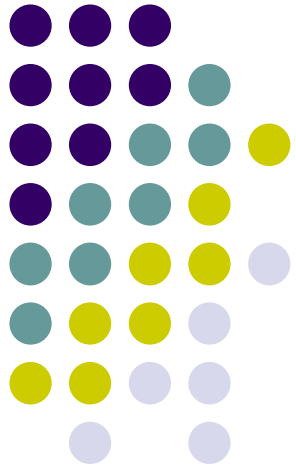
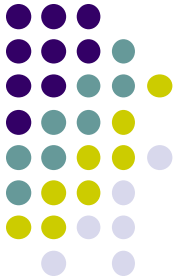
Practical Parallel Computing (実践的並列コンピューティング)

2025 Class No.12
[MPI Part] (1)
Introduction to MPI



Toshio Endo
endo@scrc.iir.isct.ac.jp

Overview of This Course



- Introduction Part
  - 2 classes
- OpenMP (OMP) Part
  - 4 classes
  - Report (required)
- OpenACC (ACC) Part
  - 2 classes
  - Report (required)
- CUDA Part
  - 3 classes
  - Report (elective)
- MPI Part
  - 3 classes
  - Report (elective)

← We are here (1/3)

Note:
Modification to ppcomp-ex github

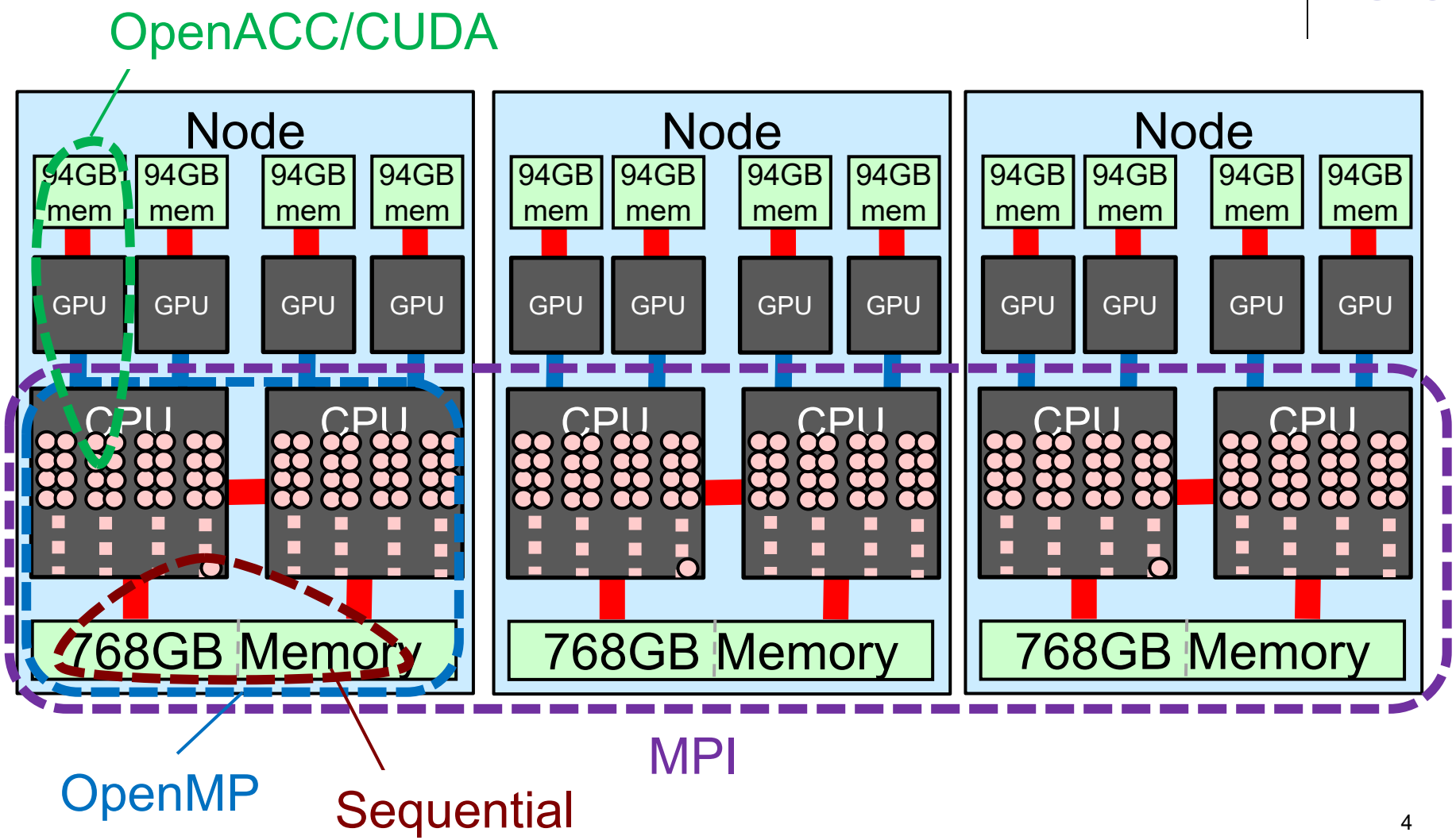
An MPI sample (mm-comm) is added related to [M3]

