Practical Parallel Computing (実践的並列コンピューティング)

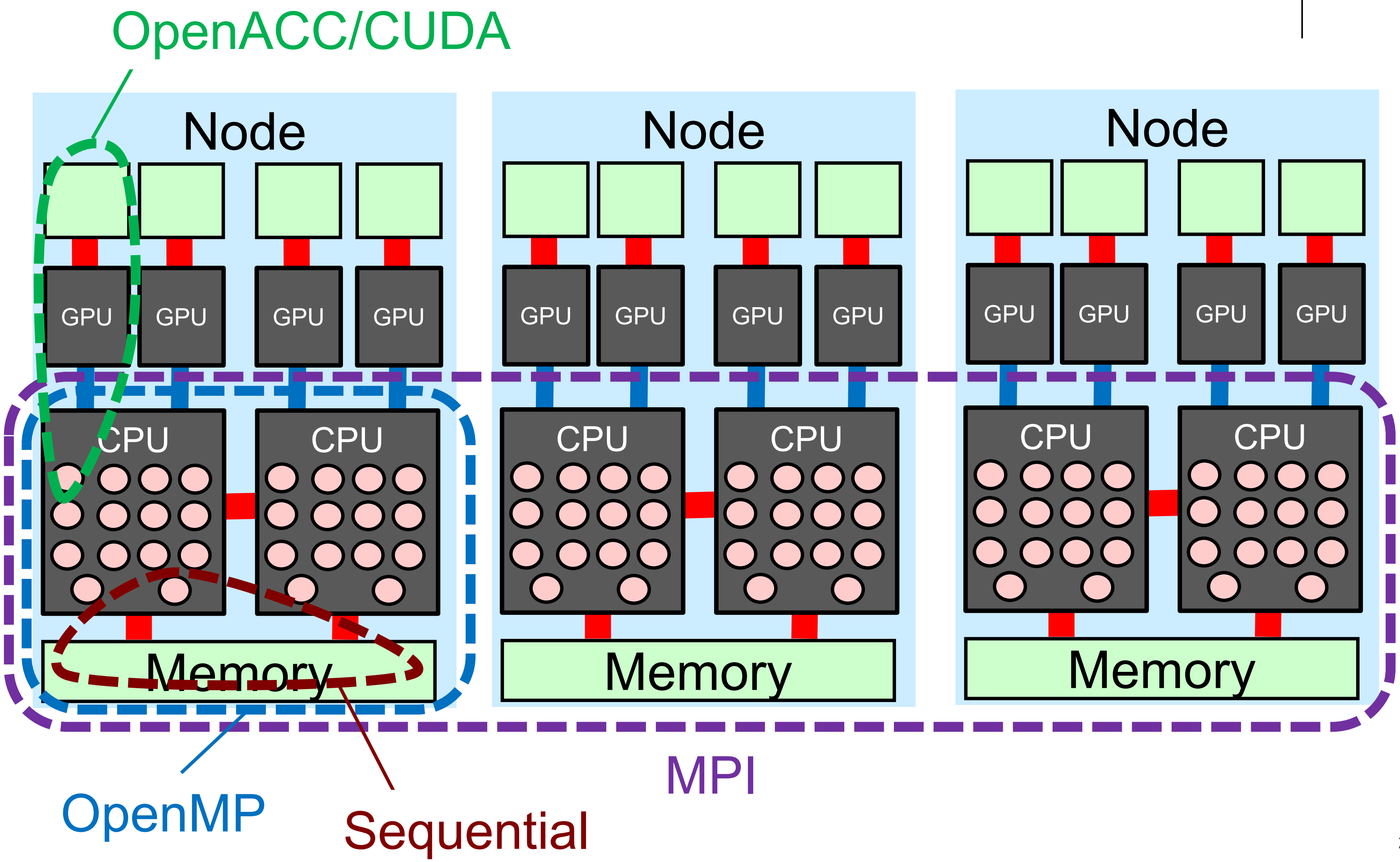
Part3: MPI (1)
June 11, 2020

Toshio Endo
School of Computing & GSIC
endo@is.titech.ac.jp

Overview of This Course

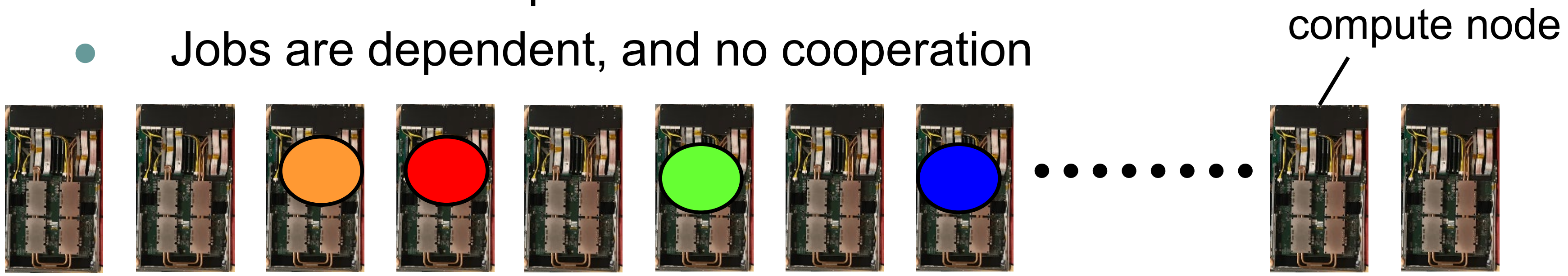
- Part 0: Introduction
  - 2 classes
- Part 1: OpenMP for shared memory programming
  - 4 classes
- Part 2: GPU programming
  - 4 classes ← We are here (1/4)
  - OpenACC (1.5 classes) and CUDA (2.5 classes)
- Part 3: MPI for distributed memory programming
  - 3 classes ← We are here (1/3)

