High Performance Computing

高性能計算論

Volume 4

Cyberscience Center, Tohoku Univ

Hiroyuki Takizawa
<takizawa@tohoku.ac.jp>

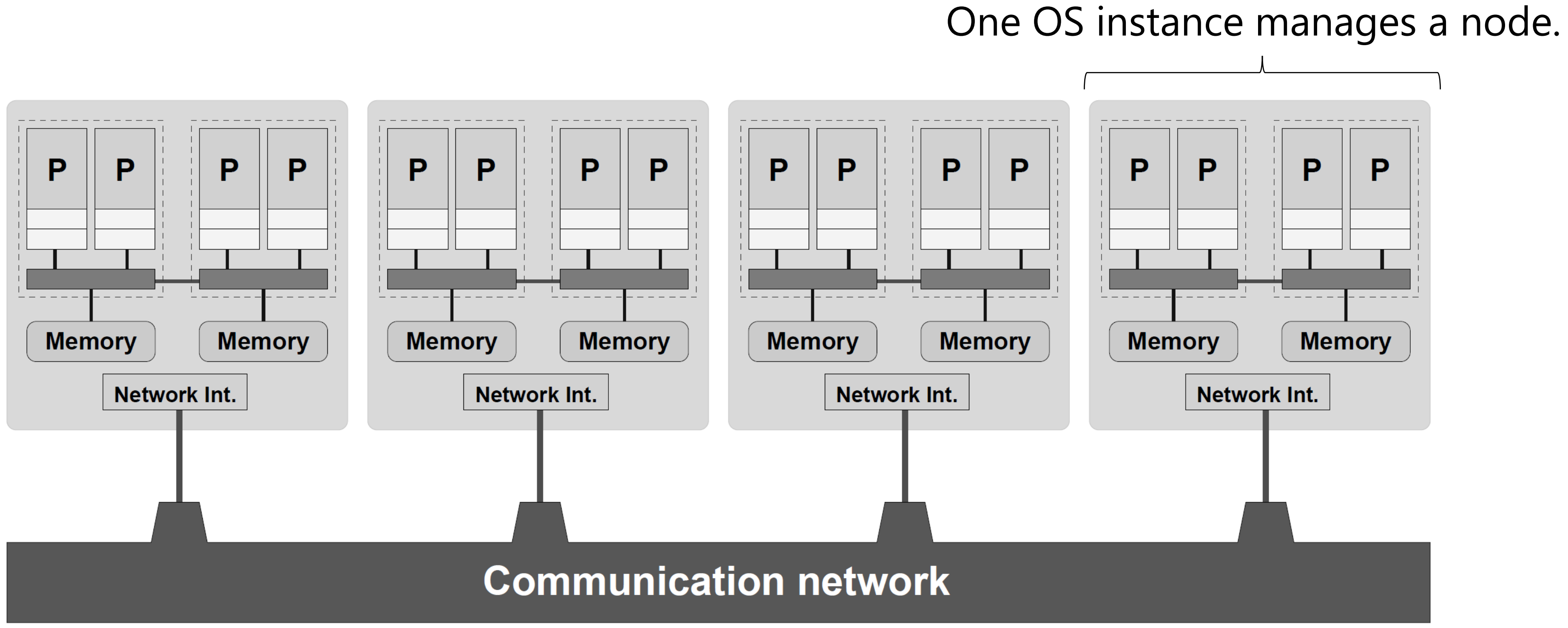
What you learnt so far (1/2)

Parallel Computers

- Vector/SIMD
- Shared-memory computers
- Distributed-memory computers
- Hierarchical (hybrid) systems
- Networks

Hybrid System

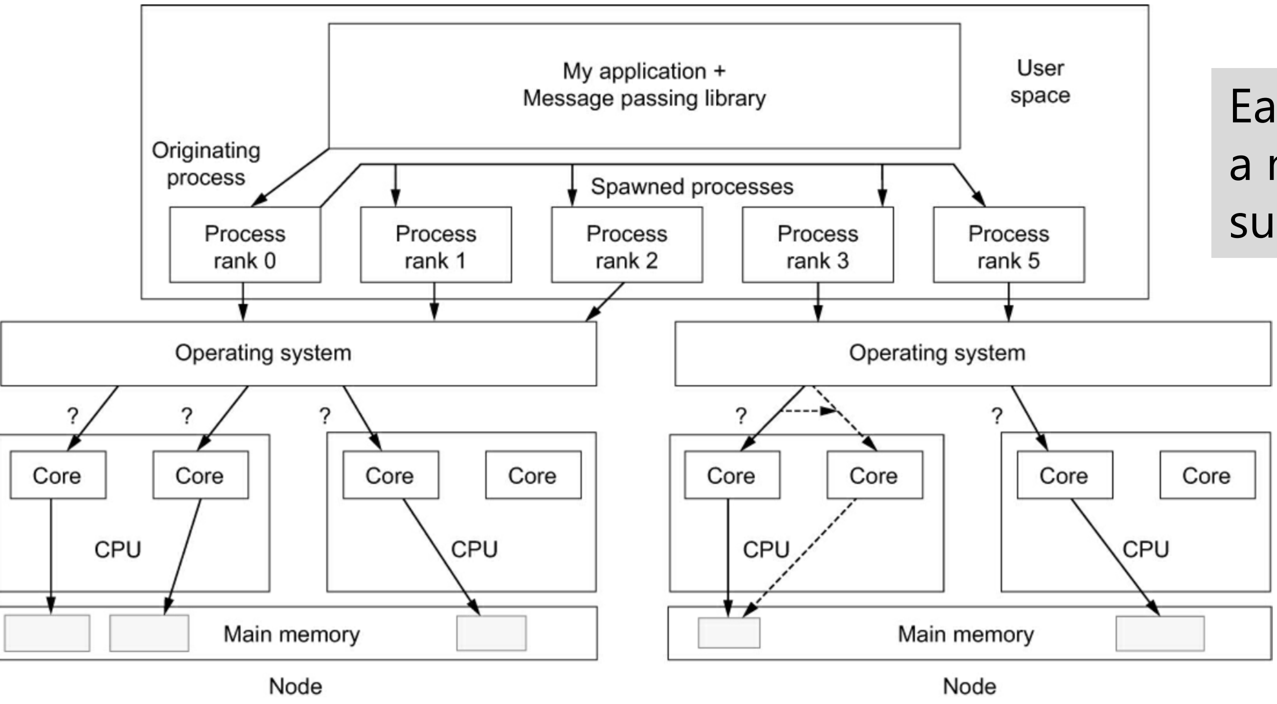
- Large-scale parallel computers
  = mixture of shared and distrib.-parallel.



In addition, each node may have accelerators such as GPUs.

Software Overview

When a program is launched, several kinds of resources such as CPU time and memory are allocated for the execution. A unit of allocated resource is called a **process**. An **application** (user program) is executed by multiple processes.



Each process is assigned to a node, which runs the **OS**, such as **Linux**.

The OS on each node decides the core(s) to execute the process. A process can be executed by multiple cores. The execution sequence on each core is called a **thread**.

What you learnt so far (2/2)

- **Job Level Parallelism**
  - What is Job?
  - Job Scheduling
- **Parallel Algorithm Design**
  - Task/Channel Model
  - Foster's Design Methodology
  - Communication Patterns

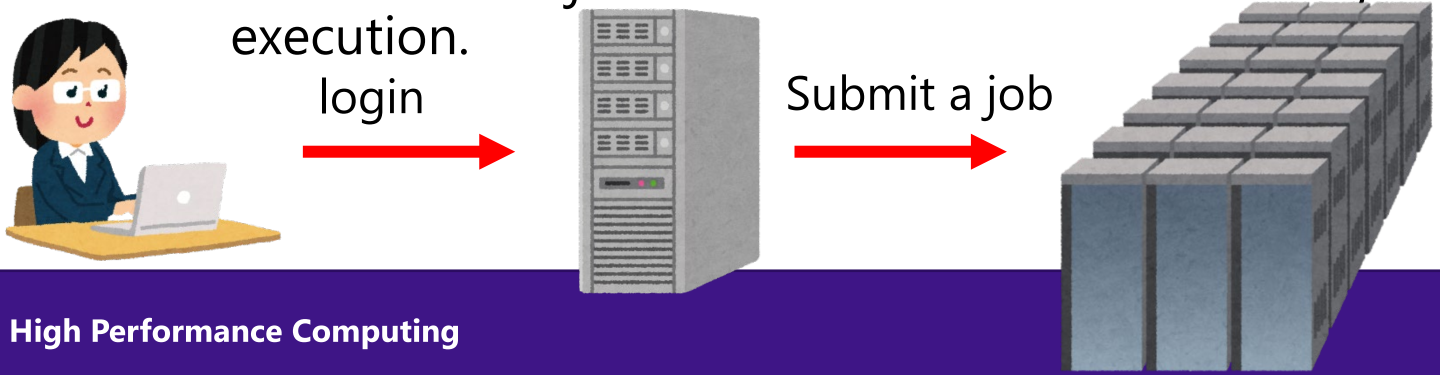
How to run a program on HPC?

