First day

Basics of parallel programming

RIKEN CCS HPC Summer School
Hiroya Matsuba, RIKEN CCS

Today's schedule: Basics of parallel programming

- 7/22 AM: Lecture
  - Goals
    - Understand the design of typical parallel computers
    - Understand how we can make programs for such parallel computers
- 7/22 PM-1: Preparation of programming environment
  - Goal
    - Log in to K computer
- 7/22 PM-2: Hands on
  - Goal
    - Create and run simple parallel programs

Why do we need parallel programing?

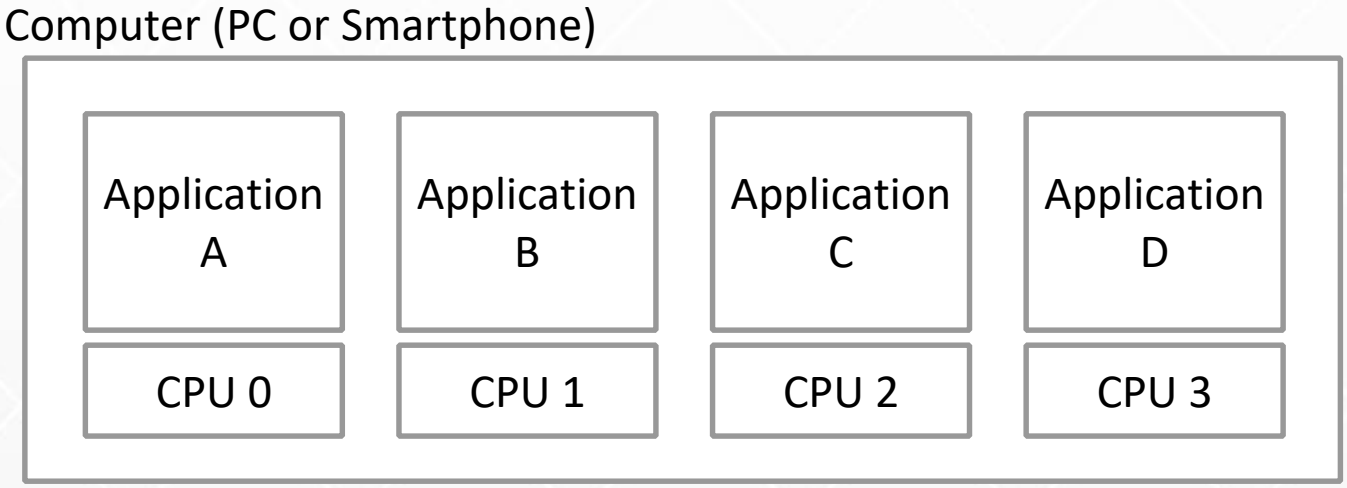
- Most of the today's computers are parallel computers
  - Cloud infrastructure
    - Amazon, Google, Microsoft have huge number of servers
  - PC
    - Intel Core i7 Processor has 4 or 6 cores
  - Smartphones
    - Samsung Galaxy S9 has an 8-core processor

- Does everyone do parallel programming?

  - No.

Why do we need parallel programing?

- We don’t have to do parallel programming if we have multiple processors for different purposes.

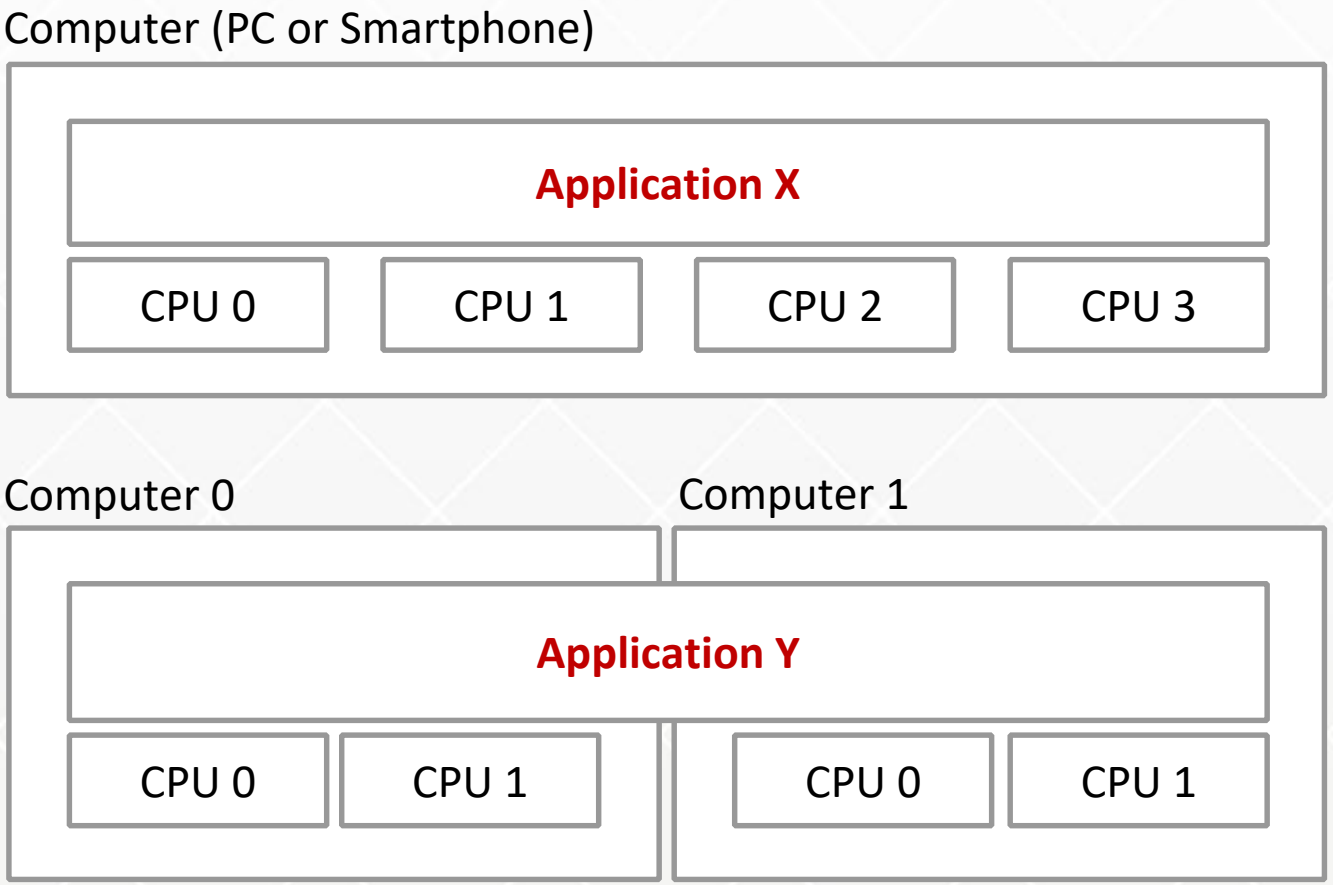


- All the applications are NOT parallel programs
- Operating system automatically selects available CPUs and different applications run on different CPUs
- This is what is happening in most cases

Why do we need parallel programing?