Parallel Programming Methods on TSUBAME

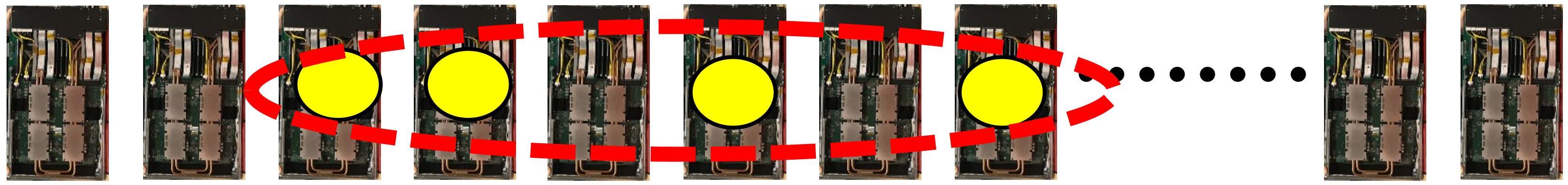


How We Can Use Many Nodes

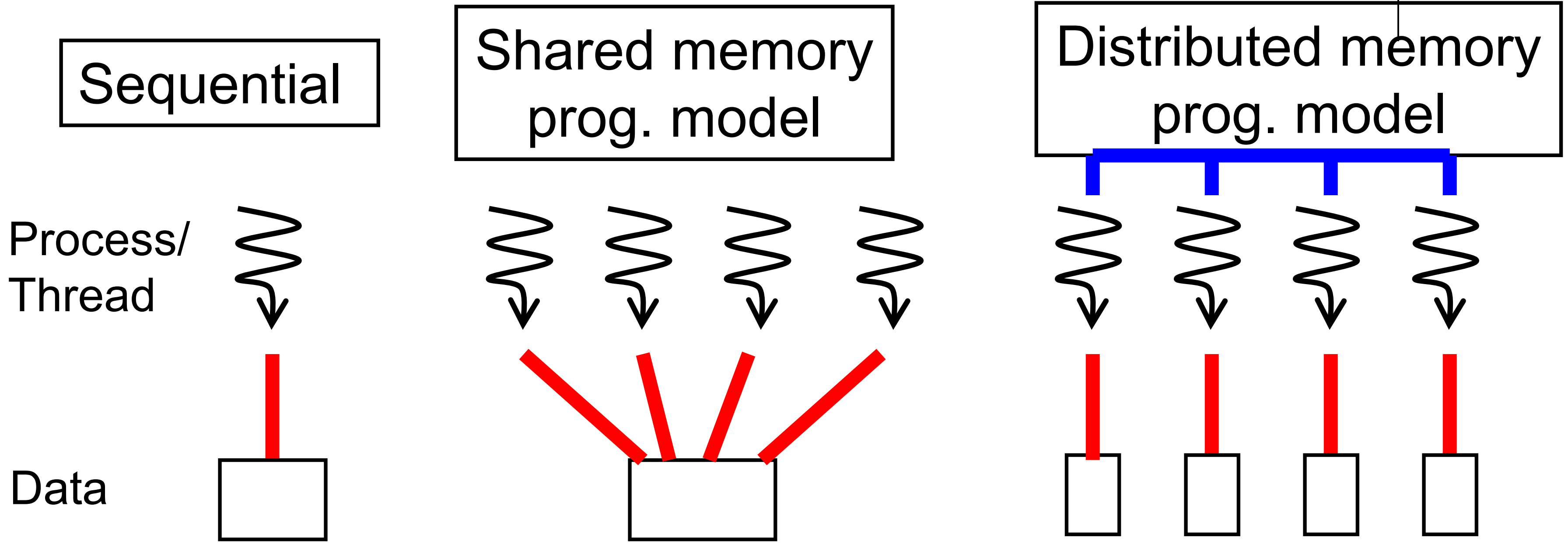
1. Submit several jobs into job scheduler
   - cf) Program executions with different parameters → Parameter Sweep
   - Jobs are dependent, and no cooperation



2. Use distributed memory programming → A single job can use multiple nodes
   - Socket programming, Hadoop, Spark…
   - And MPI



Classification of Parallel Programming Models



Programming without parallelsim

Threads have access to shared data

- OpenMP
- pthread
- Java thread…

Need communication among processes

- MPI
- socket
- Hadoop, Spark…

MPI (message-passing interface)

- Parallel programming interface based on distributed memory model
- Used by C, C++, Fortran programs
  - Programs call MPI library functions, for message passing etc.
- There are several MPI libraries
  - OpenMPI (default) ← OpenMPI ≠ OpenMP ☹
  - Intel MPI, SGI MPE, MVAPICH, MPICH…

Differences from OpenMP

In MPI,

- An execution consists of multiple processes (not threads)
  - We can use multiple nodes ☺
  - The number of running processes is basically constant
- No variables are shared. Instead message passing is used
  - Data distribution has to be programmed
- No smart syntaxes such as “omp for” or “omp task” ☹
  - Task distribution has to be programmed ☹

First MPI Sample

- /gs/hs1/tga-ppcomp/20/hello-mpi

```
[make sure that you are at a interactive node (r7i7nX) ]
module load cuda openmpi    [Do once after login]
cd ~/t3workspace            [Example in web-only route]
cp -r /gs/hs1/tga-ppcomp/20/hello-mpi  .
cd hello-mpi
make
[An executable file “hello” is created]
mpiexec -n 7 ./hello
```

Compiling and Executing MPI Programs

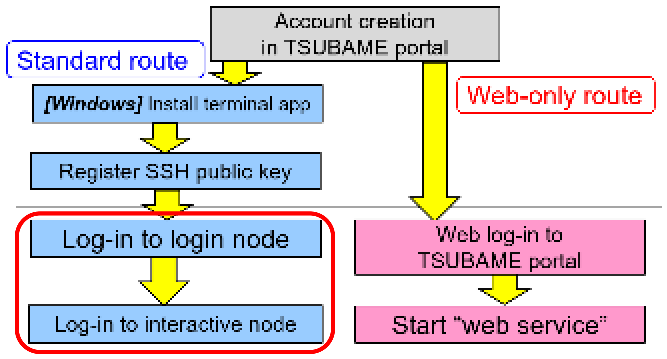
Case of OpenMPI library on TSUBAME3.0

- To compile
  Required for module dependency☹
  - module load cuda openmpi, and then use mpicc
  - For sample programs, "make" command works

- To execute
  Number of processes
  - mpiexec -n 7 ./hello
  - ./hello 🡐 only 1 process is used

↑ These methods uses 1 (current) node. For multi-nodes, we need "job submission

Notes on "Standard route"