■ Batch Job

- **A unit of work from user's point of view**
  - Submitted to an HPC System
- A job is usually a batch of tasks
  - Task is a unit of work for a computer

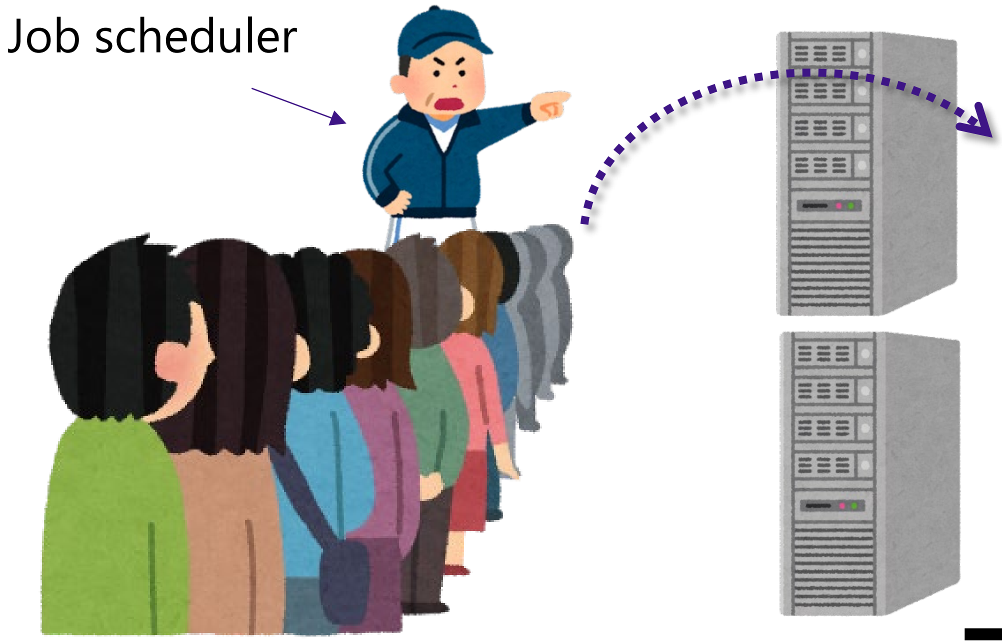
■ Front-end Server and Compute Nodes

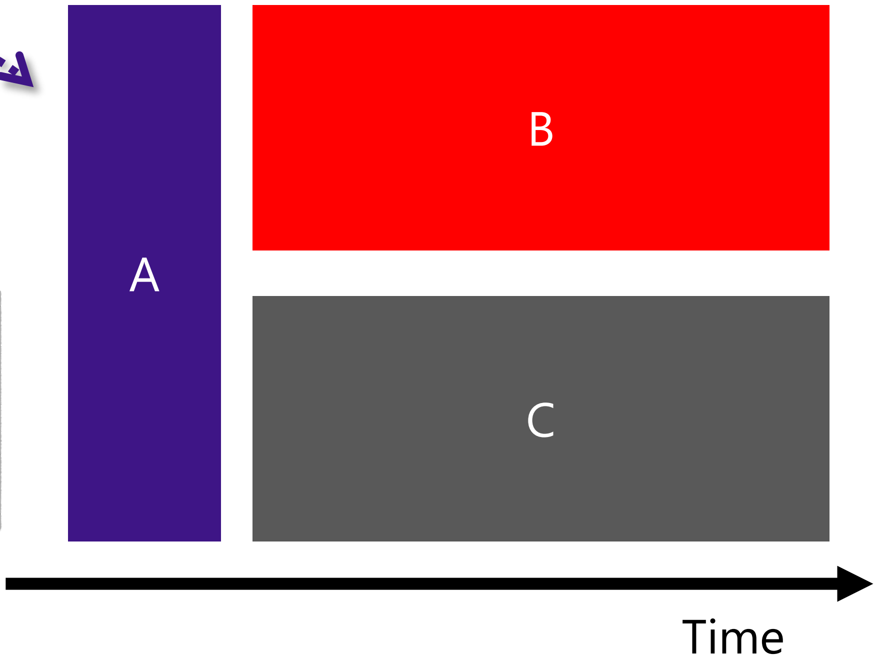
- Users can log-in to the front-end server, but not directly to the compute nodes
- How to run a job on compute nodes?
  - User submit a job from the front-end server for job execution.



Job Scheduling

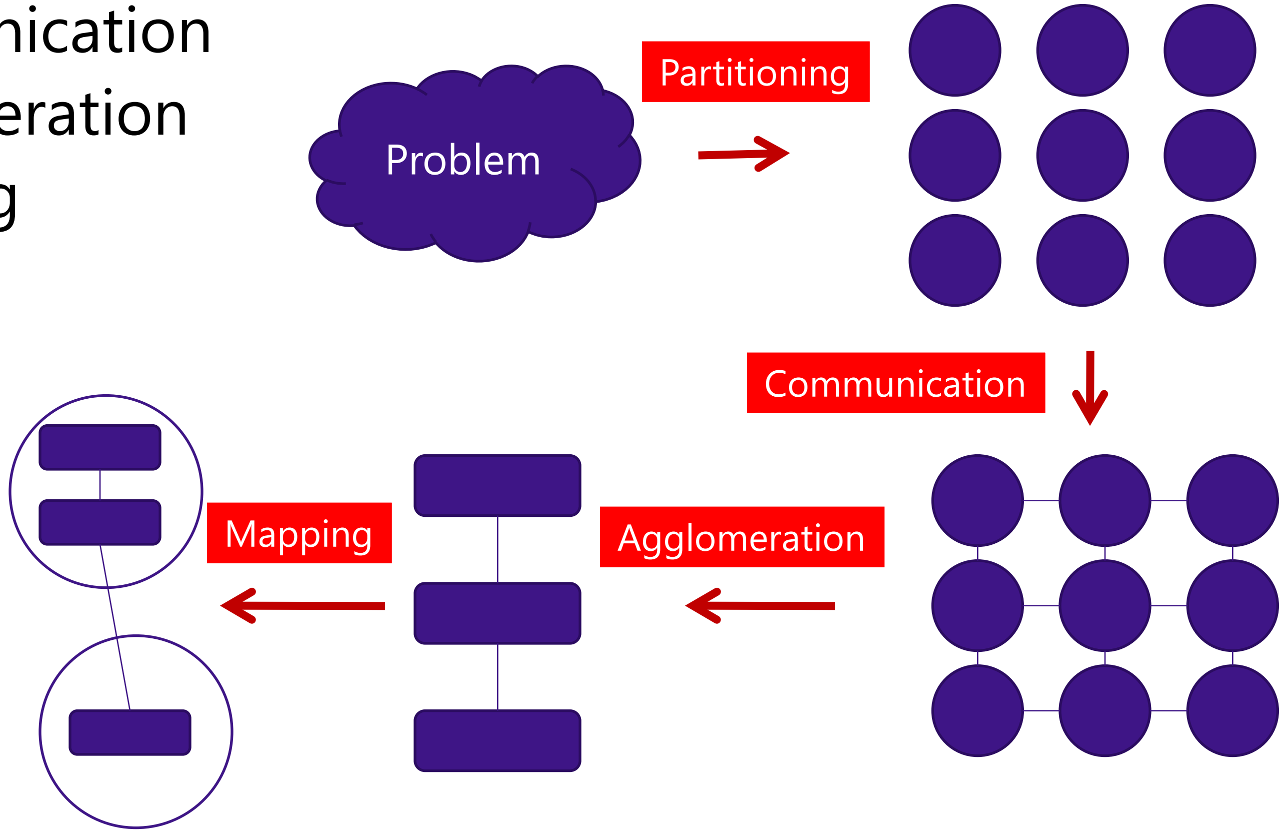
- **Decide where and when a job is executed**
  - Necessary for efficient use of shared resources
  - The most basic policy is **First Come First Serve** (**FCFS**) policy.





Foster's Design Methodology

- Four steps for designing parallel algorithms
  - Partitioning
  - Communication
  - Agglomeration
  - Mapping



Today's Topic

■ Introduction to MPI Programming

- A minimal MPI program
- Performing common communication patterns with collective MPI calls
- Peer-to-peer communication for data exchange
- Benchmarking

You may use any software for programming.
For example, I use **Visual Studio Code with Remote SSH ext.**