- What if an application needs more power than a single processor?

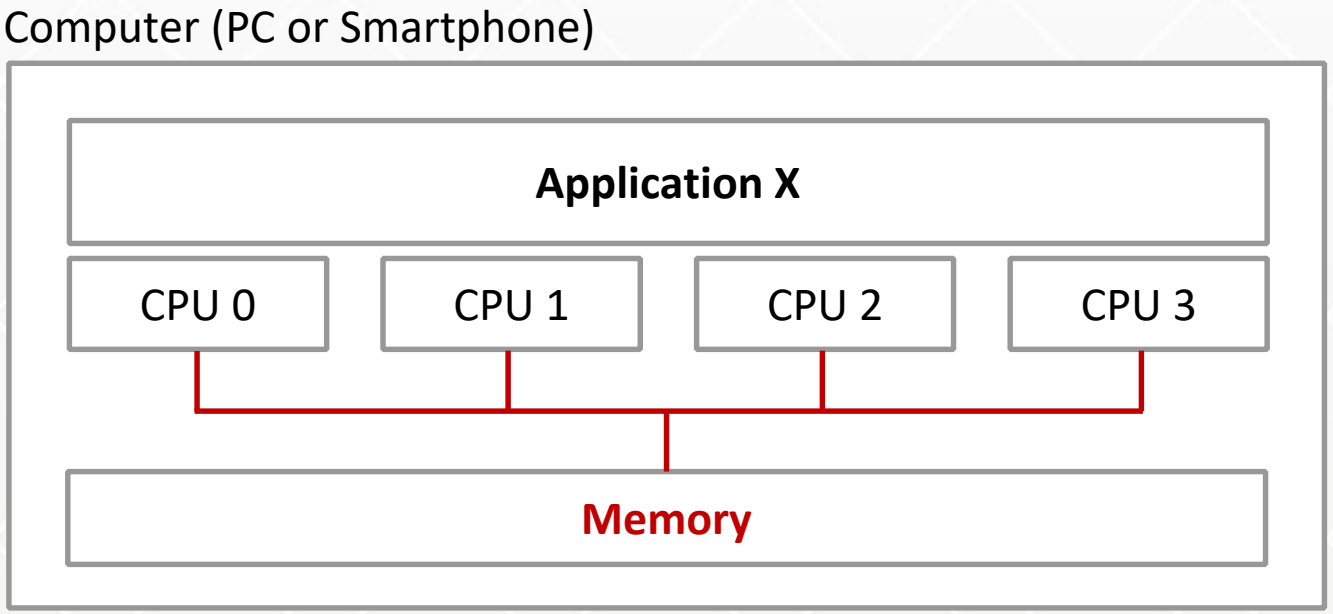
  - **Parallel programming is required.**



Two types of parallel programming

- Type 1: An application uses multiple processors in a single computer
  - "Single computer" means a computer that consists of a set of processors that share the same memory

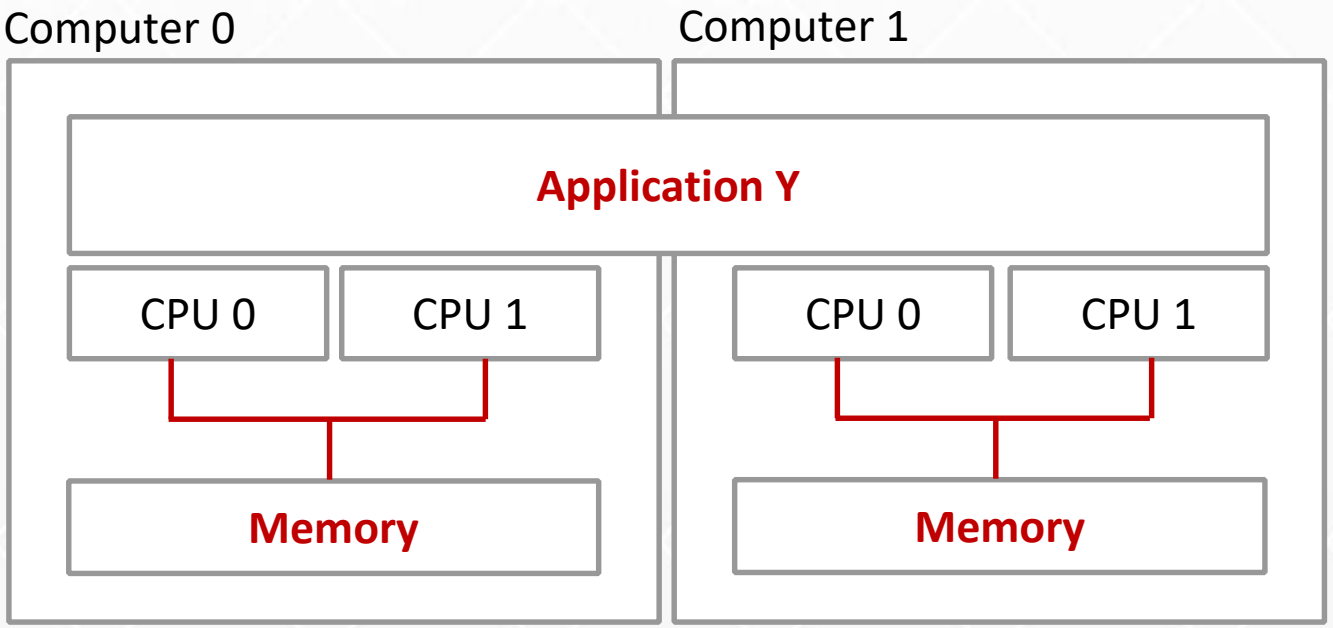
  - **Shared memory parallel programming**