Account creation in TSUBAME portal
Standard route
[Windows] Install terminal app
Register SSH public key
Log-in to login node
Log-in to interactive node
Web-only route
Web log-in to TSUBAME portal
Start "web service"

- On an interactive node via "standard route", qsub/qstat commands are not found
- Please use
  - qrsh -q interactive -l h_rt=2:00:00 -v PATH

  instead of qrsh -q interactive -l h_rt=2:00:00
  - By doing that, PATH environment variable on login node is passed to interactive node

Submit an MPI Job (case of OpenMPI)

- We are going to execute it with 4 processes × 2 nodes = 8 processes

(1) Make a script file: job.sh

```
#!/bin/sh
#$ -cwd
#$ -l q_core=2
#$ -l h_rt=00:10:00

. /etc/profile.d/modules.sh
module load cuda openmpi

mpiexec –n 8 –npernode 4 ./hello
```

4core node x 2

Module preparation

Number of processes

Number of processes per node

Program name (and option)

(2) Submit the job with "qsub"

qsub job.sh

(≦0:10:00, ≦2node for free)

qsub –g tga-ppcomp job.sh

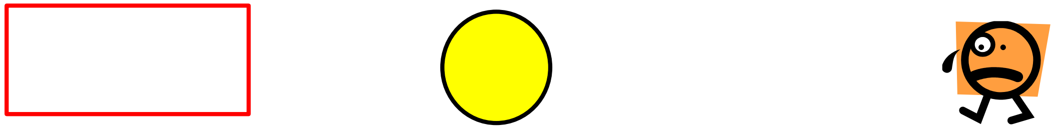
(if you use the group)

Notes in This Lecture

- Usually, avoid consumption of TSUBAME points
- 通常は無料利用の範囲にとどめてください
  - h_rt <= 0:10:00

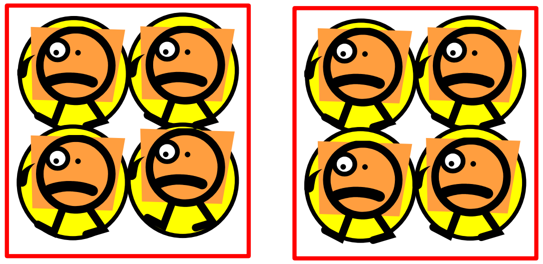
- If necessary for reports, you can use up to 72,000 points in total per student
- 本講義のレポートの作成に必要な場合、一人あたり合計で72,000ポイントまで利用を認めます
  - f_node x 1node x 20 hours

- Please check point consumption on TSUBAME portal
- The TSUBAME group name is tga-ppcomp



Nodes, Cores, MPI Processes

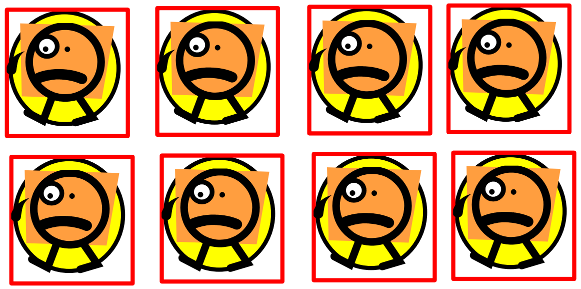
```
          :
#$ -l q_core=2
          :
mpirun –n 8 –npernode 4
  …
```



2 (virtual) nodes are prepared
Each node has 4 cores (q_core)

4 processes are created per node. Totally 8 are created
→ 2 nodes are used

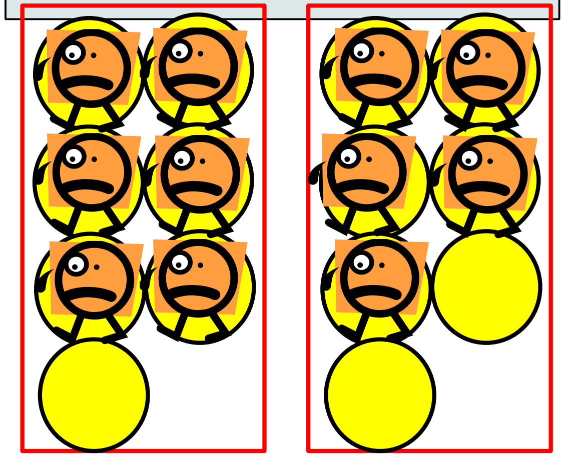
```
          :
#$ -l s_core=8
          :
mpirun –n 8 –npernode 1
  …
```



8 (virtual) nodes are prepared
Each node has 1 cores (s_core)

1 processes are created per node. Totally 8 are created
→ 8 nodes are used

```
          :
#$ -l q_node=2
          :
mpirun –n 11 –npernode 6
  …
```



2 (virtual) nodes are prepared
Each node has 7 cores (q_node)

6 processes are created per node. Totally 11 are created
→ 2 nodes are used
(There are idle cores)

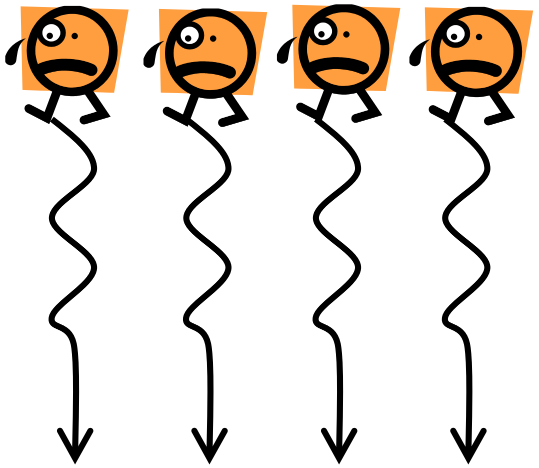
An MPI Program Looks Like

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

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

   (Computation/communication)