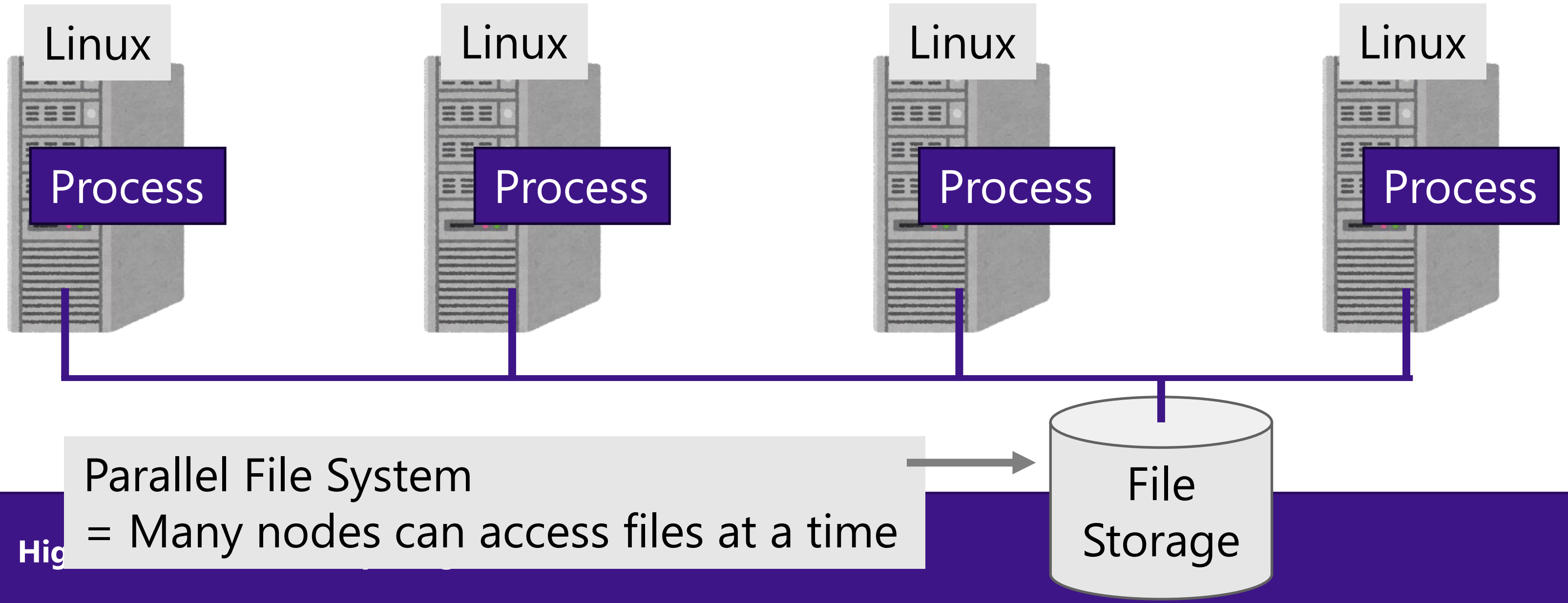
Programming Parallel Computers

- **Four distinct paths (McGraw and Axelrod, 1998)**
  - Extend an existing compiler
    - Translate a sequential programs into parallel ones
  - Extend an existing language
    - Provide new operations to express parallelism
  - A new parallel language layer
    - Added on top of an existing sequential language
  - A totally new language and compiler system
    - Fortran90, High Performance Fortran, and C*
- **C with MPI and/or OpenMP**
  - An existing language with low-level constructs

How to use multiple nodes?

Parallel Computer = multiple nodes

- Nodes are connected via high-speed network
- Parallel file system is shared by nodes in most cases
- Each node is managed by an OS instance.
  - OS on each node allocates a set of computing resources (=**process**) for program execution.
  - **A process must be created on every node.**



What we need?

- **Parallel computing with multiple nodes**
  = Parallel computing with multiple processes
  - At least one process on every node
    - Otherwise, there is no available resource on the node.

  Program launching mechanism

  - Data communication among processes
    - Otherwise, each process cannot access data of others.

  Data exchange among nodes (= **Message Passing**)

  - Synchronization among processes
    - Otherwise, each process can run only independently.

  Synchronization and/or blocking communication

What is MPI?

■ Message Passing Interface (MPI)

- Interface for parallel programs with **message passing**.
  - Multiple processes (**MPI processes**) run on a parallel computer.
    - Each MPI process has its own memory space.
    - MPI processes can pass their data to others if necessary.
  - MPI defines only the interface (not the implementation).
    - MPI defines how each MPI function should work.
    - We do not need to care about the implementation nor internal behaviors.

■ Major MPI Implementations

- MPICH (http://www.mpich.org/)
- Open MPI (http://www.open-mpi.org/)
- MVAPICH (https://mvapich.cse.ohio-state.edu/)
  - Computer vendors also provide their own implementations

Naming Conventions

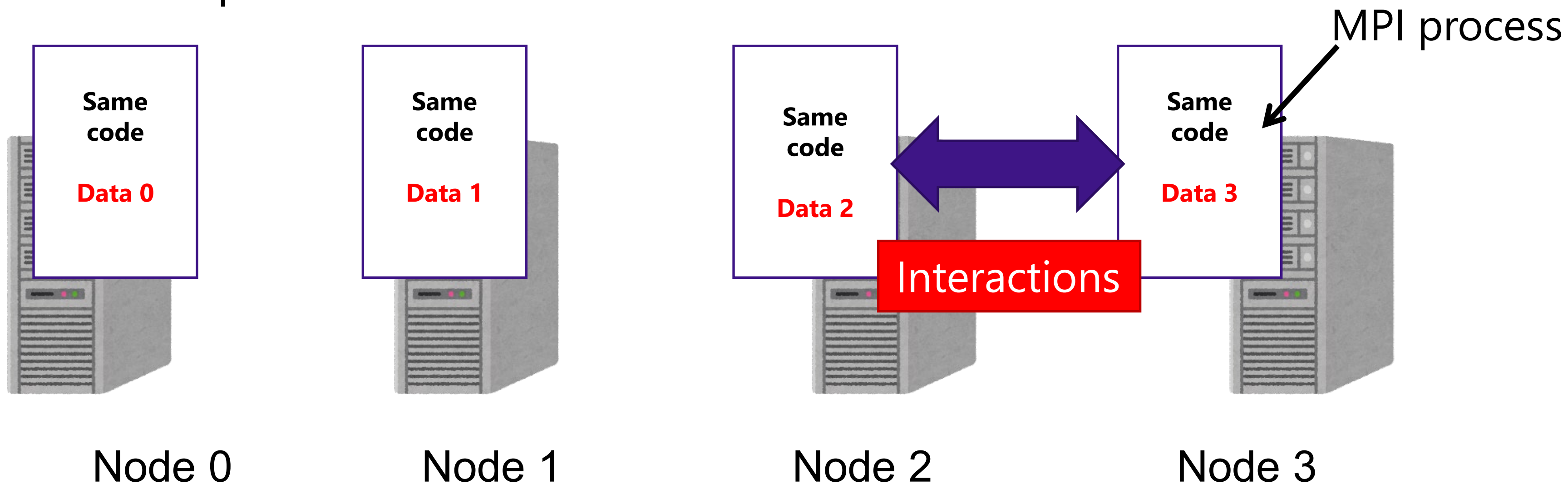
- **C Function Names**
  - C function name is a format of **`MPI_Xxx_yyy`**
    - `MPI_Init, MPI_Finalize, MPI_Send, MPI_Recv, …`
  - The error code is returned
    - `ierr = MPI_Comm_size (MPI_COMM_WORLD, &a);`

- **Keywords (Macro Definitions)**
  - Keyword is a format of **`MPI_AAA_BBB`**
    - `MPI_COMM_WORLD, MPI_INT, MPI_SUM, …`

- **Fortran Function Names**
  - Fortran is case-insensitive
    - `MPI_INIT, mpi_init, mpi_send, MPI_RECV, …`
  - Error code is passed via an argument
    - `call mpi_comm_size (MPI_COMM_WORLD, a, ierr)`

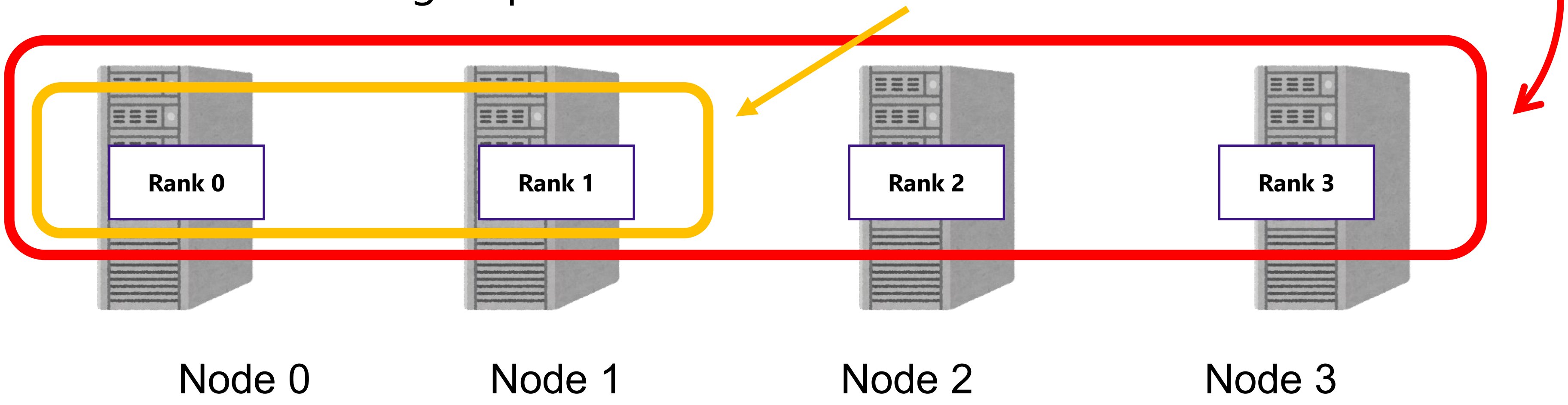
MPI Programming Model

- **Single-Program Multiple-Data (SPMD)**
  - The same program is executed on multiple nodes.
    - Programmers write only one source code
    - An **MPI process** is launched on each node.
  - Each MPI operates on different data
    - MPI is to explicitly describe interactions among MPI processes



MPI Programming Model

- **MPI Rank = ID number of each process**
  - Each MPI process has a unique ID number (=rank)
- **Communicator = Group of MPI processes**
  - MPI processes can be grouped into a communicator
  - All MPI processes are included in `MPI_COMM_WORLD`.
  - Each MPI process may join multiple communicators.
    - A subgroup can be defined as **another communicator**.