Please update your ppcomp-ex directory

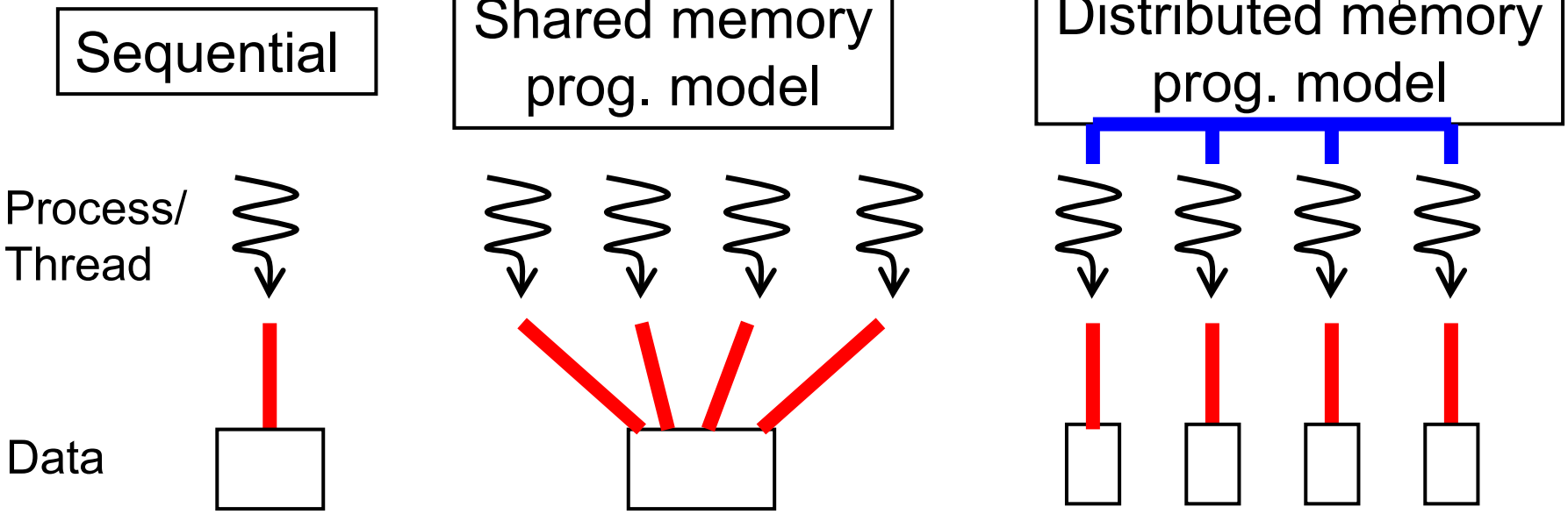
```
cd ppcomp-ex   // your ppcomp-ex directory
git pull
// ➔ mpi/mm-comm/ is added
```

Thanks for cooperation (again!)

Parallel Programming Methods on TSUBAME



Classification of Parallel Programming Models

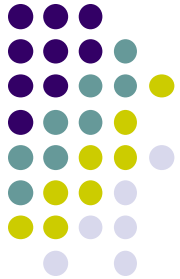


Programming without parallelsim

Threads have access to shared data
- OpenMP
- pthread
- Java thread…

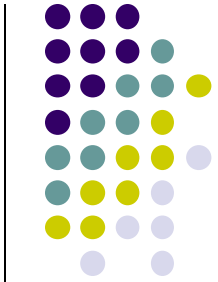
Need communication among processes
- MPI
- socket
- Hadoop, Spark…

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
  - Intel MPI
  - OpenMPI ← OpenMPI ≠ OpenMP ☹
  - ...

Compiling and Executing MPI Programs



Case of Intel MPI library on TSUBAME4.0

- To compile
  - module load intel-mpi, and then use mpicc
  - For sample programs, “make” command works
- To execute
  - mpiexec -n 4 ./hello

Number of processes

↑ These methods uses 1 (current) node.