   MPI_Finalize();
}
```

← Initialize MPI

← Finalize MPI

If number of processes=4

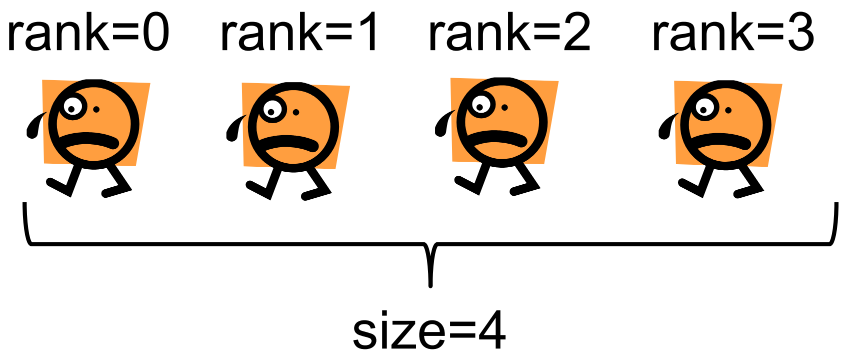


ID of Each MPI Process

- Each process has its ID (0, 1, 2…), called rank
  - `MPI_Comm_rank(MPI_COMM_WORLD, &rank);`

  → Get its rank
  - `MPI_Comm_size(MPI_COMM_WORLD, &size);`

  → Get the number of total processes
  - 0 ≦ rank < size
  - The rank is used as target of message passing



“mm” sample: Matrix Multiply

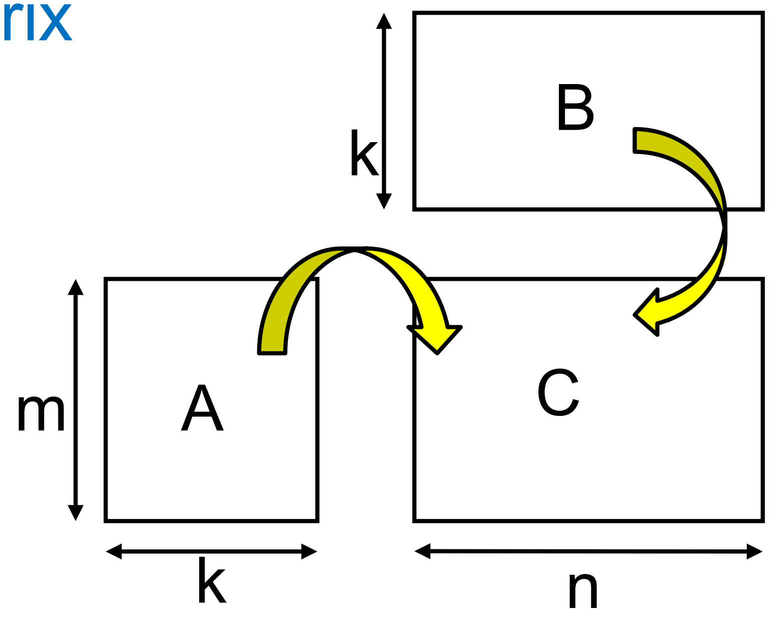
MPI version available at /gs/hs1/tga-ppcomp/20/mm-mpi/

A: a (m × k) matrix, B: a (k × n) matrix

C: a (m × n) matrix

$C \leftarrow A \times B$

- Algorithm with a triple for loop
- Supports variable matrix size.
  - Each matrix is expressed as a 1D array by *column-major* format



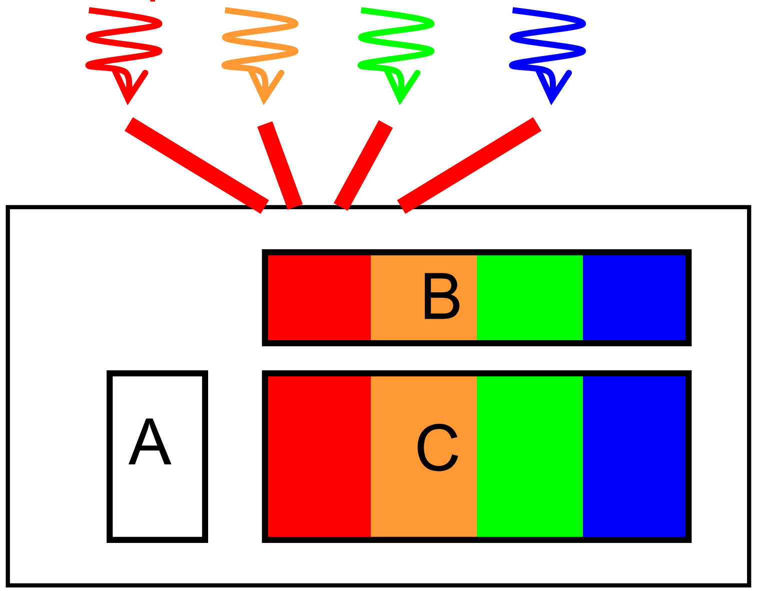
Execution:

```
mpirun -n [np] -npernode [nn] ./mm [m] [n] [k]
```

Why Distributed Programming is More Difficult (case of mm-mpi)

Shared memory with OpenMP:

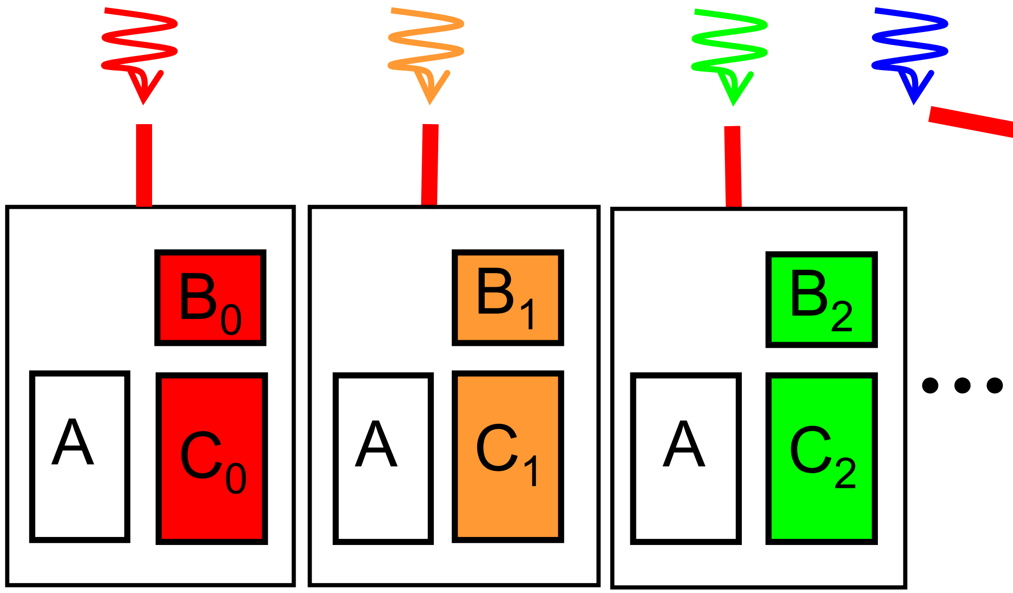
Programmers consider how computations are divided



In this case, matrix A is accessed by all threads
→ Programmers do not have to know that

Distributed memory with MPI:

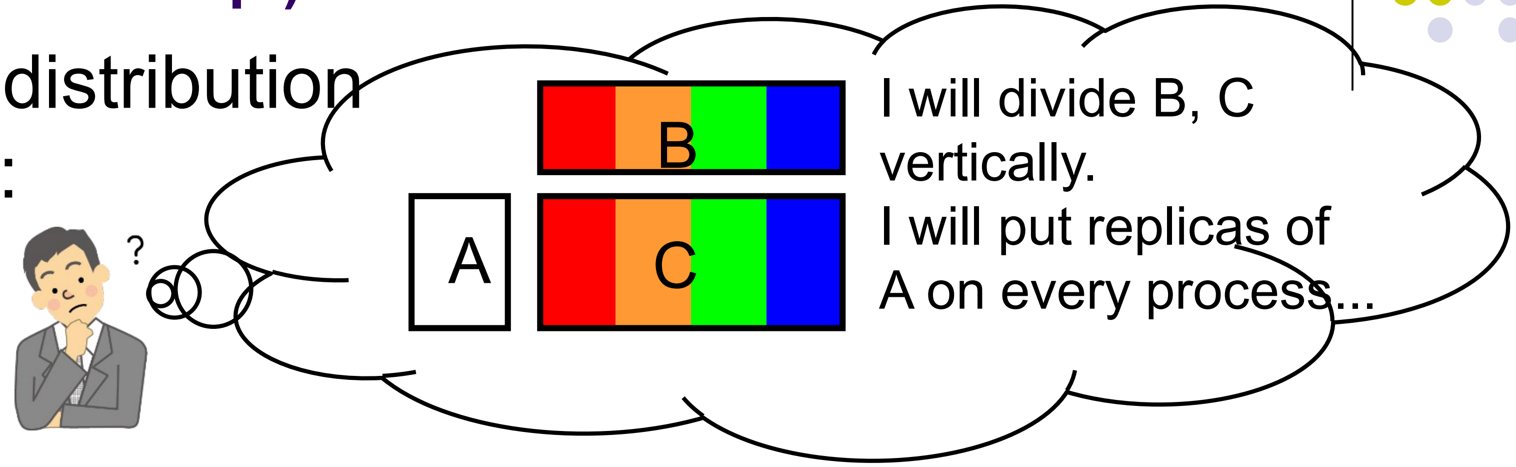
Programmers consider how data and computations are divided



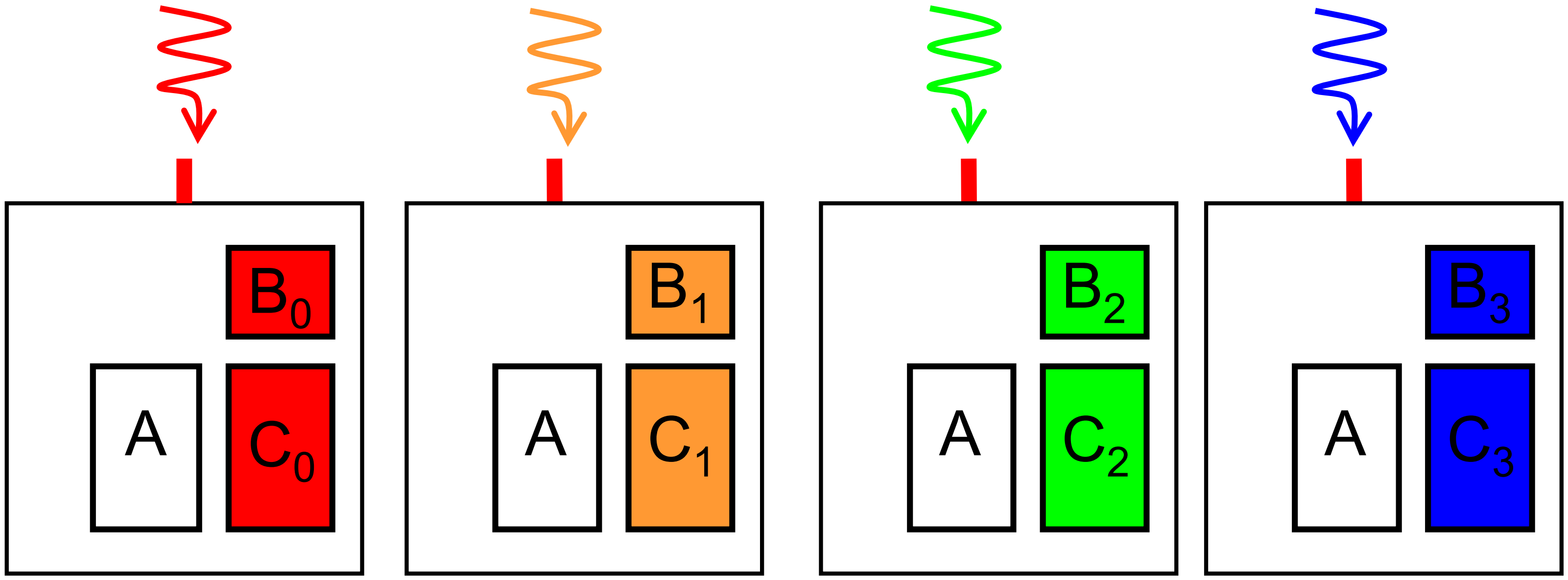
Programmers have to design which data is accessed by each process

Programming Data Distribution
(case of mm-mpi)

Design distribution method:


?
B
A
C
I will divide B, C vertically.
I will put replicas of A on every process...

Programming actual location:



Programming Actual Data Distribution

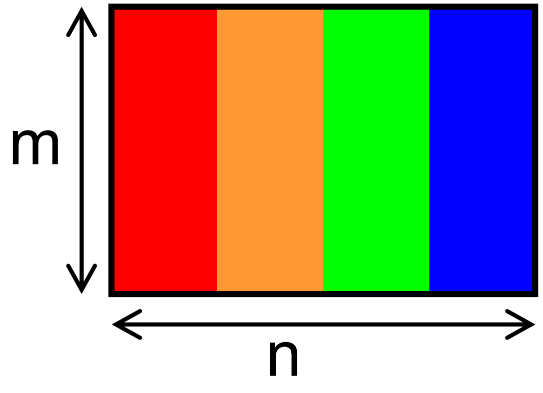
- We want to distribute a $m \times n$ matrix among p processes
  - We assume n is divisible by p
- Each process has a partial matrix of size $m \times (n/p)$
  - We need to "malloc" m*(n/p)*sizeof(data-type) size
  - We need to be aware of relation between partial matrix and entire matrix
    - (i,j) element in partial matrix owned by Process r ⇔

      (i, n/p*r + j) element in entire matrix

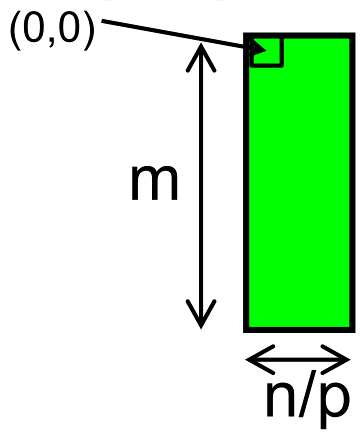
local index

global index

Entire matrix



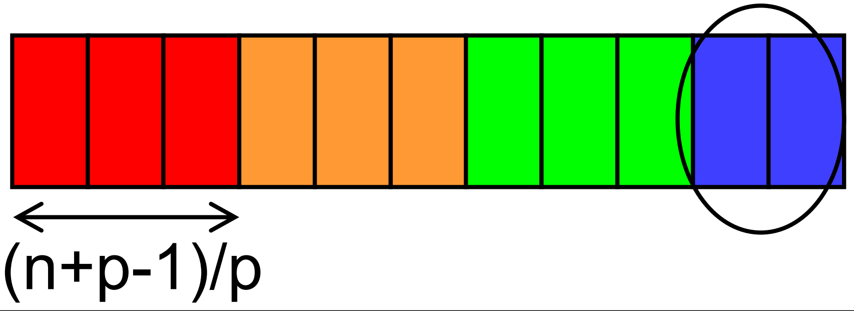
Actual matrix per process



What is Done for Indivisible Cases

- What if data size n is indivisible by p?
- We let n=11, p=4
  - How many data each process take?
  - n/p = 2 is not good (C division uses round down). Instead, we should use round up division
  - → (n+p-1)/p = 3 works well

  Note that the "final" process takes less than others

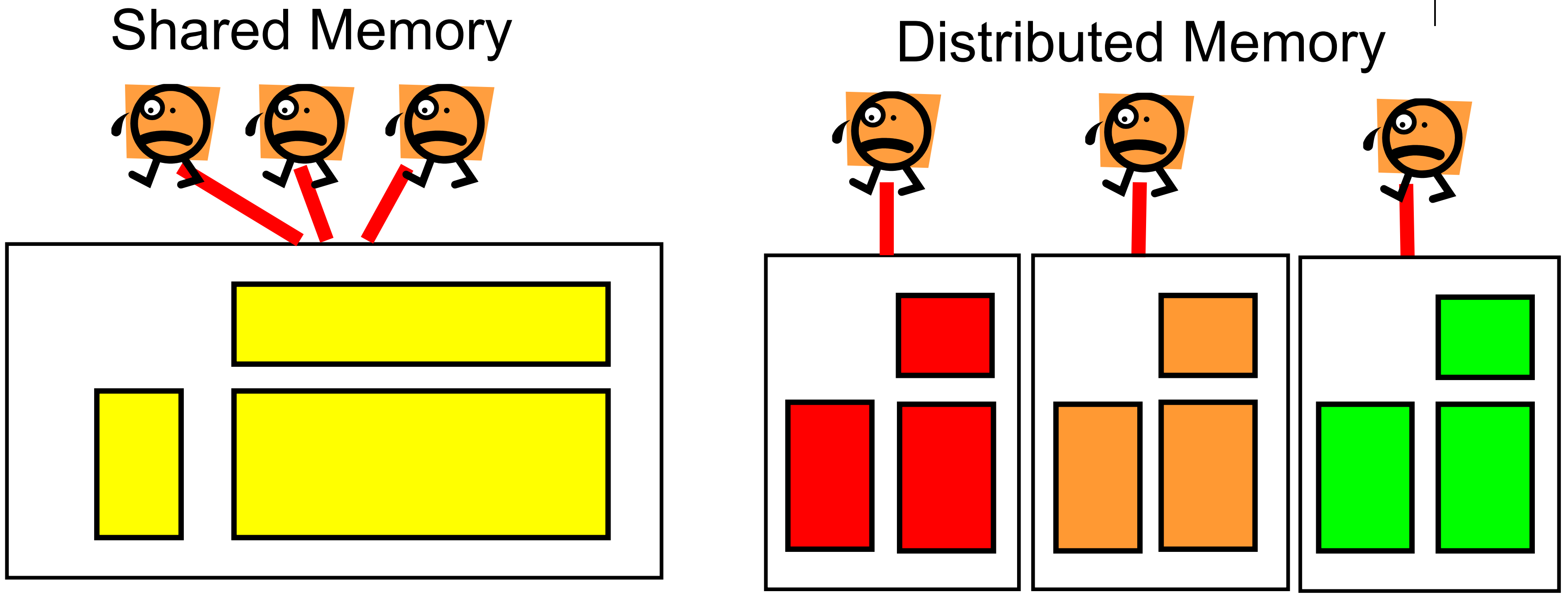


See divide_length() function in mm-mpi/mm.c
It calculates the range the process should take (first index s and last index e)

Notes in Time Measurement

- In mm-mpi, gettimeofday() is used for time measurement
- For accurate measurement, we should call MPI_Barrier(MPI_COMM_WORLD) before measurement
  - This synchronizes all processes
  - All processes need to call this

Shared Memory Model and Distributed Memory Model



- In distributed memory model, a process CANNOT read/write other processes' memory directory
- How can a process access data, computed by others?

→ Message passing (communication) is requried

Basics of Message Passing: Peer-to-peer Communication

Example at: /gs/hs1/tga-ppcomp/20/test-mpi/

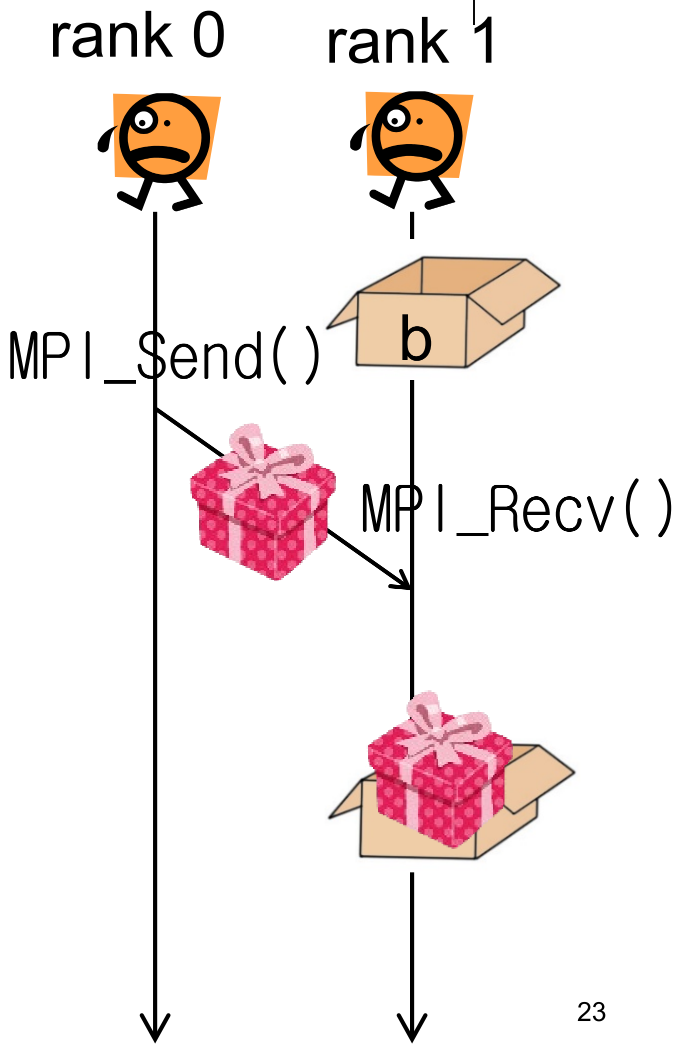
Execute: mpiexec -n 2 ./test

Rank 0 computes "int a[16]"

Rank 1 wants to see contents of a!

Rank0:
- Computes data of a
- Send data of a to rank1

Rank1:
- Prepares a memory region (b here)
- Receive data from rank0 and store it to b
- Now b has copy of a !



MPI_Send