All the CPUs are connected to a single memory

Two types of parallel programming

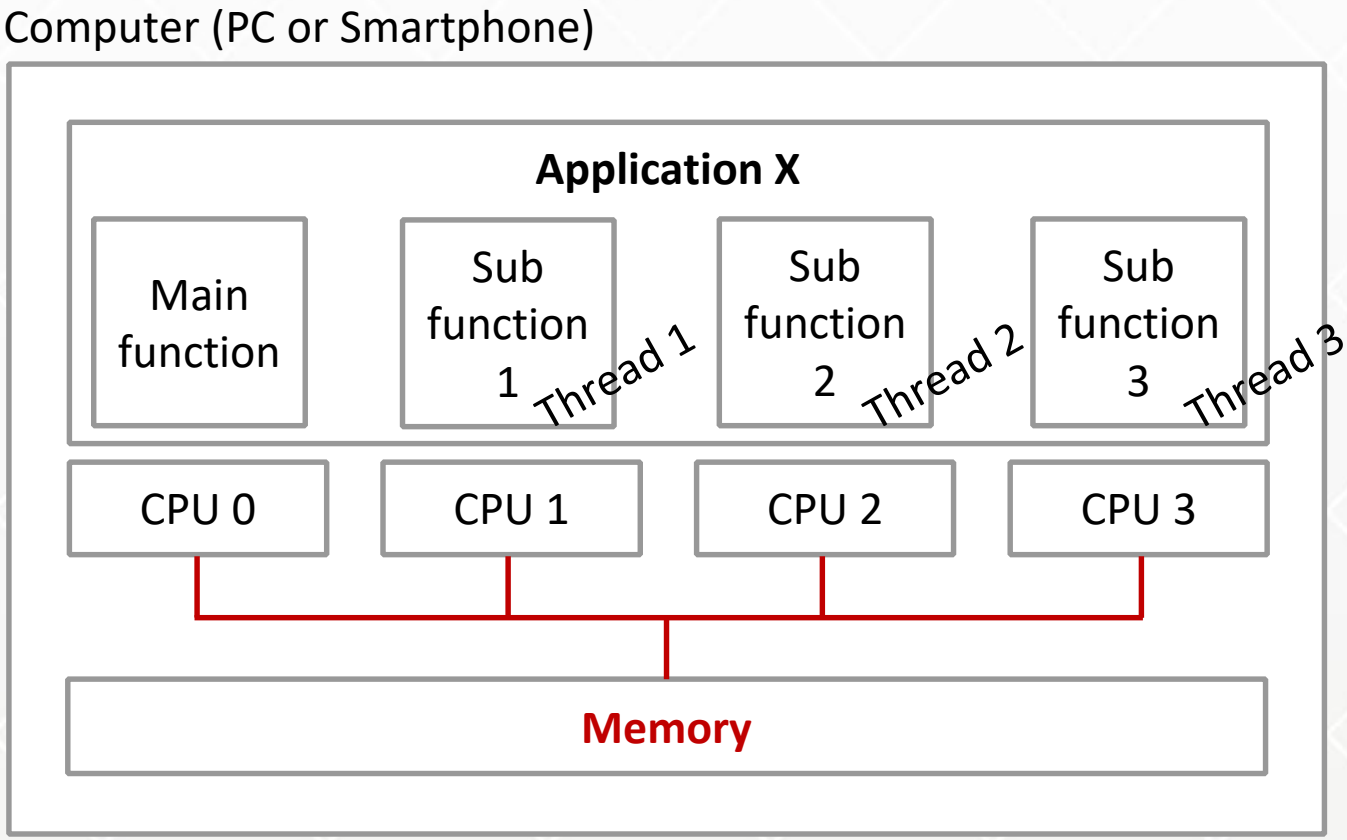
- Type 2: An application uses multiple computers
  - **Distributed memory parallel programming**



CPUs can access only their local memory

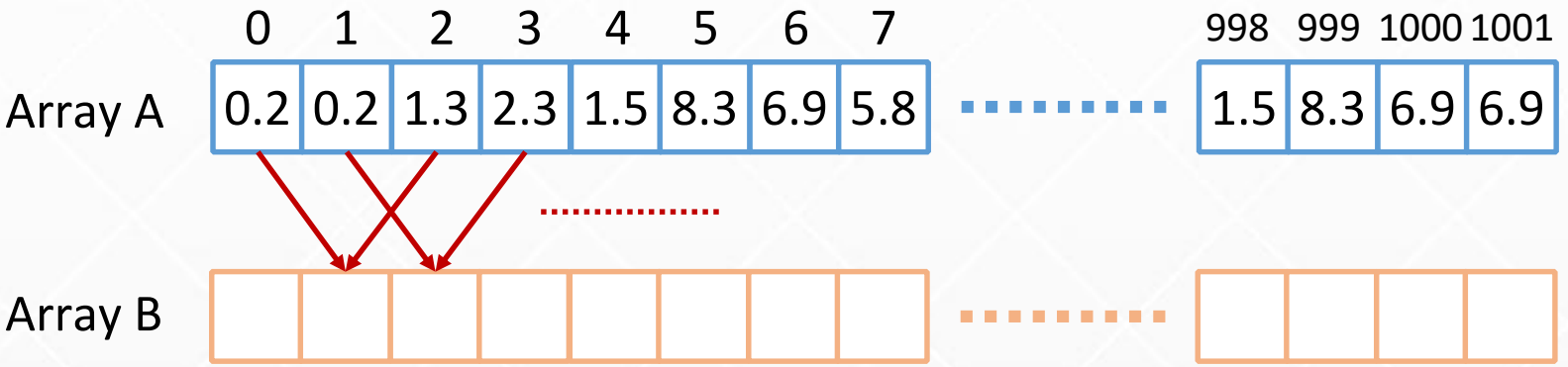
Shared memory parallel programming

- We usually use threads of operating systems
  - This is a functionality of operating systems that enable us to run a part of application programs on different processors



Example program

- The following program is used throughout this lecture




```
for(int i = 1; i <= 1000; i++) {
    b[i] = (a[i-1] + a[i+1]) / 2.0;
}
```

Shared memory parallel programming

- How we program with threads
  - You don’t have to understand this program

```
#include <pthread.h>

/* They are shared among threads */
char a[1002], b[1002];

void *calc(void *arg) {
    int start = (int)(long)arg;
    for(int i = start; i < start + 250; i++)
        b[i] = (a[i-1] + a[i+1]) / 2.0;
}

int main(int argc, char *argv[]) {
    load_a();
    for(int i = 0; i < 4; i++) {
        pthread_create(calc, (void *)(i * 250 + 1));
        /* NOTE: this returns immediately */
    }
    pthread_join(…) /* Wait for the threads done */
}
```

This part is run in parallel

* This doesn't compile because some arguments are omitted

Shared memory parallel programming

- Multi thread programming
  - The concept is easy to understand
    - Target data exists on the shared memory
    - Single- and multi-thread programming can essentially be the same
      - ~ Just the difference of how many processors are participating
  - Writing a stable multithread program is extremely difficult
    - Some data structures may be destroyed if we access them simultaneously from multiple threads
      - ~ You don’t have to worry about it today

Shared memory parallel programming

- Shared memory parallel programming for HPC
  - **OpenMP**
    - Very useful language extension that enables easy multi-thread programming
  - Example of OpenMP program
    - Just to add one line to the unparallelized program!