First MPI Sample

- ppcomp-ex/mpi/hello

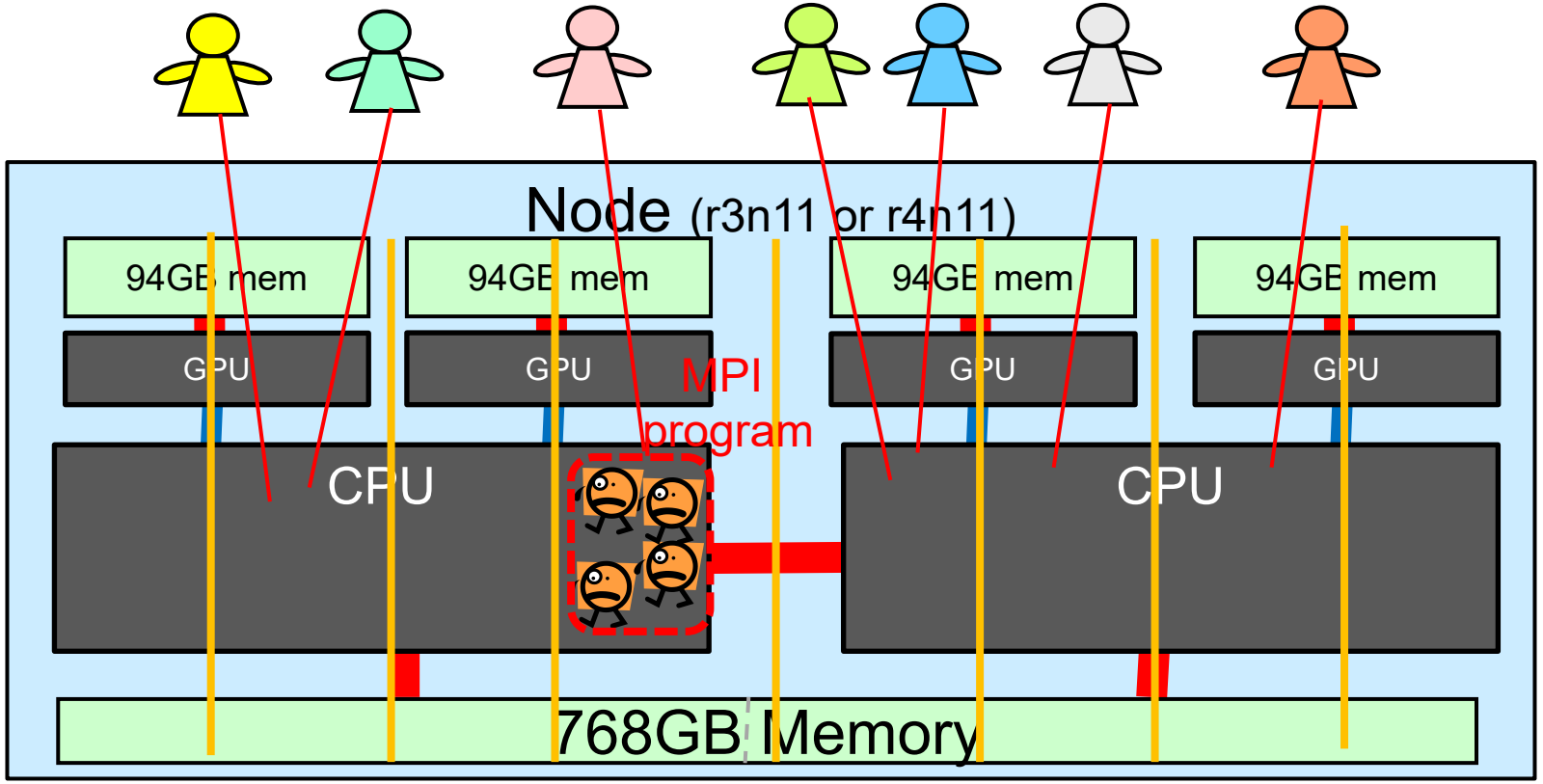
```
[make sure that you are at a interactive node (rXn11) ]
module load intel-mpi    [Do once after login]
[please go to your ppcomp-ex directory]
cd mpi/hello
make
[An executable file "hello" is created]
mpiexec -n 4 ./hello
```

Number of processes

Name of program
(using options are ok)