```
MPI_Send(a, 16, MPI_INT, 1, 100, MPI_COMM_WORLD);
```

- a: Address of memory region to be sent
- 16: Number of data to be sent
- MPI_INT: Data type of each element
  - MPI_CHAR, MPI_LONG. MPI_DOUBLE, MPI_BYTE･･･
- 1: Destination process of the message
- 100: An integer tag for this message (explained later)
- MPI_COMM_WORLD: Communicator (explained later)

rank 0
source:0
dest: 1
tag:100

MPI_Recv

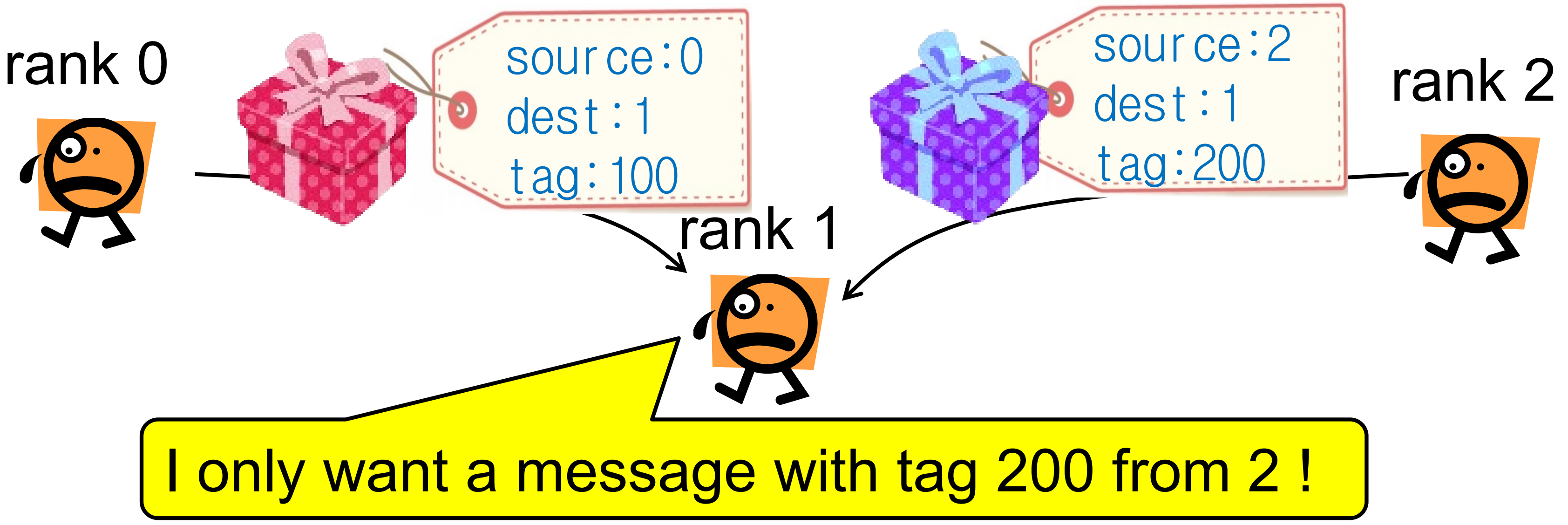
```
MPI_Status stat;
MPI_Recv(b, 16, MPI_INT, 0, 100, MPI_COMM_WORLD, &stat);
```

- b: Address of memory region to store incoming message
- 16: Number of data to be received
- MPI_INT: Data type of each element
- 0: Source process of the message
- 100: An integer tag for a message to be received
  - Should be same as one in MPI_Send
- MPI_COMM_WORLD: Communicator (explained later)
- &stat: Some information on the message is stored

Note: MPI_Recv does not return until the message arrives

Notes on MPI_Recv: Message Matching (1)