Minimal MPI Program

```
#include <mpi.h>

int main(int argc, char** argv)
{
  MPI_Init(&argc, &argv);
  MPI_Finalize();
  return 0;
}
```

Each MPI function name is a format of **MPI_Xxxx_yyyy**

Compiling and Running MPI program

Compiling an MPI program

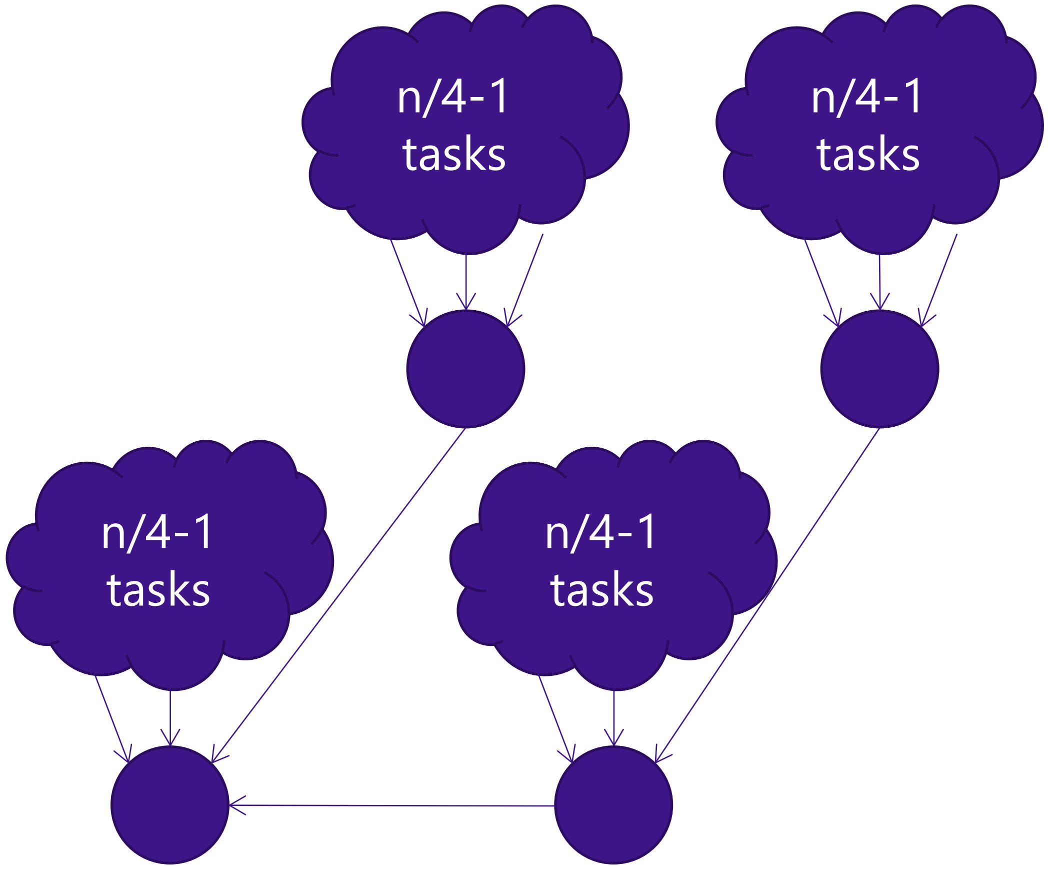
- `mpicc -o min-mpi min-mpi.c`
  - `mpicc` is a complier command
  - C code program named "min-mpi.c" is converted to an executable "min-mpi".

Launching an MPI program

- `mpirun -np 1 ./min-mpi`
  - Run "min-mpi" on a single node
- `mpirun -np 4 ./min-mpi`
  - Run "min-mpi" on 4 nodes

Parallel Reduction

- Finding the sum of *n* values



MPI functions

MPI_Init & MPI_Finalize

- Startup and cleanup an MPI runtime environment
  - MPI_Init must be called before calling any other MPI functions
  - MPI_Finalize must be called at the end of program execution

MPI_Comm_size

- Retrieves the number of processes in a communicator.
  - MPI processes can be grouped into communicators.
  - `MPI_COMM_WORLD` is the default communicator that contains all MPI processes working together.

MPI_Comm_rank

- Retrievs the rank of the process.
  - Rank is an ID assigned to each process.
  - MPI process can operate different data by using the rank.

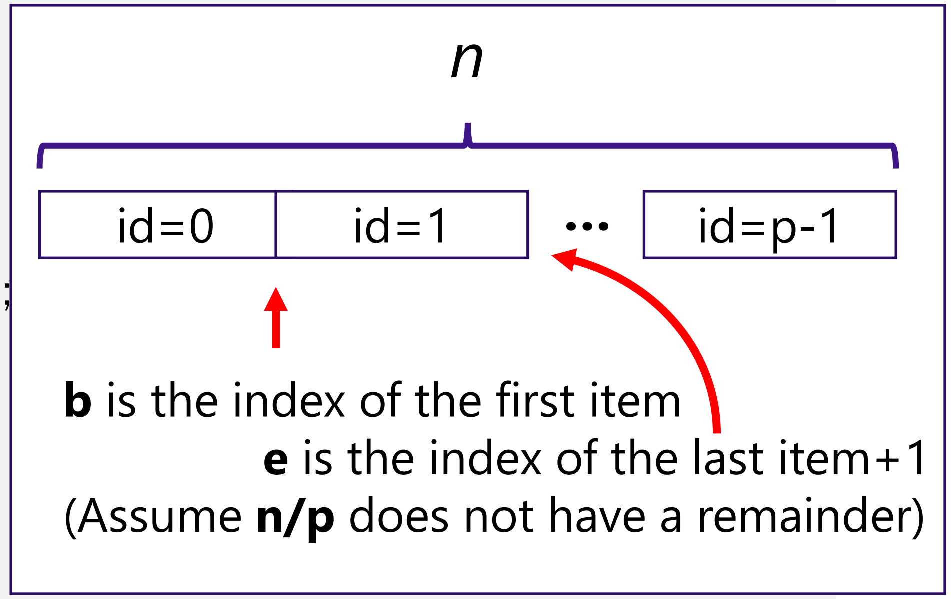
Sample Code ver. 1

```
#include <mpi.h>    /* MPI functions */
#include <stdio.h>  /* fprintf      */

int main(int argc, char* argv[])
{
  int i, id, p, b, e, s=0, n=10000;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
  b = (n/p)*id;
  e = (n/p)*(id+1);

  for(i=b;i<e;i++)
     s += i;
  fprintf(stderr, "Process %d is done. ", id);
  MPI_Finalize();
  return 0;
}
```



b is the index of the first item
e is the index of the last item+1
(Assume **n/p** does not have a remainder)

Compiling and Running MPI programs

Compiling MPI programs

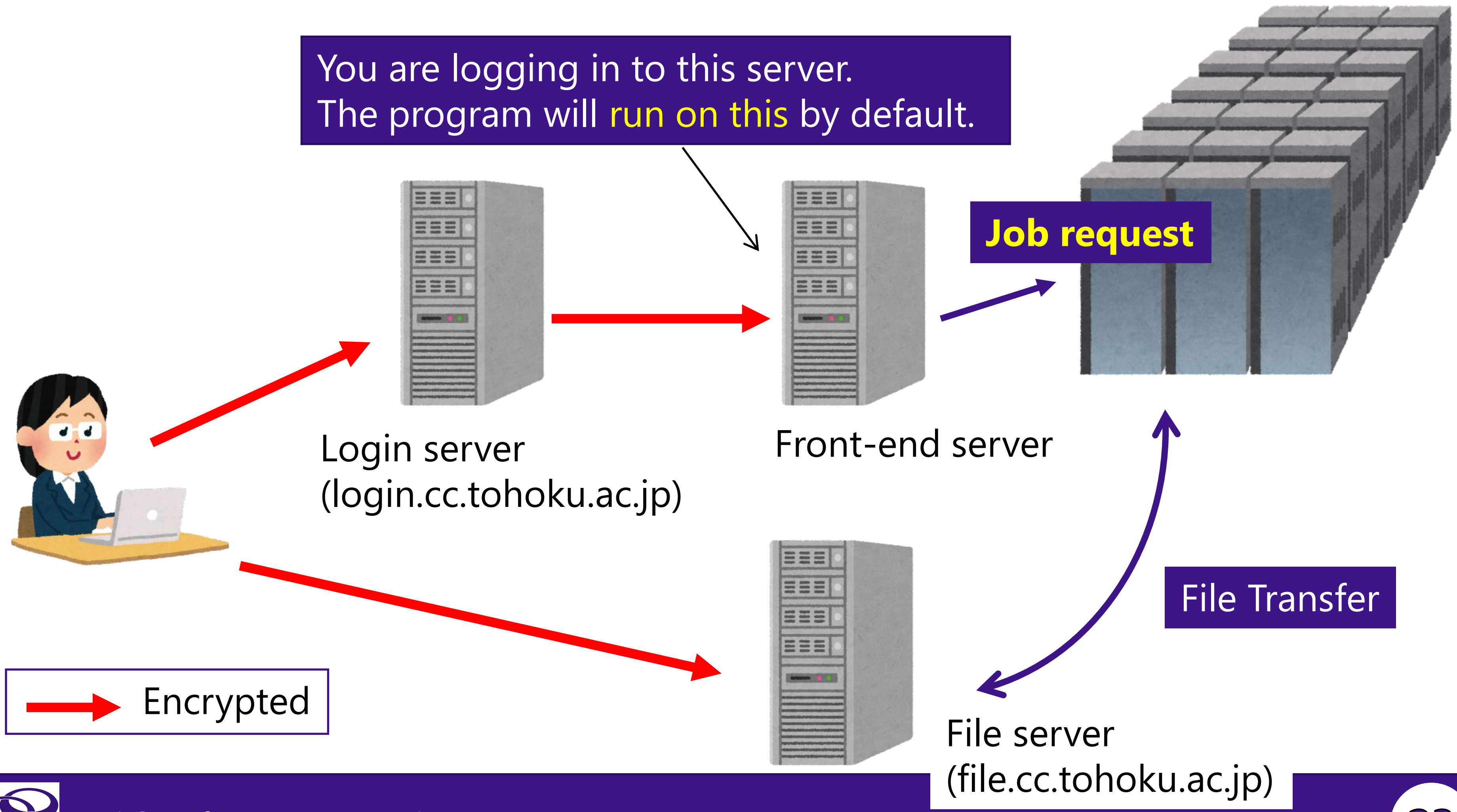
- `mpicc -o sum1 sum1.c`
  - `mpicc` is a complier command
  - C code "sum1.c" is converted to an executable "sum1".