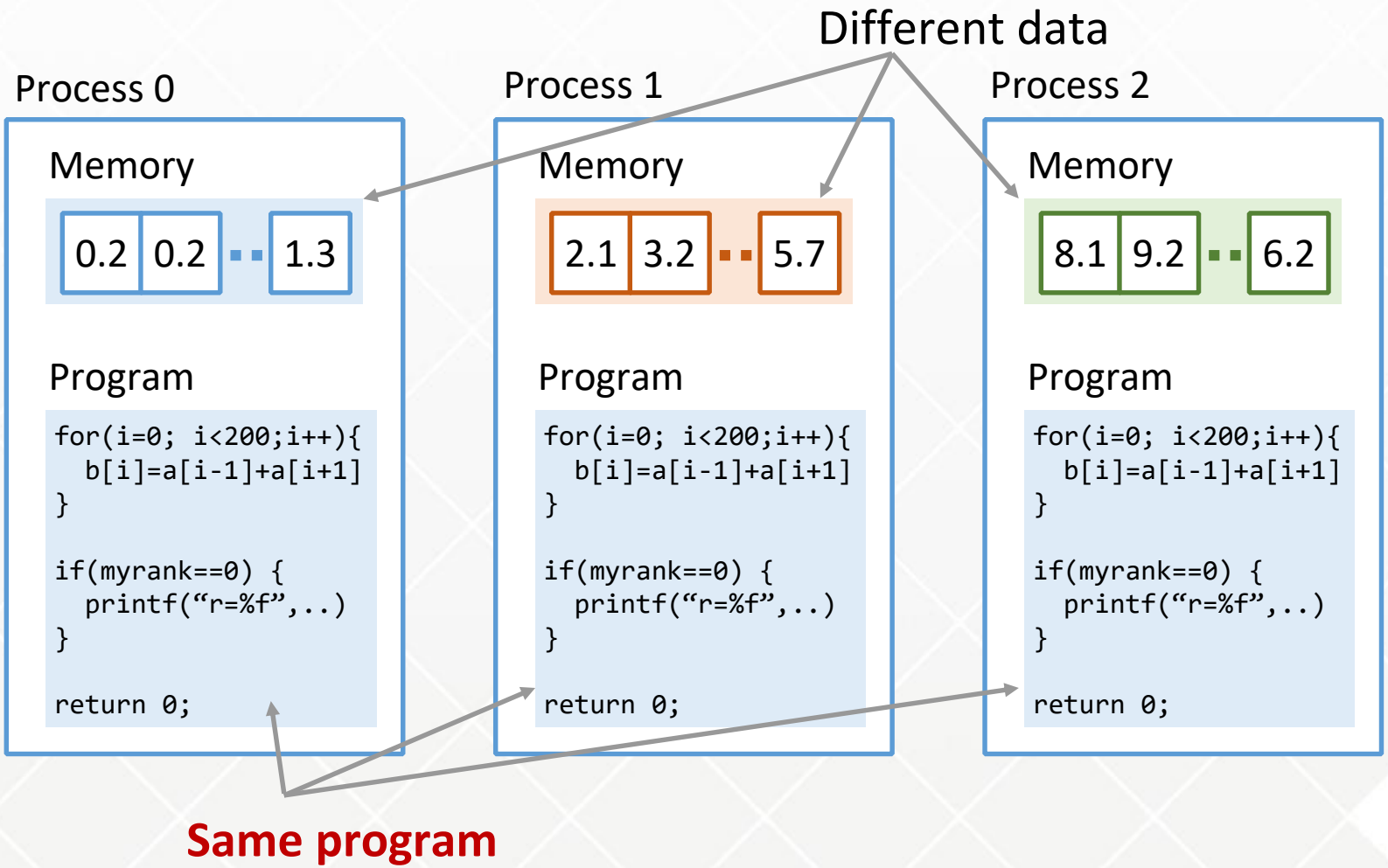
```
char a[1002], b[1002];

int main(int argc, char *argv[]) {
   load_a();

   #pragma omp parallel for
   for(int i = 1; i <= 1000; i++) {
       b[i] = (a[i-1] + a[i+1]) / 2.0;
   }
}
```

Distributed memory parallel programming

- Today's main interest
- It is difficult and complicated but we have to learn because:
  - most of today's supercomputers are distributed memory parallel computers
  - it is only the way to speed up programs by more than hundreds times
  - it is only the way to solve problems that are larger than the size of single computer's memory

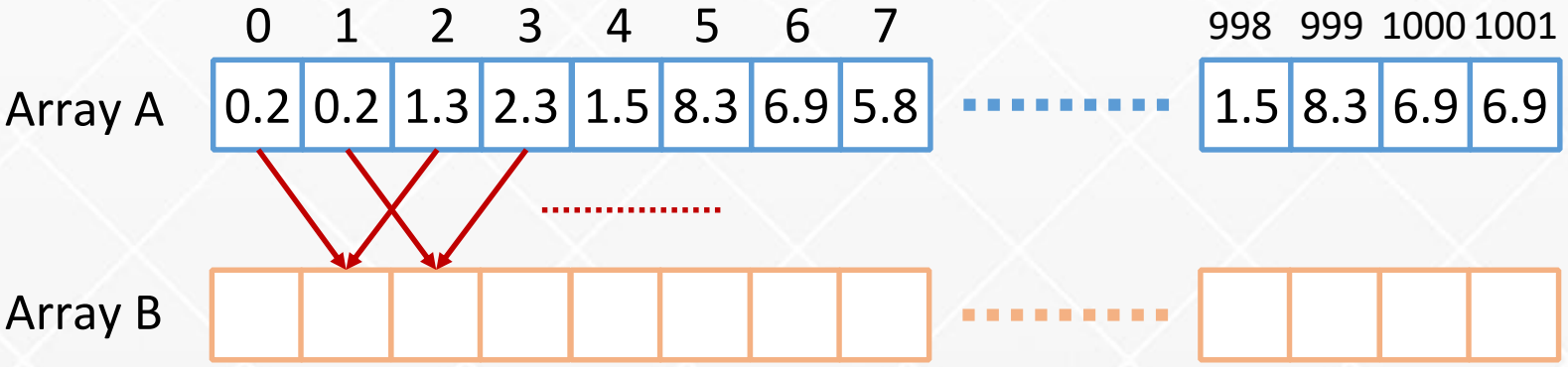
Distributed memory parallel programming

- Basic structure:
  A single program is run on different CPUs with different data