※ On TSUBAME interactive node,
using only ≦ 4 processes is allowd ☹

On TSUBAME Interactive Node



In this lecture, we are using an interactive node

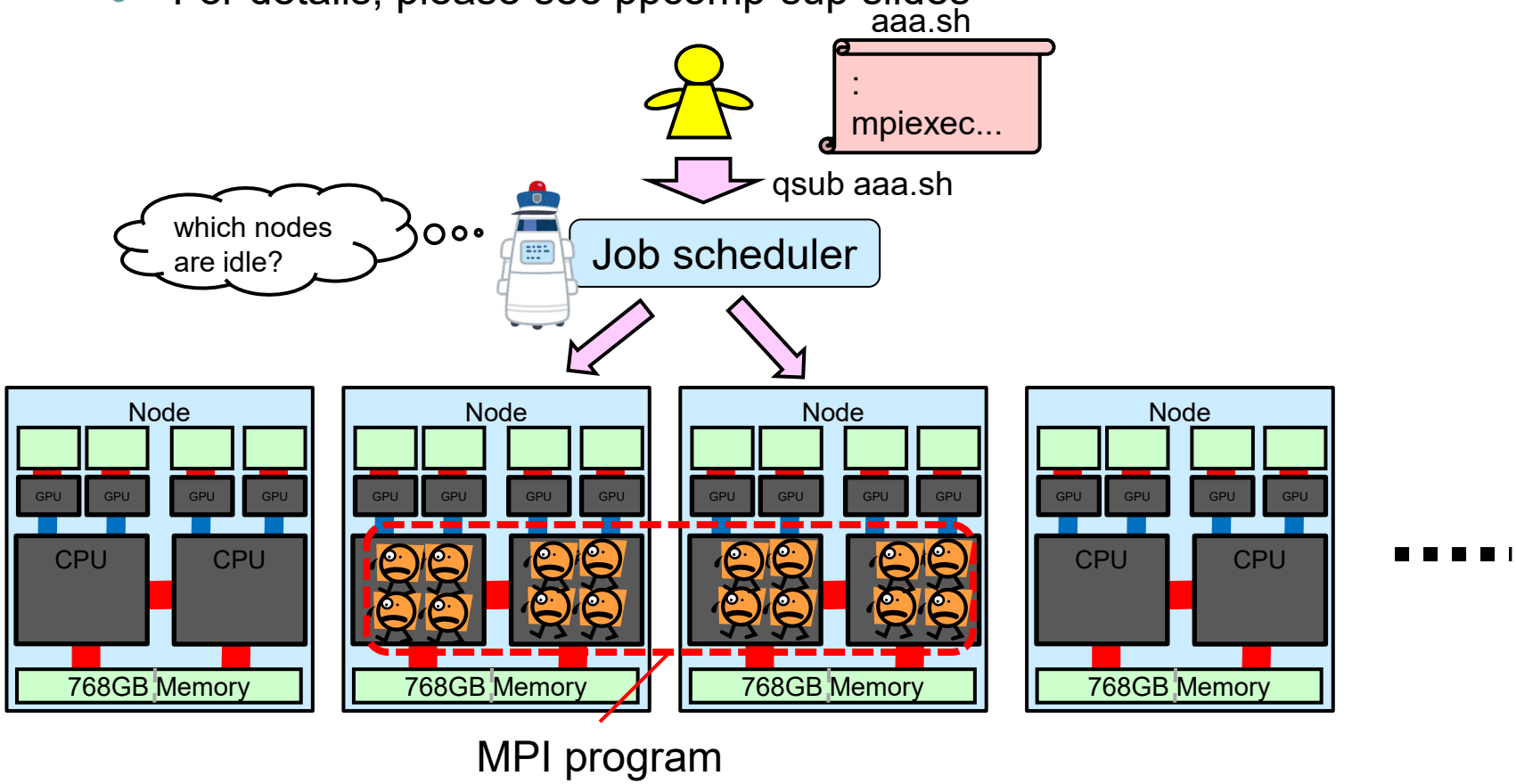
- 1/8 node has 24 CPU cores

Multiple processes are invoked on a interactive node

If you want to use multiple nodes, use the job scheduler (ppcomp-sub slides)

Using Multiple Nodes is Possible

- MPI programs can run on multiple nodes
  - On TSUBAME, it is possible with the job scheduler
  - For details, please see ppcomp-sup slides



An MPI Program Looks Like

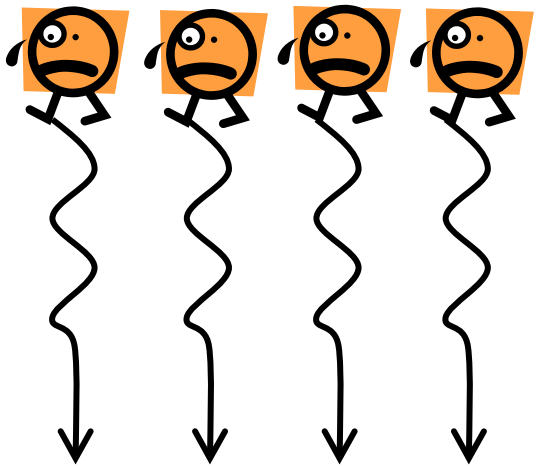
```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[])
{
   MPI_Init(&argc, &argv);
   (do something)
   MPI_Finalize();
}
```