Running MPI programs

- `mpirun -np 1 ./sum1`
  - Run "sum1" on a single node
- `mpirun -np 4 ./sum1`
  - Run "sum1" on 4 nodes

Why Job Submission Needed?

- Supercomputer AOBA = **Shared** Resource



Job Submission

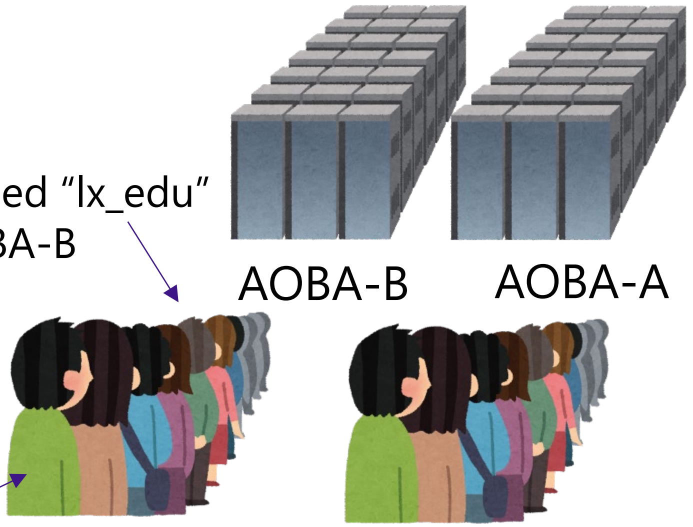
lx_edu is dedicated to this class.
Use lx usually.

■ Write a shell script file (run.sh)

```
#!/bin/sh –
#PBS -q lx_edu
#PBS -l elapstim_req=0:01:00
cd $PBS_O_WORKDIR
mpirun –np 4 ./sum1
```

This job will be executed on **AOBA-B**.

Waiting Queue named "lx_edu" associated with AOBA-B



■ Submit it to job scheduler

`qsub run.sh` Job submission

Your job will be appended at the end of the queue.

`qstat` Check the status

■ Get the result

`run.sh.exxxxx` stderr

`run.sh.oxxxxx` stdout

These files will be created in the same directory.
xxxxx is the job ID. (5 digits)

What happens?

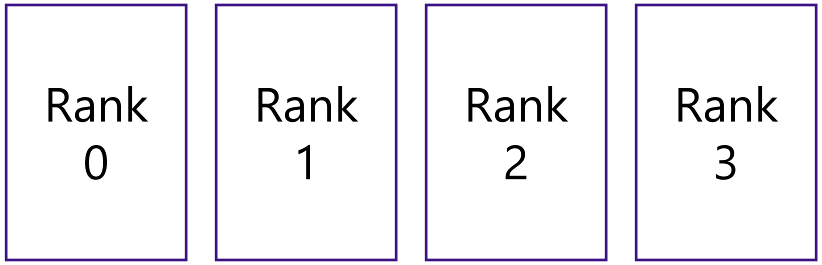
```
mpirun -np 4 sum1
```

```
Process 1 is done.
Process 0 is done.
Process 3 is done.
Process 2 is done.
```

```
less run.sh.exxxxx
```

The content of run.sh.exxxxx

Each process asynchronously works. So the messages may differ every time they run.



A process of Rank ***id*** computes a subtotal
= There is no process that has the total sum.

MPI Communications

- **Collective communications**
  - MPI implementation offers optimized implementations of typical communication patterns
    - Reduction, gather, scatter, broadcast…
- **Peer-to-peer communications**
  - In MPI, any pair of two MPI processes can communicate with each other.
    - Send/recv, Isend/irecv, Get/put,

Sample Code ver. 2

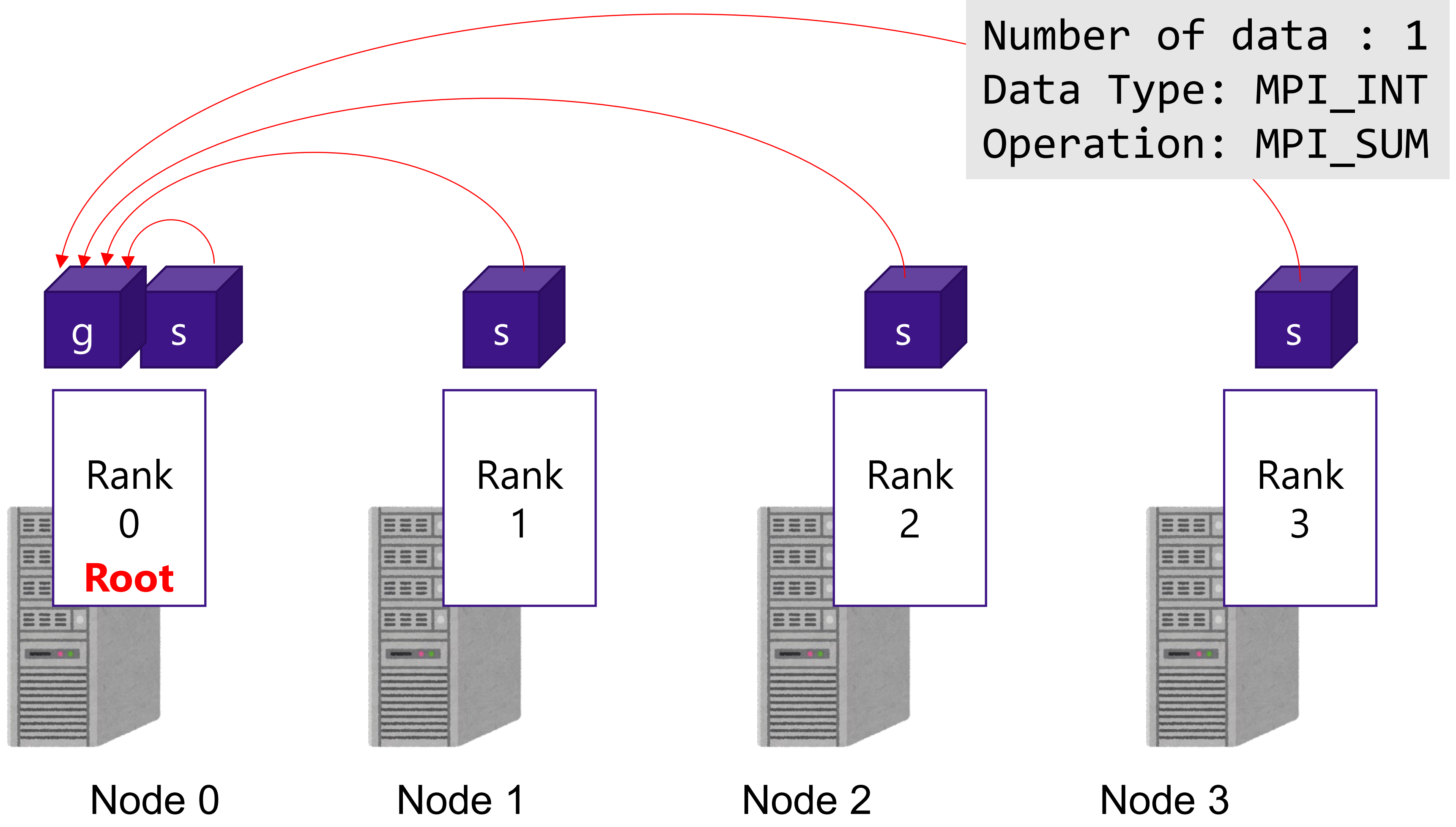
```
#include <mpi.h>    /* MPI functions */
#include <stdio.h>  /* fprintf       */

int main(int argc, char* argv[])
{
  int i, id, p, b, e, s=0, n=10000;
  int g;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
  b = (n/p)*id;
  e = (n/p)*(id+1);

  for(i=b;i<e;i++)
      s += i;
  MPI_Reduce(&s,&g,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
  if (id==0)
    fprintf(stderr, ”Total=%d¥n”, g);
  MPI_Finalize();
  return 0;
}
```