```
MPI_Recv(b, 16, MPI_INT, 2, 200, MPI_COMM_WORLD, &stat);
```



- Receiver specifies “source” and “tag” that it wants to receive

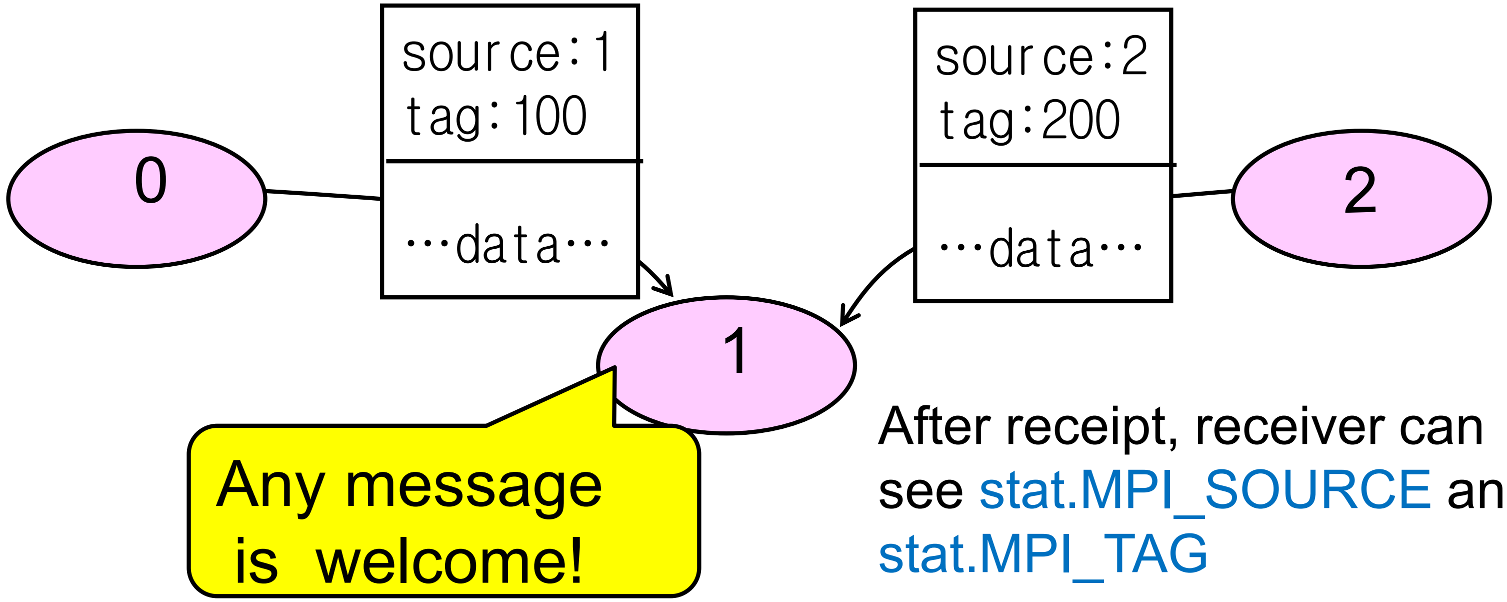
→ The message that matches the condition is delivered

- Other messages should be received by other MPI_Recv calls

Notes on MPI_Recv: Message Matching (2)

- In some algorithms, the sender may not be known beforehand
  - cf) client-server model
- For such cases, MPI_ANY_SOURCE / MPI_ANY_TAG can be used