← Initialize MPI

← Finalize MPI

Case with
4 processes

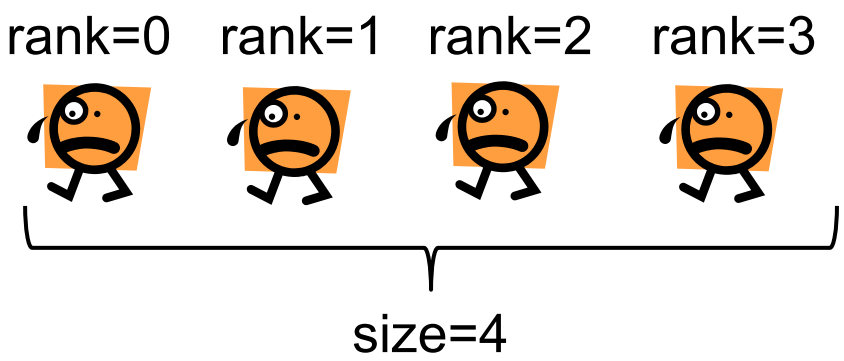


ID of Each MPI Process

- 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

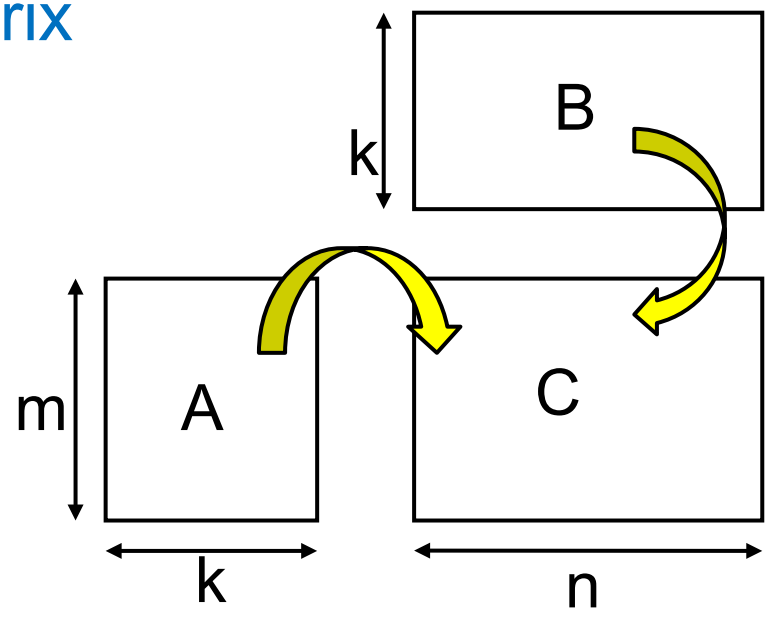
MPI version available at ppcomp-ex/mpi/mm
- Another version: ppcomp-ex/mpi/mm-comm

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

C: a (m × n) matrix

C ← A × B

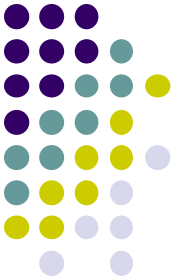
- Algorithm with a triple for loop
- Supports variable matrix size.
  - Each matrix is expressed as a 1D array by *column-major* format



Execution:

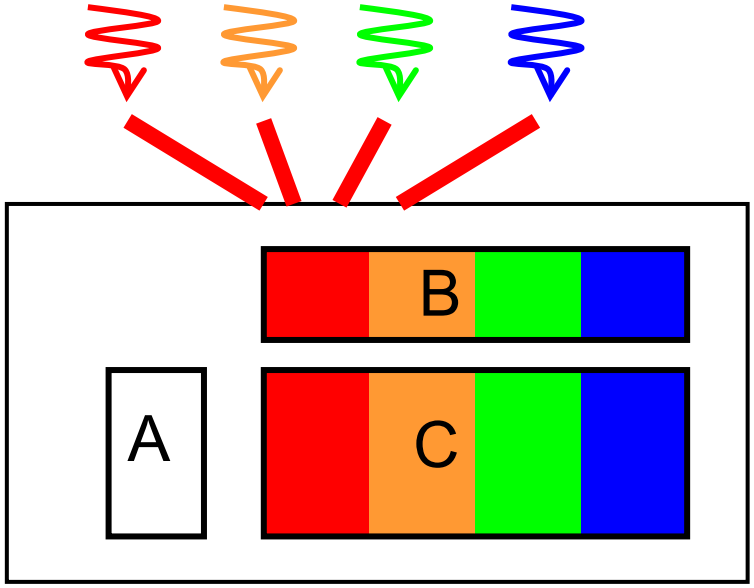
mpiexec -n [np] ./mm [m] [n] [k]

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



Shared memory with OpenMP:

Programmers consider how computations are divided

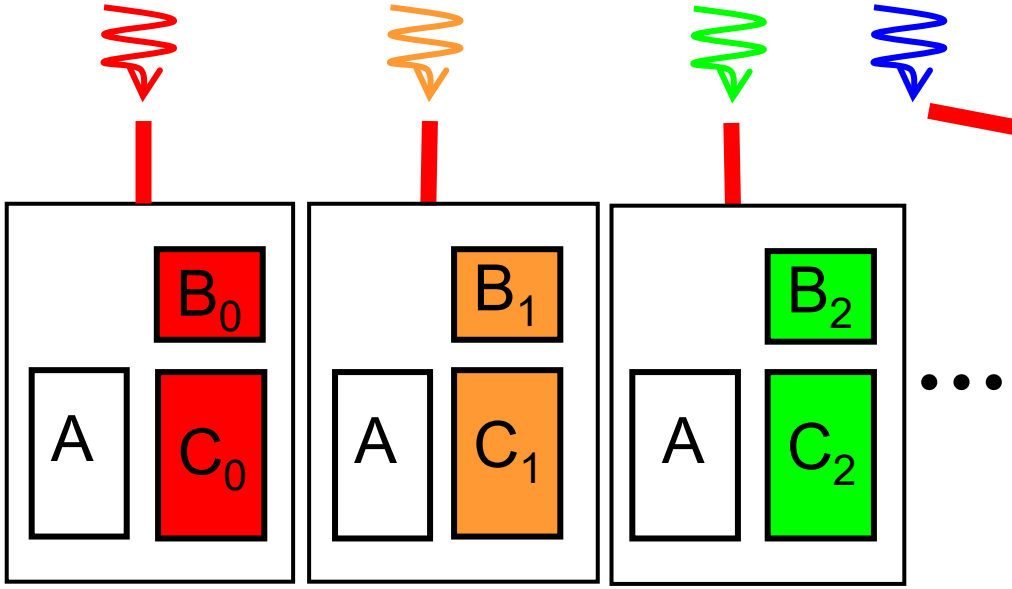


In this case, matrix A is accessed by all threads

→ Programmers do not have to know that

Distributed memory with MPI:

Programmers consider how data and computations are divided

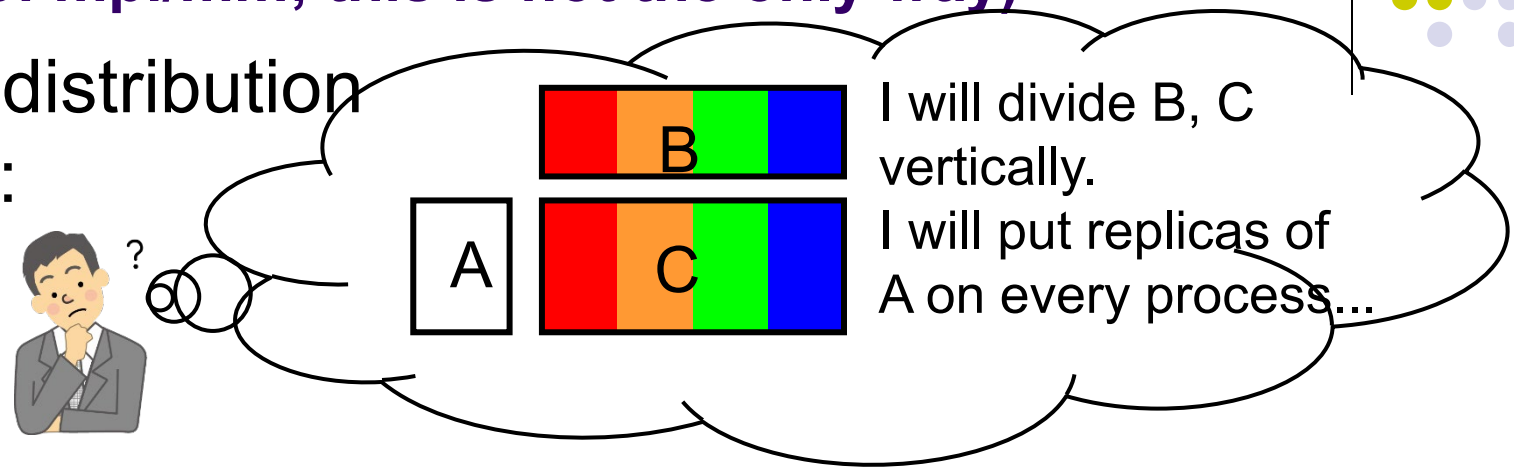


Programmers have to design which data is accessed by each process

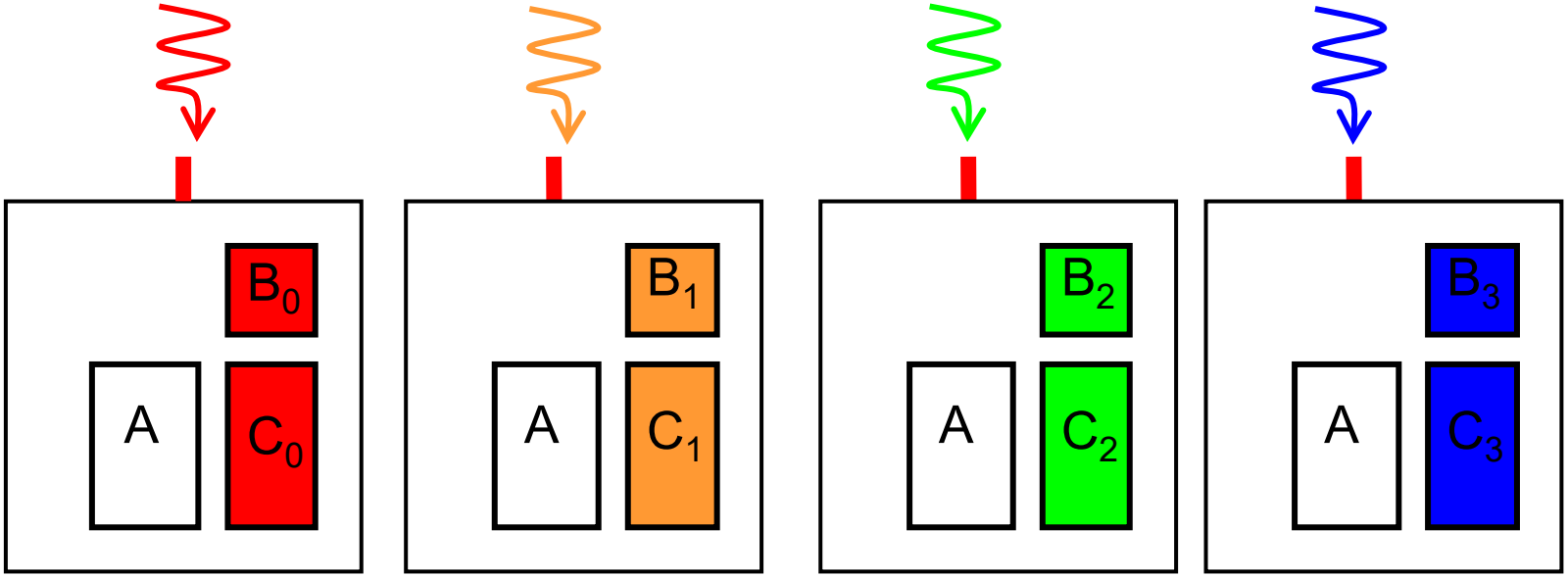
Programming Data Distribution

(Case of mpi/mm; this is not the only way)

Design distribution method:



Programming actual location:



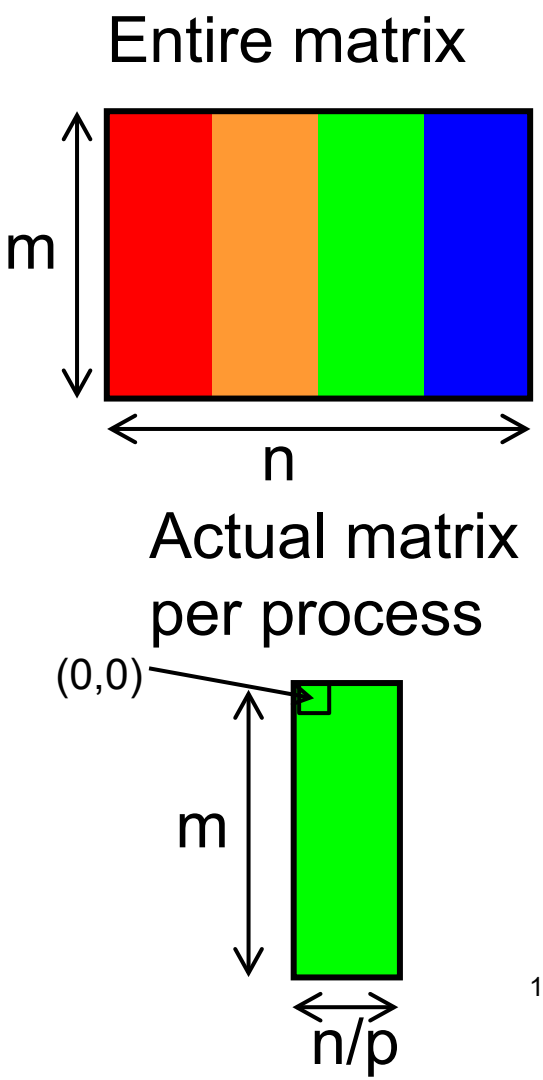
Programming Actual Data Distribution

- 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 ⇔

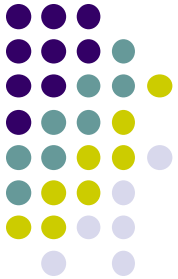
      (i, n/p*r + j) element in entire matrix