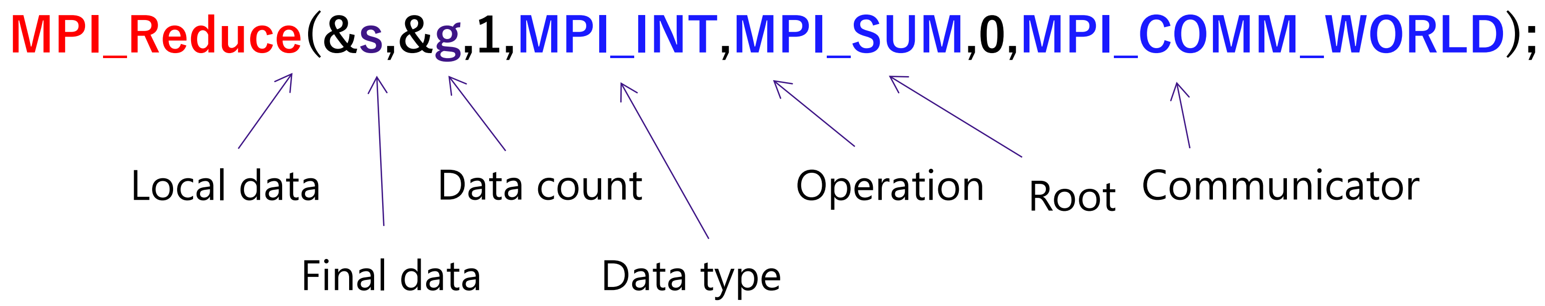
How MPI_Reduce Works



Collective Communication

■ MPI_Reduce

- One process becomes a root process of a reduction operation, gathering values from all the other processes in a communicator, and summing them up (or other operations).
- Suppose each process has an integer value **s**, and Process 0 calculates the sum of the values, **g**. Then, the reduction operation is



```
MPI_Reduce(&s,&g,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
```

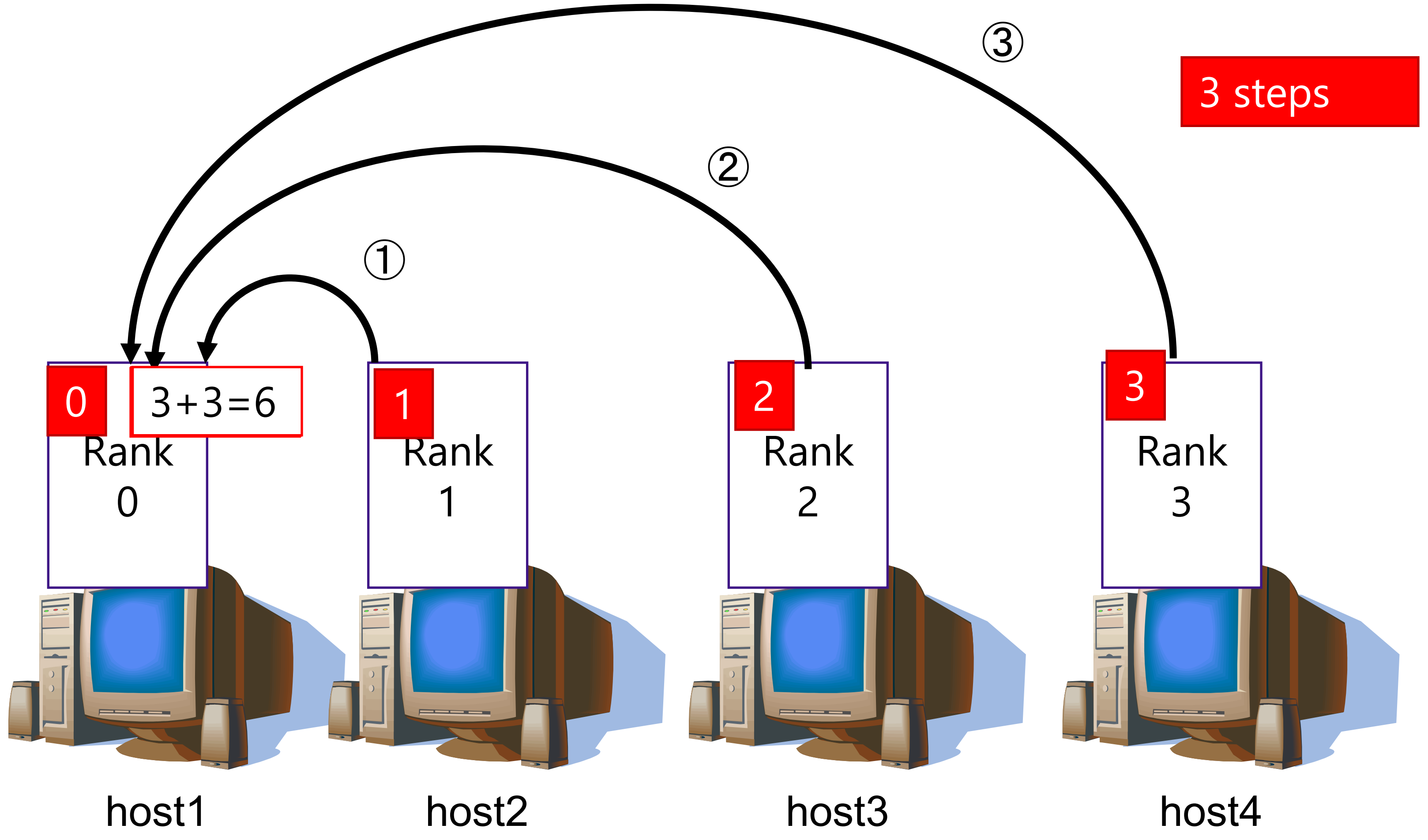
Data Types

Name	C Type
MPI_CHAR	signed char
MPI_DOUBLE	double
MPI_FLOAT	float
MPI_INT	int
MPI_LONG	long
MPI_LONG_DOUBLE	long double
MPI_SHORT	short
MPI_USIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_SHORT	unsigned short

Reduction Operators

Name	Meaning
MPI_BAND	Bitwise AND
MPI_BOR	Bitwise OR
MPI_BXOR	Bitwise eXclusive OR (XOR)
MPI_LAND	Logical AND
MPI_LOR	Logical OR
MPI_LXOR	Logical eXclusive OR (XOR)
MPI_MAX	Maximum
MPI_MAXLOC	Maximum and its location
MPI_MIN	Minimum
MPI_MINLOC	Minimum and its location
MPI_PROD	Product
MPI_SUM	Sum

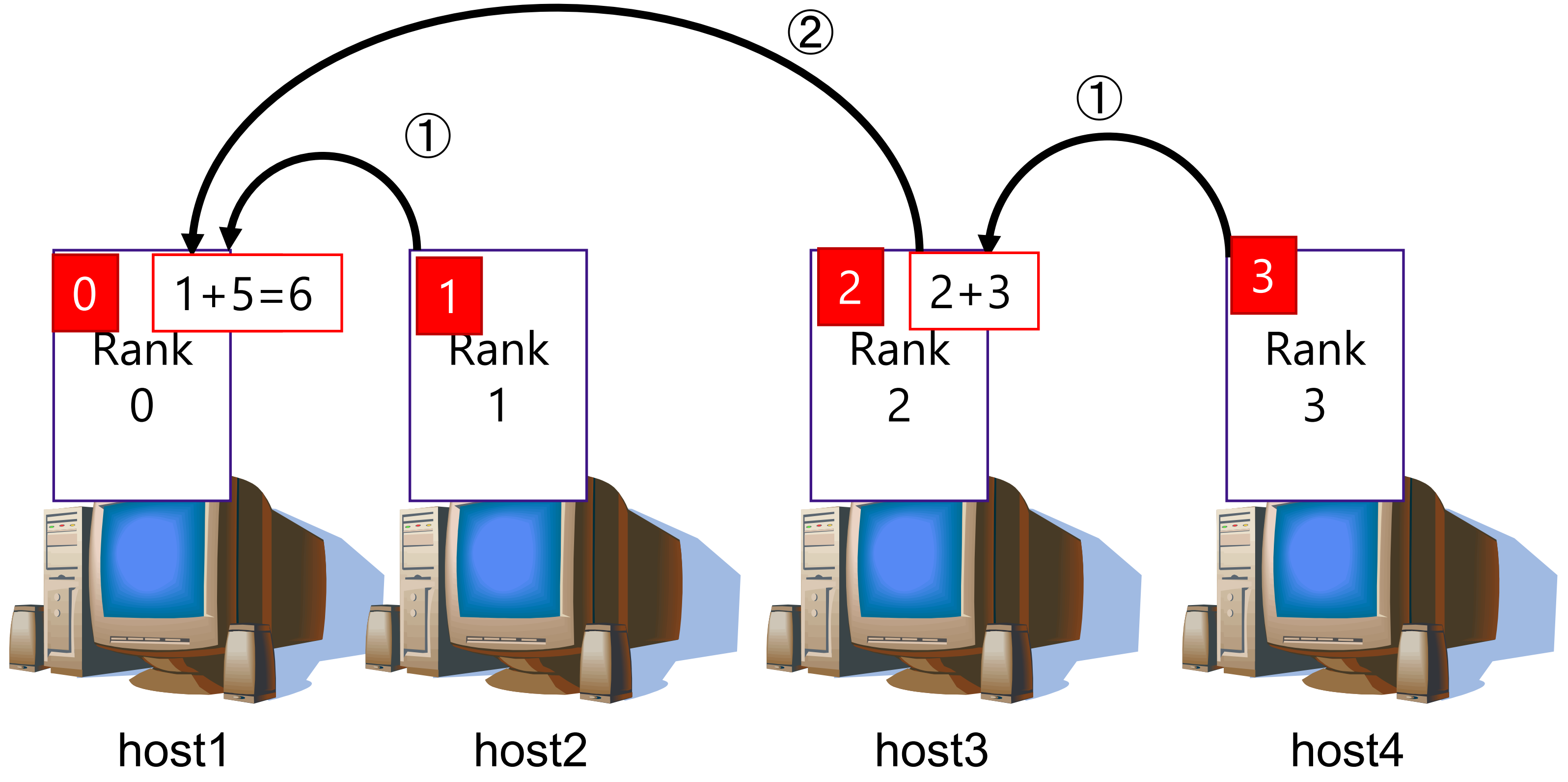
Parallel Reduction



Parallel Reduction (cont'd)

Optimized communication implementation may be able to reduce the communication cost.