Distributed memory parallel programming

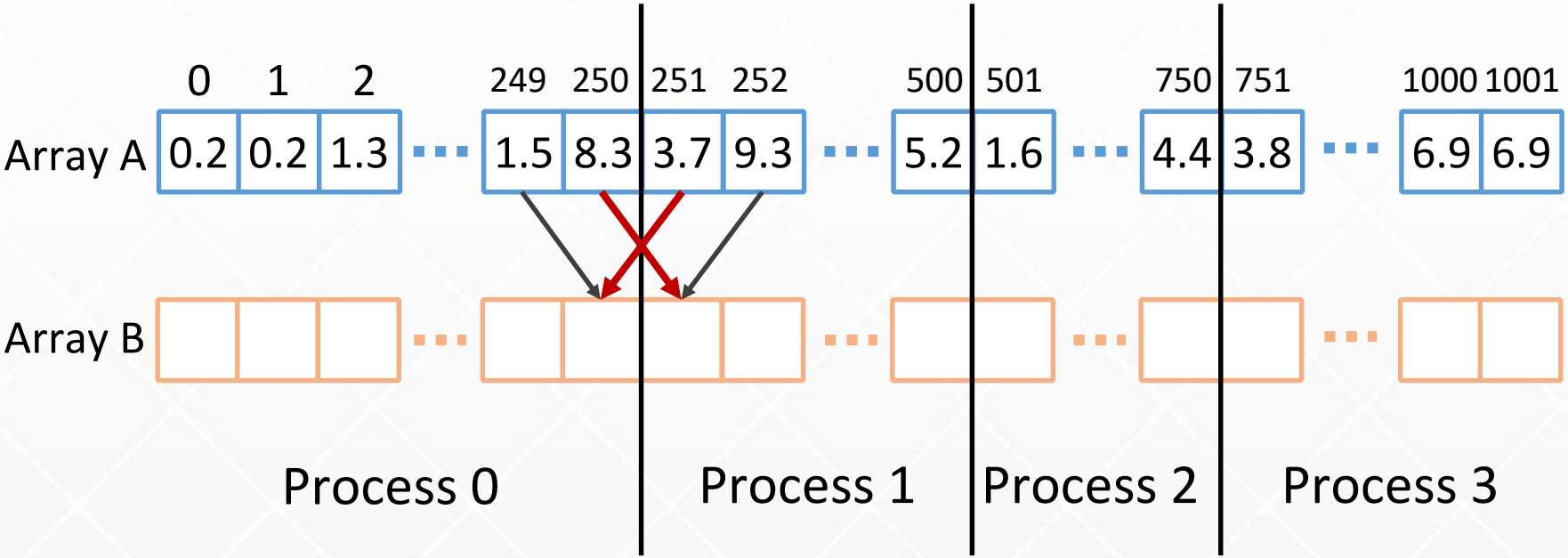
- How to write a distributed parallel program
  1. Divide data
  2. Write a program so that each node processes its own data
- What's difficult is some data must be shared between multiple processes



Let's do this calculation with 4 processes

Distributed memory parallel programming

- Divide the arrays A and B

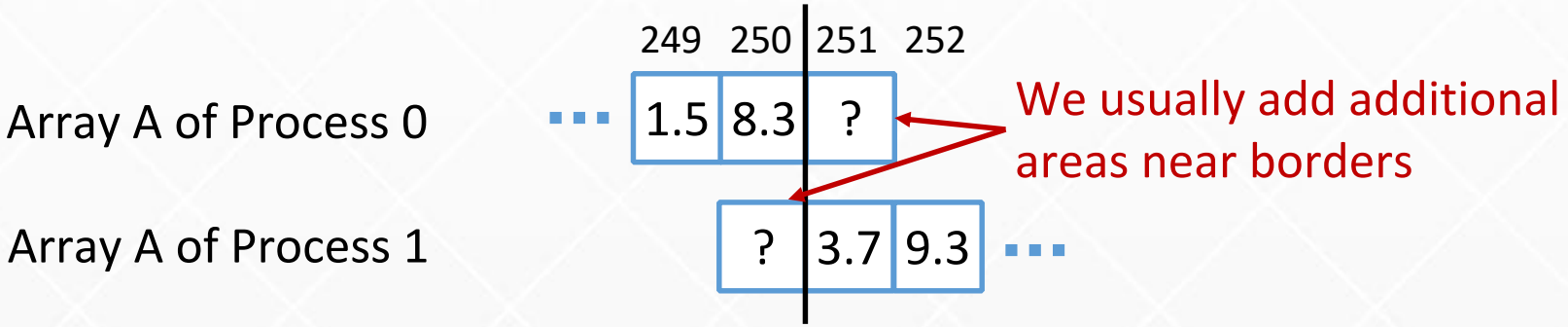


Let's recall b[i] requires a[i-1] and a[i+1]