```
MPI_Status stat;
MPI_Recv(b, 16, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
   MPI_COMM_WORLD, &stat);
```



After receipt, receiver can see stat.MPI_SOURCE and stat.MPI_TAG

Assignments in this Course

- There is homework for each part. Submissions of reports for 2 parts are required

Part 1 OpenMP	[O1] diffusion [O2] sort [O3] free	*Select 1 problem*
Part 2 GPU	[G1] diffusion [G2] mm [G3] free	*Select 1 problem*
Part 3 MPI	[M1] diffusion [M2] mm [M3] free	*Select 1 problem*

Select 2 parts

Assignments in MPI Part (1)

Choose one of [M1]—[M3], and submit a report

Due date: 11AM, June 29 (Monday)

[M1] Parallelize "diffusion" sample program by MPI.

- Do not forget to change Makefile and job.sh appropriately
- Use deadlock-free communication
  - see neicomm_safe() in neicomm-mpi sample

Optional:

- To make array sizes (NX, NY) variable parameters
- To consider the case with NY is indivisible by p
  - see divide_length() in mm_mpi sample
- To improve performance further. Blocking, 2D division, etc

Assignments in MPI Part(2)

[M2] Improve “mm-mpi” sample in order to reduce memory consumption

Optional:

- To consider indivisible cases
- To try advanced algorithms, such as SUMMA
  - the paper “*SUMMA: Scalable Universal Matrix Multiplication Algorithm*” by Van de Geijn
  - http://www.netlib.org/lapack/lawnspdf/lawn96.pdf

Assignments in MPI Part (3)

[M3] (Freestyle) Parallelize *any* program by MPI.

- cf) A problem related to your research
- More challenging one for parallelization is better
  - cf) Partial computations have dependency with each other

Notes in Report Submission (1)

- Submit the followings via OCW-i

  (1) A report document

  - PDF, MS-Word or text file
  - 2 pages or more
  - in English or Japanese (日本語もok)

  (2) Source code files of your program

  - Try “zip” to submit multiple files

Notes in Report Submission (2)

The report document should include:

- Which problem you have chosen
- How you parallelized
  - It is even better if you mention efforts for high performance or new functions
- Performance evaluation on TSUBAME
  - With varying number of processes

    Either is ok
    - Using up to 7 processes on an interactive node
    - Using qsub (≦2nodes)
    - Using qsub (>2nodes)
  - With varying problem sizes
  - Discussion with your findings
  - Other machines than TSUBAME are ok, if available

Next Class

- MPI (2)
  - How to parallelize diffusion sample with MPI
- Class Evaluation (授業アンケート)