2 steps

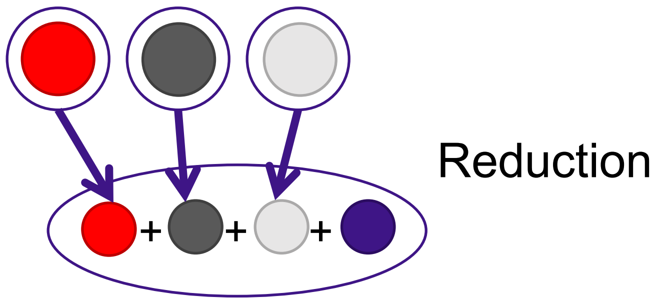


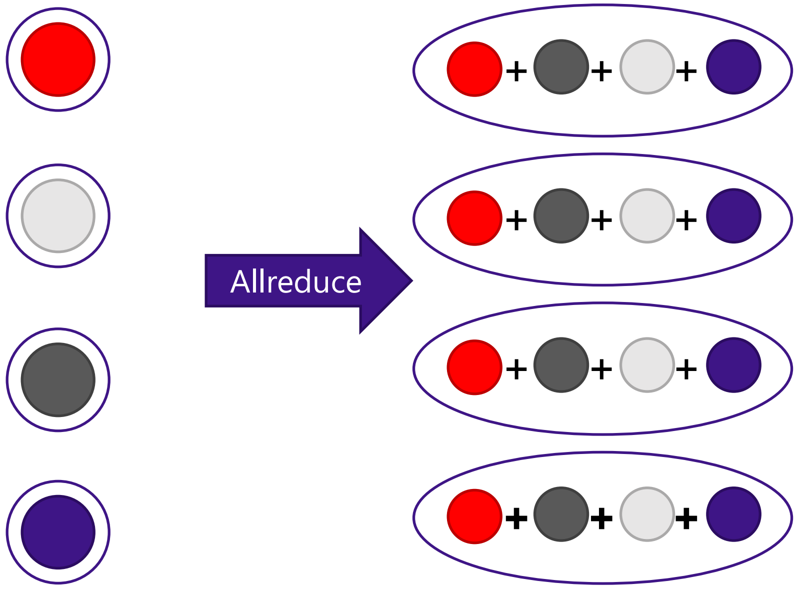
Other Comm. Functions

- **MPI_Allreduce**: allreduce
- **MPI_Gather**: gather
- **MPI_Allgather**: allgather
- **MPI_Scatter**: scatter
- **MPI_Bcast**: broadcast
- **MPI_Send**: send data (blocking)
- **MPI_Recv**: receive data (blocking)
- **MPI_Isend**: send data(non-blocking)
- **MPI_Irecv**: receive data (non-blocking)
- **MPI_Alltoall**: all-to-all communication

Reduce and Allreduce

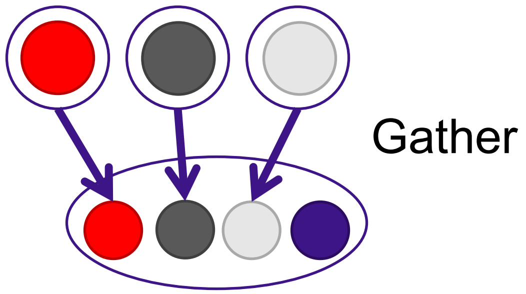
- **Reduce operation:**
  - **A single process** finally has the result of reduction op.
- **All-reduce operation:**
  - **Every process** finally has the result of reduction op.

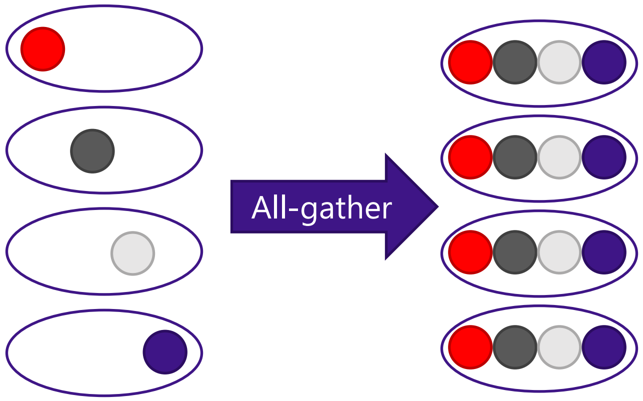




Gather and Allgather

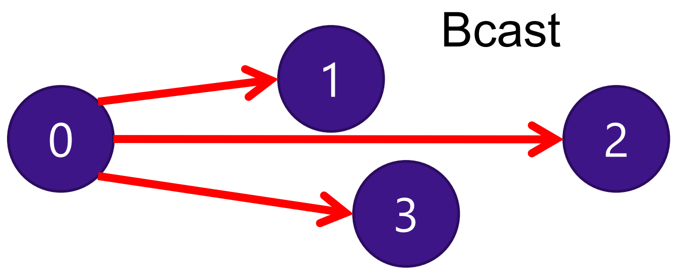
- **Gather operation:**
  - Global communication for **a single process** to collect data items distributed among others.
- **All-gather operation:**
  - Global communication for **every process** to collect data items distributed among others.

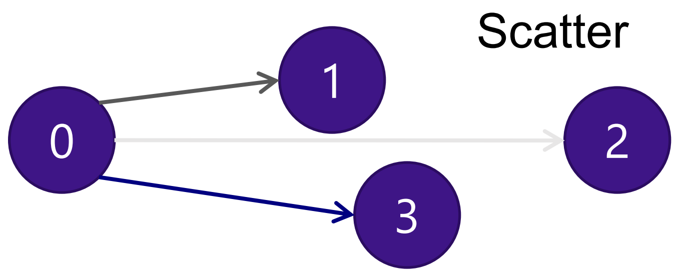




Scatter and Bcast

- **Scatter operation:**
  - Global communication like a **gather operation in reverse**.
  - One MPI process sends **different data** to each of the others.
- **Broadcast operation:**
  - One MPI process sends **the same data** to the others.





Sample Code ver. 3

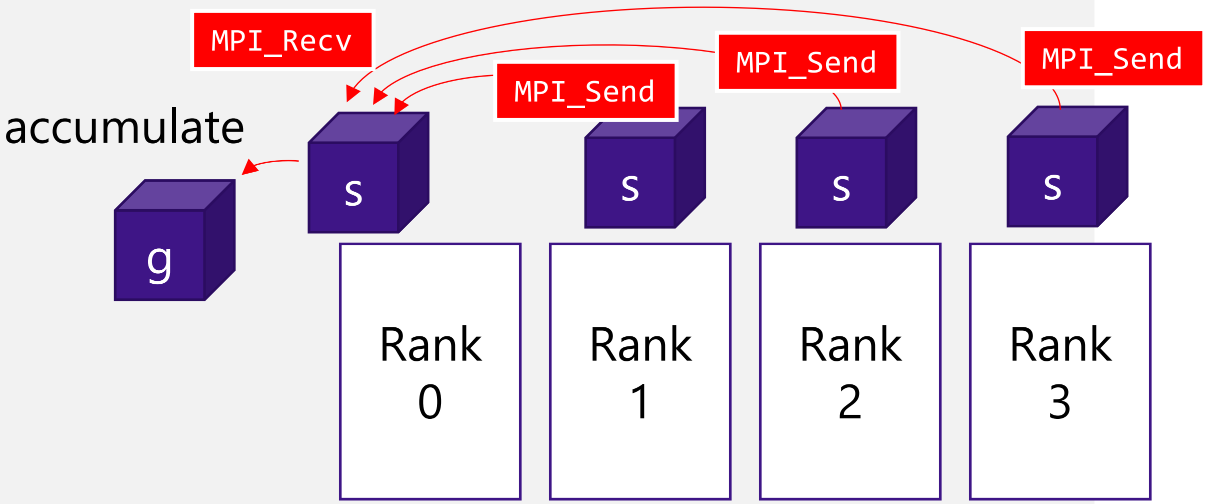
```
#include <mpi.h>     /* MPI functions */
#include <stdio.h>   /* fprintf       */

int main(int argc, char* argv[])
{
  int i, id, p, b, e, s=0, n=10000;
  int g;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
  b = (n/p)*id;
  e = (n/p)*(id+1);

  for(i=b;i<e;i++)
      s += i;
  if(id==0) { /* MPI process 0  */
    g = s;
    for(i=1;i<p;i++) {  /* receiving values from the others */
      MPI_Recv( &s, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &status);
      g += s;
    }
    printf(”The sum is %d.¥n”, g);
  }
  else { /* The other processes */
      /* sending data to MPI process 0 */
      MPI_Send( &s, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
  }
  MPI_Finalize();
  return 0;
}
```

The rank of each process is sent one by one.