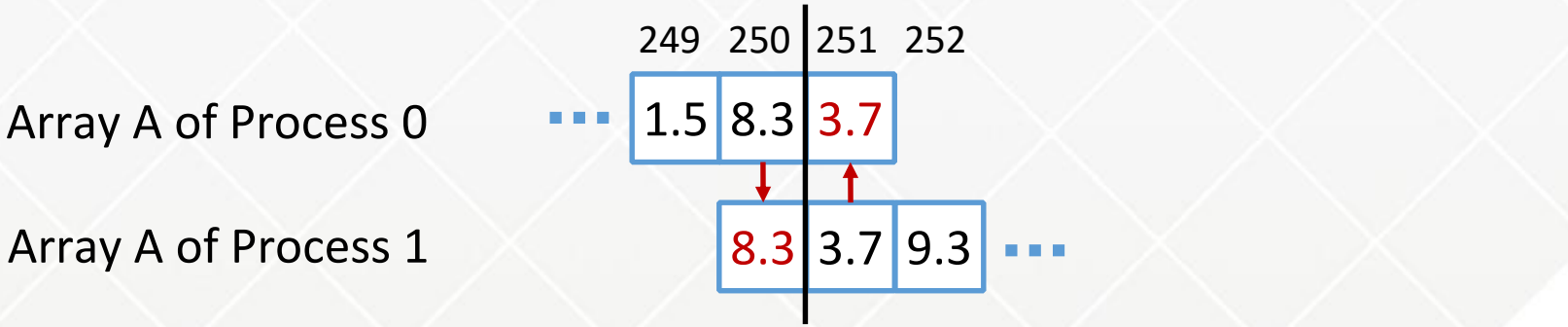
Distributed memory parallel programming

- Communication is usually required in distributed parallel programming

Step 1: Initial state

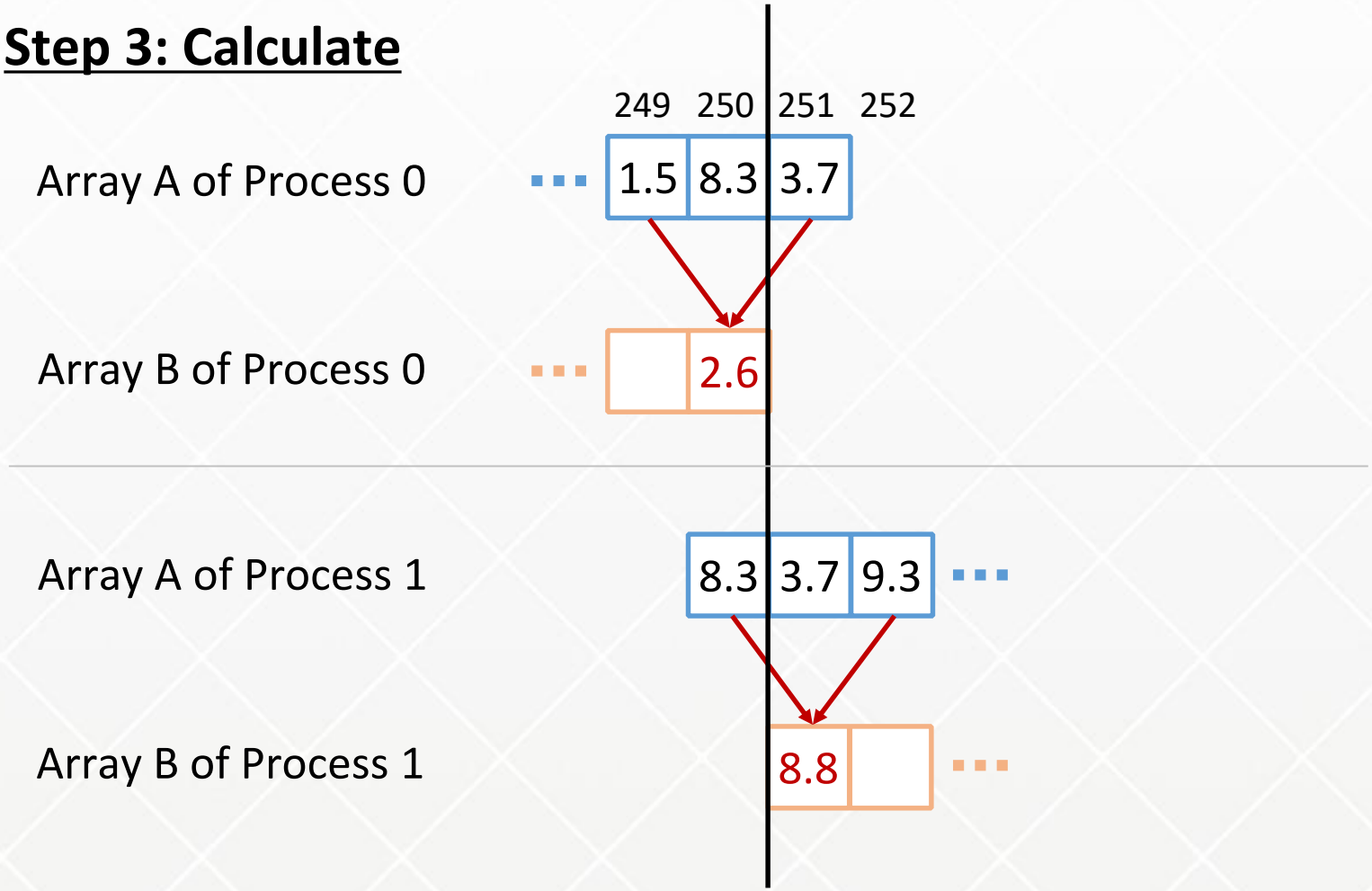


Step 2: Communicate



Distributed memory parallel programming

- Communication is usually required in distributed parallel programming



Distributed memory parallel programming

- Example program
  - Many tedious parts are simplified

```
char a[252], b[252]; /* We only have ¼ of the array */

int main(int argc, char *argv[]) {
   int myrank = Get_my_rank();       /* shown later */
   initialize_a(myrank);             /* load only my part */

   /* Pay attention to the first and last processes! */
   send_border_left_to_right(a, myrank);
   send_border_right_to_left(a, myrank);

   for(int i = 1; i <= 250; i++) {
       b[i] = (a[i-1] + a[i+1]) / 2.0;
   }

   /* NOTE: looking at the resulting b[] is more difficult
      than in a serial version because every node has only
      a part of b[] */
}
```

Message Passing Interface (MPI)

- MPI
  - A de facto standard communication library that provides communication APIs for data exchange in distributed memory parallel programs
  - Benefits
    - MPI program can run almost on any supercomputers
    - MPI is usually optimized by the administrator of each supercomputer so that we can take full advantage of high-speed network of the supercomputer
  - Problems
    - Not very easy to program with

Message Passing Interface (MPI)

- Example
  - Transferring a message between the two processes

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

int main(int argc, char *argv[]) {
    int nprocs, myrank;
    double number;

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    if (myrank == 0) {
        number = 1.0;
        MPI_Send(&number, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
    } else if (myrank == 1) {
        MPI_Recv(&number, 1, MPI_DOUBLE, 1, 1, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        printf("recv: %f¥n", number);
    }
    MPI_Finalize();
}
```

Message Passing Interface (MPI)

- How to compile and run MPI programs
  - An example of running C program on a standard PC cluster

```
# mpicc -O -o myprog myprog.c

# cat ~/machines
node00 slots=8
node01 slots=8
node02 slots=8
node03 slots=8

# mpiexec -hostfile ~/machines -np 32 myprog
```

  - The format of *hostfile* varies depending on the MPI implementation
  - It is usually unnecessary to provide *hostfile* in supercomputers
  - `mpiexec` part should be written in a job script at supercomputers

Message Passing Interface (MPI)

- MPI_Send/MPI_Recv
  - Sends/receives a message

```
int MPI_Send(const void *buf, int count,
             MPI_Datatype datatype, int dest, int tag,
             MPI_Comm comm)

int MPI_Recv(void *buf, int count,
             MPI_Datatype datatype, int source, int tag,
             MPI_Comm comm, MPI_Status *status)
```

  - **datatype**: MPI_INT, MPI_LONG, MPI_FLOAT, MPI_DOUBLE, …
  - **tag**: each matching pair of send/recv must have the same tag number (any integer number works)
    - I do not recommend always setting 0 (although it works)
  - **comm**: Almost always MPI_COMM_WORLD

Message Passing Interface (MPI)

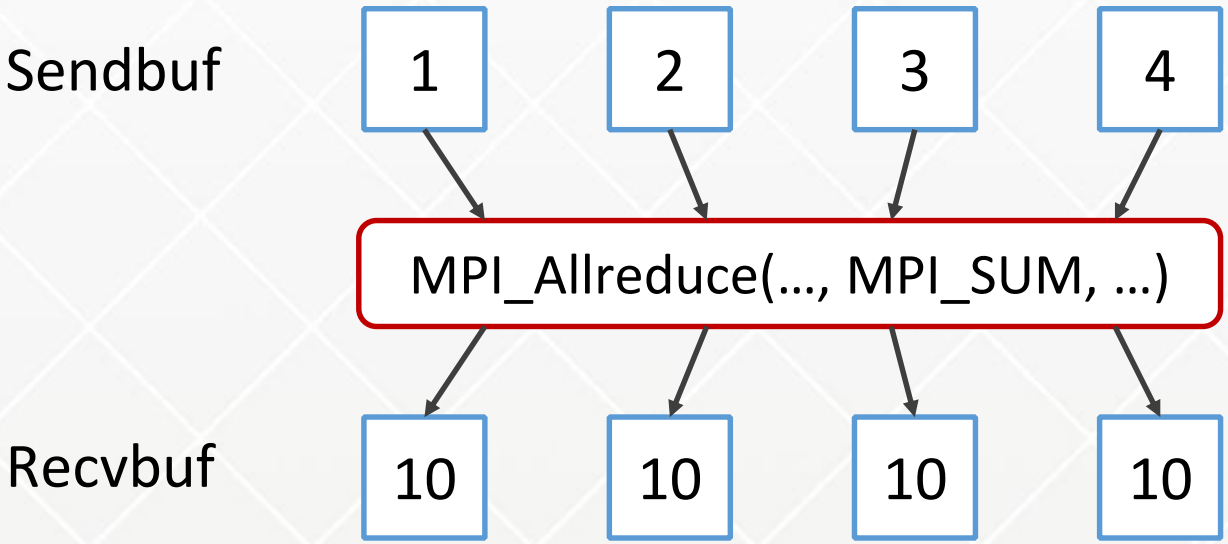
- MPI_Barrier
  - Waits for all processes to call this function

