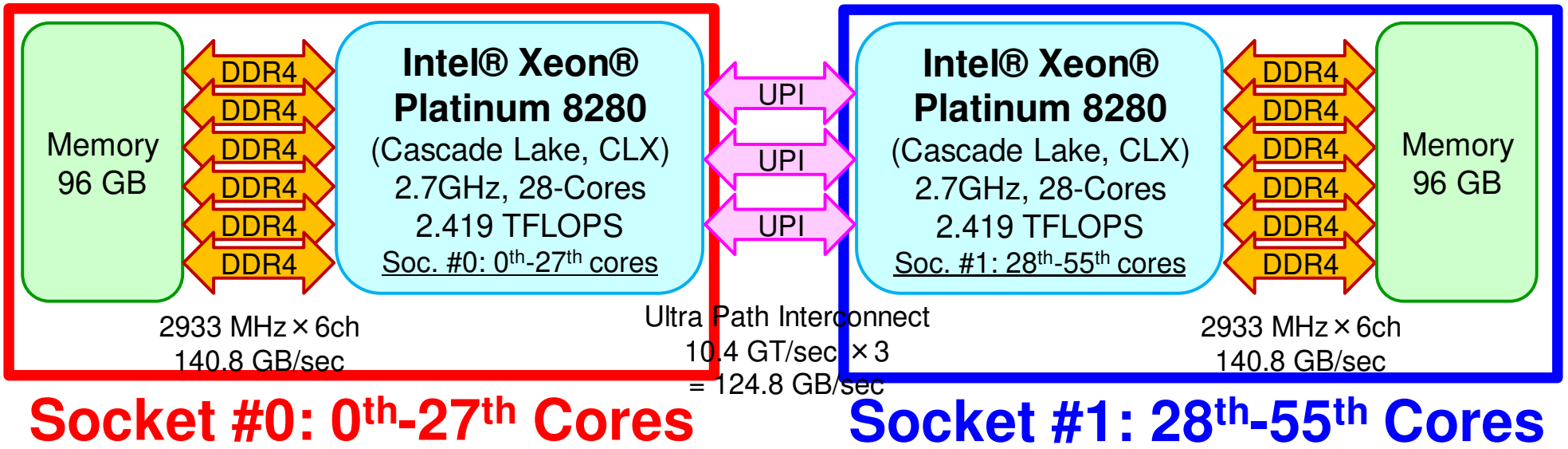


b48b.sh: Use 8x48-cores (0th-23rd, 28th-51st)

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=lecture3
#PJM -L node=8
#PJM --mpi proc=384                                  384÷8= 48-cores/node
#PJM -L elapse=00:15:00
#PJM -g gt73
#PJM -j
#PJM -e err
#PJM -o test.lst

export I_MPI_PIN_PROCESSOR_LIST=0-23,28-51

mpiexec.hydra -n ${PJM_MPI_PROC} numactl -l ./a.out
```



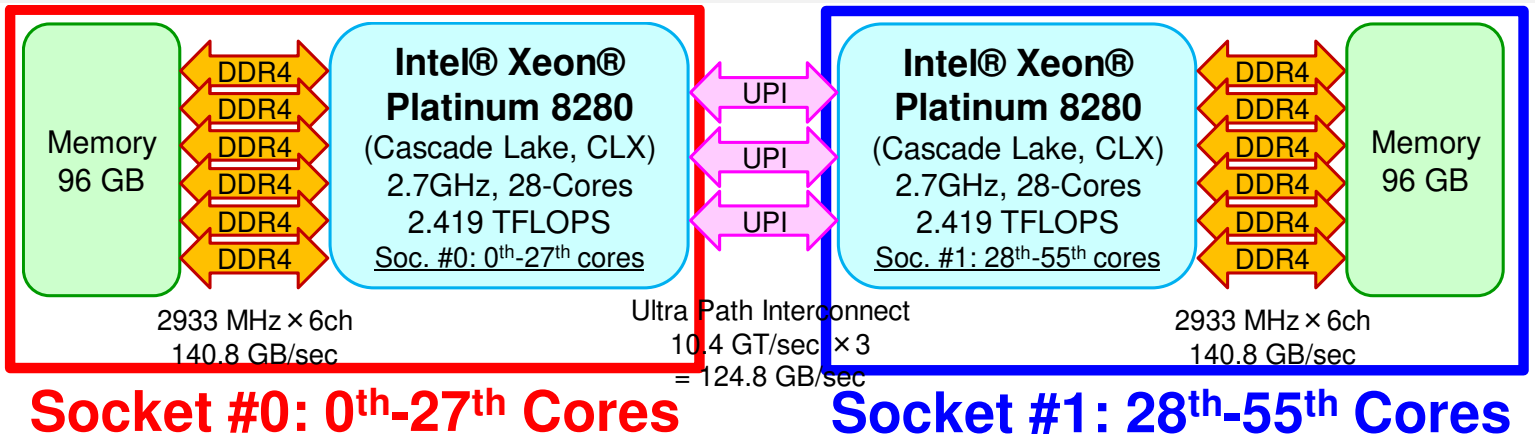
NUMA Architecture



- Each Node of Oakbridge-CX (OBCX)
  - 2 Sockets (CPU's) of Intel CLX
  - Each socket has 28 cores
- Each core of a socket can access to the memory on the other socket : NUMA (Non-Uniform Memory Access)
  - **`numactl -l`** : local memory to be used
  - Sometimes (not always), more stable with this option

Use 8x48-cores, 48-cores are randomly selected from 56-cores

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=lecture3
#PJM -L node=8
#PJM --mpi proc=384                                  384÷8= 48-cores/node
#PJM -L elapse=00:15:00
#PJM -g gt73
#PJM -j
#PJM -e err
#PJM -o test.lst

export I_MPI_PIN_PROCESSOR_LIST=0-23,28-51

mpiexec.hydra -n ${PJM_MPI_PROC} ./a.out
```



Use 8x56-cores

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=lecture3
#PJM -L node=8
#PJM --mpi proc=448                                  448÷8= 56-cores/node
#PJM -L elapse=00:15:00
#PJM -g gt73
#PJM -j
#PJM -e err
#PJM -o test.lst

mpiexec.hydra -n ${PJM_MPI_PROC} ./a.out
```