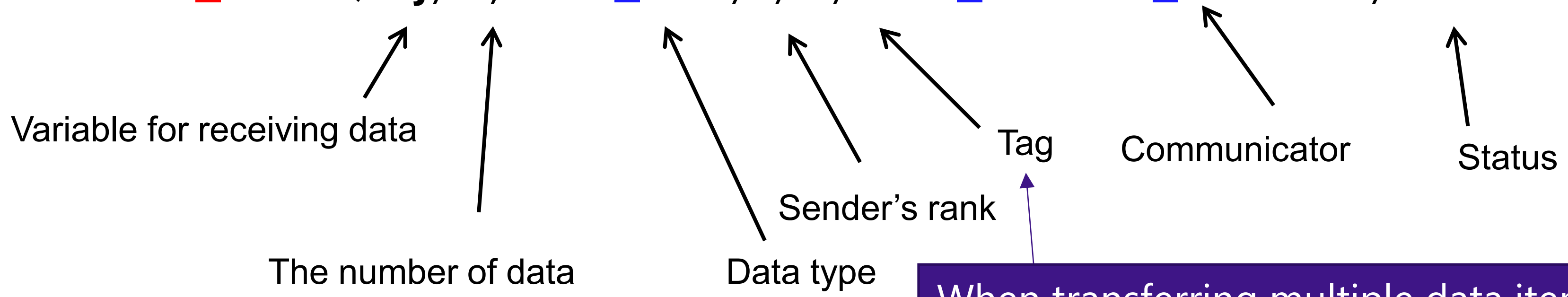


P2P Communication

MPI_Send and MPI_Recv

- The most basic MPI functions for peer-to-peer comm.
  - These functions return after the communication is completed
  - **MPI_Send**: Sending data to another MPI process
  - **MPI_Recv**: Receiving data from another MPI process
- Blocking communication
  - These functions return after the communication is completed
  - **MPI_Isend** and **MPI_Irecv** are their non-blocking version.

```
MPI_Recv( &j, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &status);
```



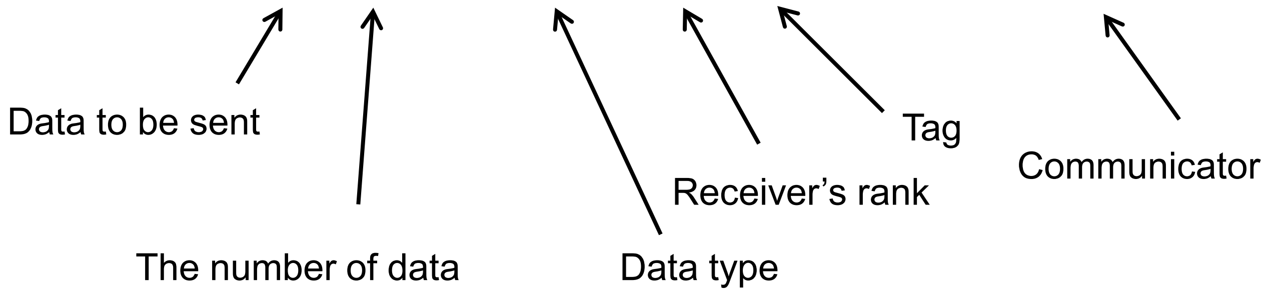
When transferring multiple data items at once, a different tag is used to identify each item.

P2P Communication

MPI_Send and MPI_Recv

- The most basic MPI functions for peer-to-peer comm.
    - **MPI_Send**: Sending data to another MPI process
    - **MPI_Recv**: Receiving data from another MPI process
- Blocking communication
    - These functions return after the communication is completed
    - **MPI_Isend** and **MPI_Irecv** are their non-blocking version.

```
MPI_Send( &j, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
```



Deadlocks

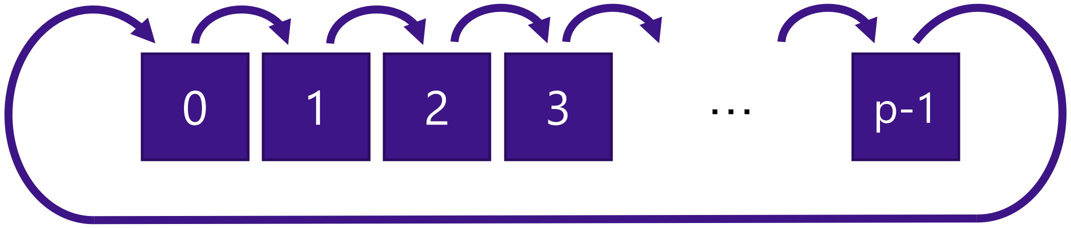
```
#include <mpi.h>    /* MPI functions */
#include <stdio.h>  /* printf        */

int main(int argc, char* argv[])
{
  int i, j, id, p, src,dst;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Comm_size(MPI_COMM_WORLD, &p);

  dst = (id+1)%p;
  src = id-1<0?p-1:id-1;
  MPI_Send( &id, 1, MPI_INT, dst, 0, MPI_COMM_WORLD);
  /* never reach this point */
  MPI_Recv( &id, 1, MPI_INT, src, 0, MPI_COMM_WORLD, &status);

  MPI_Finalize();
  return 0;
}
```



Every process is sending data, and waiting (blocked) until the receiver receives it.
= every process is waiting here.
= no process can receive the data.

Deadlock

Synchronous / Asynchronous

■ Synchronous communication

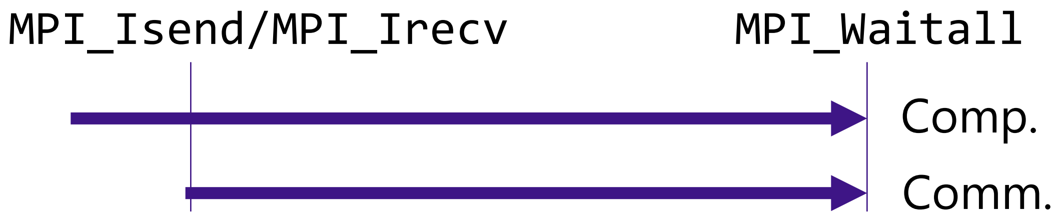
`MPI_Send & MPI_Recv`

- The task will wait until the data are sent/received.
  (= the task is **blocked**.)

■ Asynchronous communications

`MPI_Isend & MPI_Irecv`

- The task is never blocked by sending/receiving data.
  (= an asynchronous or non-blocking operation)
  - A task can initiate a data communication operation and then do another thing until the operation is completed.
  - MPI_Waitall is a function to explicitly block the task until the data communication is finished.



Overlapping computation and communication!

MPI_Isend & MPI_Irecv

```
MPI_Request req[2];
MPI_Status stat[2];

// Sending a data item is initiated but the task is not blocked
MPI_Isend(&send_data[0], send_data_count, MPI_DOUBLE,
           dest, 0, MPI_COMM_WORLD, &req[0]);

// Receiving a data item is initiated but the task is not blocked
MPI_Irecv(&recv_data[0], recv_data_count, MPI_DOUBLE,
           src, 0,MPI_COMM_WORLD, &req[1]);

// The task is blocked here until the operations associated with
// req[0] and req[1] are completed.
// Error codes of req[0] and req[1] are in stat[0] and stat[1]
MPI_Waitall(2, req, stat);
```

Benchmarking Performance

- How to measure the elapsed time for parallel processing?
  - **MPI_Wtime**
    - If **`MPI_Wtime`** is called twice, the different between their return values indicates the elapsed time [sec] between the calls.

```
double etime;
etime = -MPI_Wtime();
 /* some parallel task */
etime += MPI_Wtime();
printf(”elapsed time: %lf [sec]¥n”, etime);
```

The time measurement could be inaccurate!!

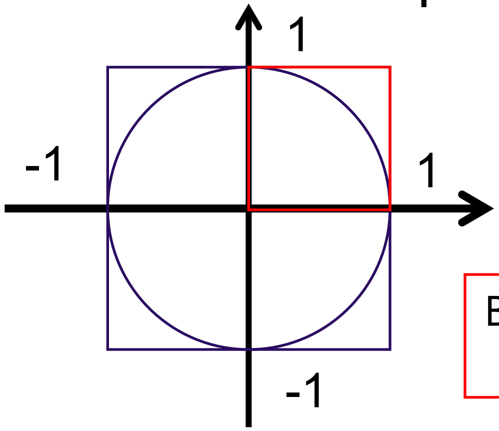
Barrier Synchronization

- How to ensure every process calls MPI_Wtime at the same time?
  - **Barrier synchronization**
    - No process can proceed beyond it until all processes have reached it.
  - `MPI_Barrier`
    - Barrier synchronization of one communicator group.
    - Time-consuming, so prevent unnecessary synchronizations.

```
double etime;
MPI_Barrier(MPI_COMM_WORLD);
etime = -MPI_Wtime();
 /* some parallel task */
MPI_Barrier(MPI_COMM_WORLD);
etime += MPI_Wtime();
printf(”elapsed time: %lf [sec]¥n”, etime);
```

Exercise :
Monte Carlo π Calculation

- The area of a circle is π=3.1415... if the radius is 1.
- Suppose that a lot of points are randomly generated within the region of 0<x<1 and 0<y<1.
  - The probability that a point is in a circle is π/4.



Numerical integration by Monte Carlo approach

By increasing the number of random points
  (The number of points in the circle)/(the total number of points) → π/4.

Sequential Code

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

int main(int argc,char* argv[])
{
  int N = 1000;
  int i,total = 0;
  double x,y;

  srand(time(NULL)); /* initialization */
  for(i=0;i<N;i++) {
    /* two random numbers 0<x,y<1 */
    x = (double)rand()/RAND_MAX;
    y = (double)rand()/RAND_MAX;
    if( x*x + y*y < 1 ) {
      total = total+1;
    }
  }
 printf( "pi=%lf¥n",4.0*total/N);
}
```

Statistically speaking, the accuracy will improve by increasing N.

Hint:
Divide the loop into some pieces, and assign each piece to one MPI process.
Use reduction to calculate the total from subtotals.

Parallelize this code and measure the performance!

Summary

- **Introduction to MPI Programming**
  - A minimal MPI program
  - Performing common communication patterns with collective MPI calls
  - Peer-to-peer communication for data exchange
  - Blocking and non-blocking operations
  - Benchmarking
- **T/C model naturally fits MPI paradigm.**
  - Let’s try more complicated parallel program design!