```
int MPI_Barrier(MPI_Comm comm)
```

Message Passing Interface (MPI)

- MPI_Allreduce
  - Combines values from all processes and distributes the result back to all processes

```
int MPI_Allreduce(const void *sendbuf, void *recvbuf,
                  int count, MPI_Datatype datatype,
                  MPI_Op op, MPI_Comm comm)
```



Message Passing Interface (MPI)

- Other important functions
  - MPI_Sendrecv
  - MPI_Bcast
  - MPI_Gather/MPI_Allgather
  - MPI_Alltoall
  - …

Hands-on session

Preparation

Let's write the following program

If you want to use C (hello.c)

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

int main(int argc, char *argv[]) {
   int myrank, nnodes;

   MPI_Init(NULL, NULL);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   MPI_Comm_size(MPI_COMM_WORLD, &nnodes);

   printf("Hello MPI (C) from %d/%d¥n", myrank, nnodes);

   MPI_Finalize();
   return 0;
}
```

Preparation

Let's write the following program

If you want to use Fortran (hello.f90)

```
program hello_mpi
  use mpi
  implicit none
  integer:: ierr
  integer:: myrank, nnodes

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, nnodes, ierr)

  write (*, '(a,i2,a,i2)') 'Hello MPI (Fortran) from ', &
        myrank, '/', nnodes

  call MPI_FINALIZE(ierr)
end
```

Preparation

Let’s compile the program

Compiling C code

```
klogin3$ mpifccpx  -o hello hello.c
```

Compiling Fortran code

```
klogin3$ mpifrtpx  -o hello hello.f90
```

Nothing will be displayed if you wrote the program correctly

Preparation

Let’s prepare the following job script

Job.sh

```
#!/bin/bash -x
#
#PJM --rsc-list "node=8"
#PJM --mpi "proc=8"
#PJM --mpi "rank-map-bynode"
#PJM --rsc-list "elapse=00:01:00"
#PJM --stg-transfiles all
#PJM --stgin "./hello* ./"
#PJM -s

. /work/system/Env_base

mpiexec -np 8 ./hello
```

Preparation

Let's submit the job

```
klogin3$ pjsub job.sh
```

Check job status by `pjstat` command

```
klogin3$ pjstat
  ACCEPT QUEUED  STGIN  READY RUNNING RUNOUT STGOUT   HOLD  ERROR   TOTAL
       0      1      0      0       0      0      0      0      0       1
s      0      1      0      0       0      0      0      0      0       1

JOB_ID   JOB_NAME  MD  ST   USER      GROUP      START_DATE       ELAPSE_TIM  NODE_RE
7414437  job.sh    NM  QUE  a03573    ra001016  [--/-- --:--:--]  0000:00:00      8:-
```

`pjstat` will display as follows if your job is done

```
klogin3$ pjstat
  ACCEPT QUEUED  STGIN  READY RUNNING RUNOUT STGOUT   HOLD  ERROR   TOTAL
       0      0      0      0       0      0      0      0      0       0
s      0      0      0      0       0      0      0      0      0       0
```

Preparation

Let's look at the results

```
klogin3$ ls -ltr job.sh.o*
(pick up the last one)

klogin3$ cat job.sh.o7414437
Env_base: K-1.2.0-24
Hello MPI (C) from 3/8
Hello MPI (C) from 4/8
Hello MPI (C) from 0/8
Hello MPI (C) from 1/8
Hello MPI (C) from 2/8
Hello MPI (C) from 6/8
Hello MPI (C) from 7/8
Hello MPI (C) from 5/8
```

If you see the output like this, you are all set!

Problem 1

1. Let's write the following simple MPI program and run it with two nodes (either in C or Fortran)

```
#define BUFSIZE 1024
char sendbuf[BUFSIZE], recvbuf[BUFSIZE];

int main(int argc, char *argv[]) {
   int msgsize = 1024;
   1. Initialize MPI
   2. Write something to the entire sendbuf[]
   3. Wait until two process reaches here (MPI_Barrier)
   4. Get the current time (MPI_Wtime)
   5. Send a message with the size "msgsize" from
      processes 0 to 1
   6. Send back the message from 1 to 0
      (use recvbuf at 0)
   7. Get the current time again
   if (myrank == 0) {
      8. Check if sendbuf and recvbuf have same data
      9. Calculate the elpsed time on process 0
      10. Calculate the bandwidth (MB/s) and print it
   }
   11. Finalize MPI
}
```

Problem 1

2. Rewrite the program to iterate over various buffer and message size
   - 4, 8, 16, 32 byte, …, 128 MByte
3. Plot the resulting bandwidth (if you have time)

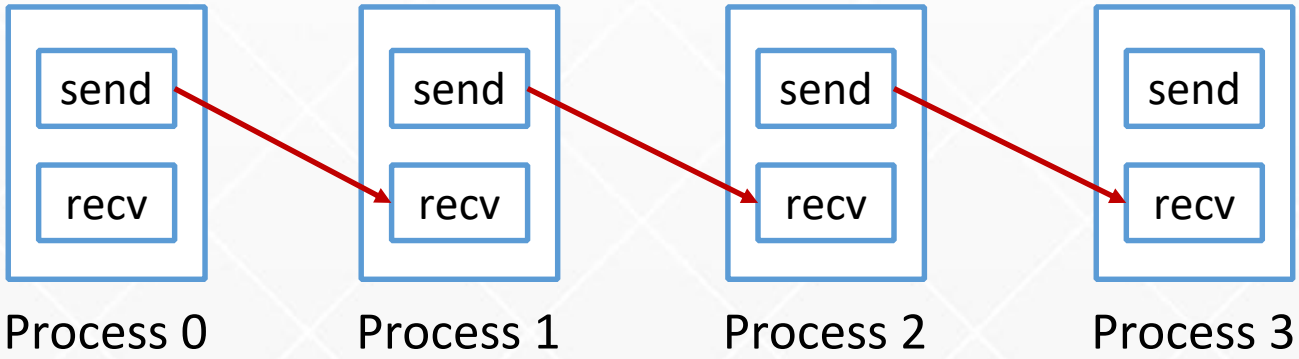
NOTE:

Don’t forget to change the name of your executable program

```
#!/bin/bash -x
#PJM --rsc-list "node=2"
#PJM --mpi "proc=2"
#PJM --mpi "rank-map-bynode"
#PJM --stgin "./problem1 ./“
…

mpiexec -np 2 ./problem1
```

Problem 2

1. Let's write a program in which each process sends data to the next process
   - Your program must work with any number of processes
   - Check if communication was correctly performed

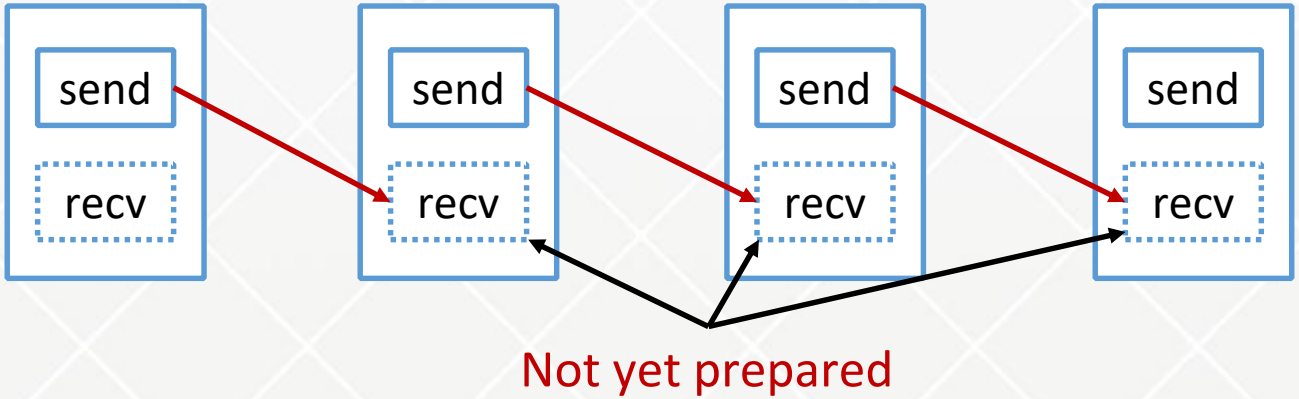


   - Hint

```
int MPI_Sendrecv(const void *sendbuf, int sendcount,
                 MPI_Datatype sendtype, int dest,
                 int sendtag, void *recvbuf,
                 int recvcount, MPI_Datatype recvtype,
                 int source, int recvtag, MPI_Comm comm,
                 MPI_Status *status)
```

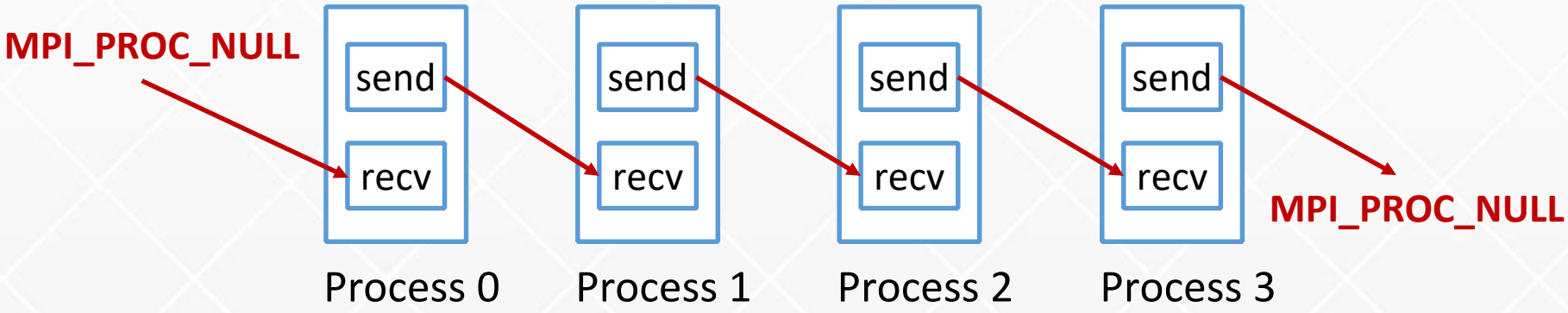
Problem 2 (IMPORTANT!)

- Writing MPI_Send and MPI_Recv separately is wrong even if it looks working
  - Writing MPI_Recv first is obviously wrong (no one sends data)
  - Writing MPI_Send first is also wrong. It causes deadlocks because there are no spaces to store incoming data
    - It does not always deadlock due to internal buffers of MPI library (never rely on this buffering!)
    - Another way to avoid deadlock: using non-blocking communication (MPI_Irecv, MPI_Wait)



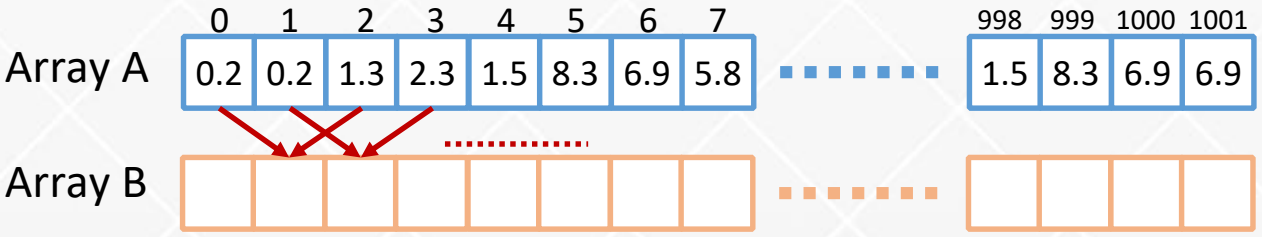
Problem 2

- If you set MPI_PROC_NULL as destination or source rank, communication functions returns immediately without doing anything
- It allows us to handle the first and last process in the same way as other processes



Problem 3

- Let's write a program of today's example application with MPI
  - Hints are available around slide 16
  - You may initialize the array A as you like, but let's assume each node knows only its part of the array (so you have to exchange values)
  - You may make some restrictions on the size of array and the number of processes
    - E.g.) Array size is a multiple of the number of processes




```
for(int i = 1; i <= 1000; i++) {
    b[i] = (a[i-1] + a[i+1]) / 2.0;
}
```

Problem 3

- This problem is a great preparation of tomorrow's lecture
- If you have time, let's parallelize the calculation in each process using OpenMP
  - Use -Kopenmp option to compile your OpenMP program
  - You may not see performance gain unless you iterate the calculation over and over again
    - Copy back the array B to A and do the same thing again
  - Set OMP_NUM_THREADS in your job script
    - The number of CPUs (cores) of each node of K computer is 8

```
#!/bin/bash -x
# PJM…
…
export OPM_NUM_THREADS=8
mpiexec …
```