local index

global index

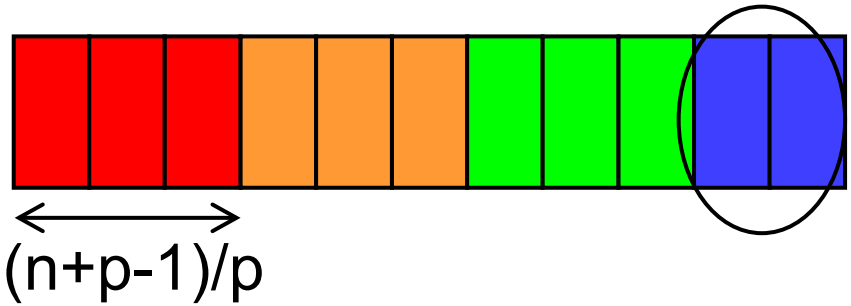


Considering Indivisible Cases



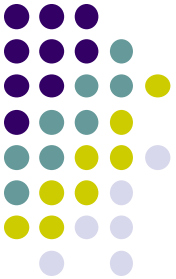
- What if data size n is indivisible by p?
- We let n=11, p=4
  - How many data each process take?
  - n/p = 2 is not good (C division uses round down). Instead, we should use round up division
  - → (n+p-1)/p = 3 works well

Note that the "final" process takes less than others



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

Consideration of Data Placement: Towards *mm-comm* sample

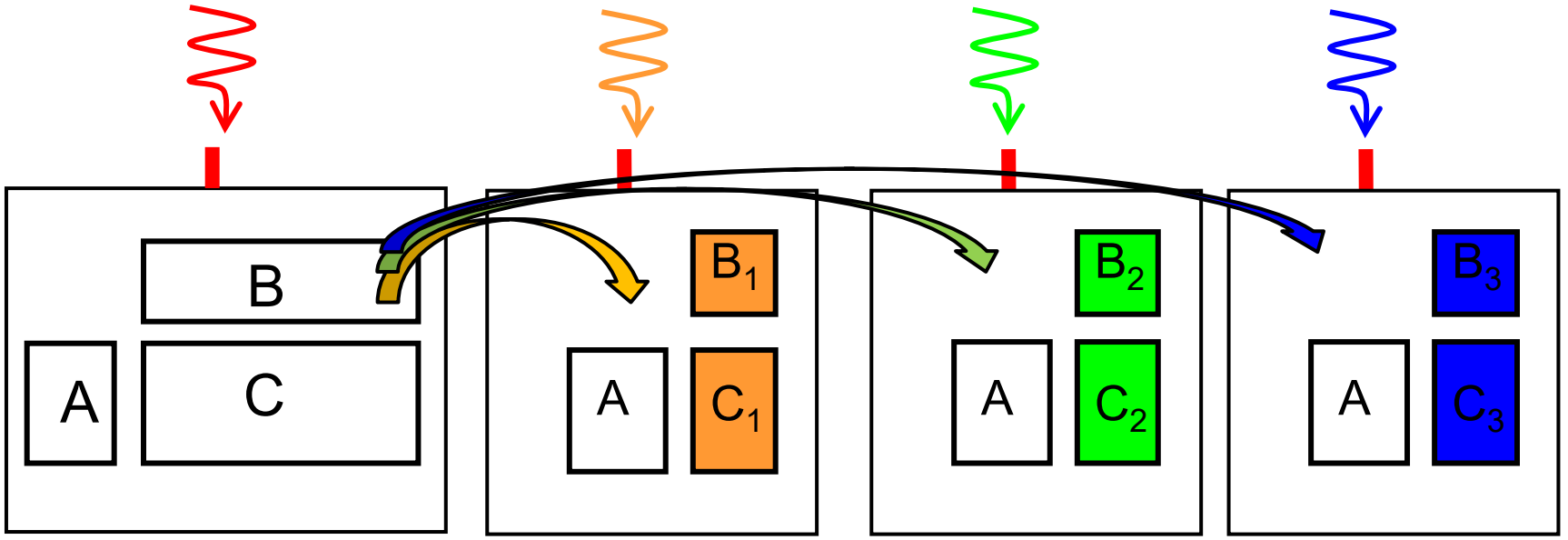


- mpi/mm assumes that initial data placement is convenient for parallelization

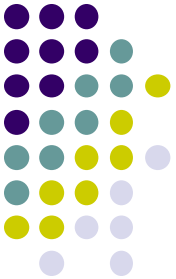
➔ How should we do if placement is not so good?

cf) Only 1 process has the entire data at first. How can we parallelize with MPI?

➔ Communication among processes is required

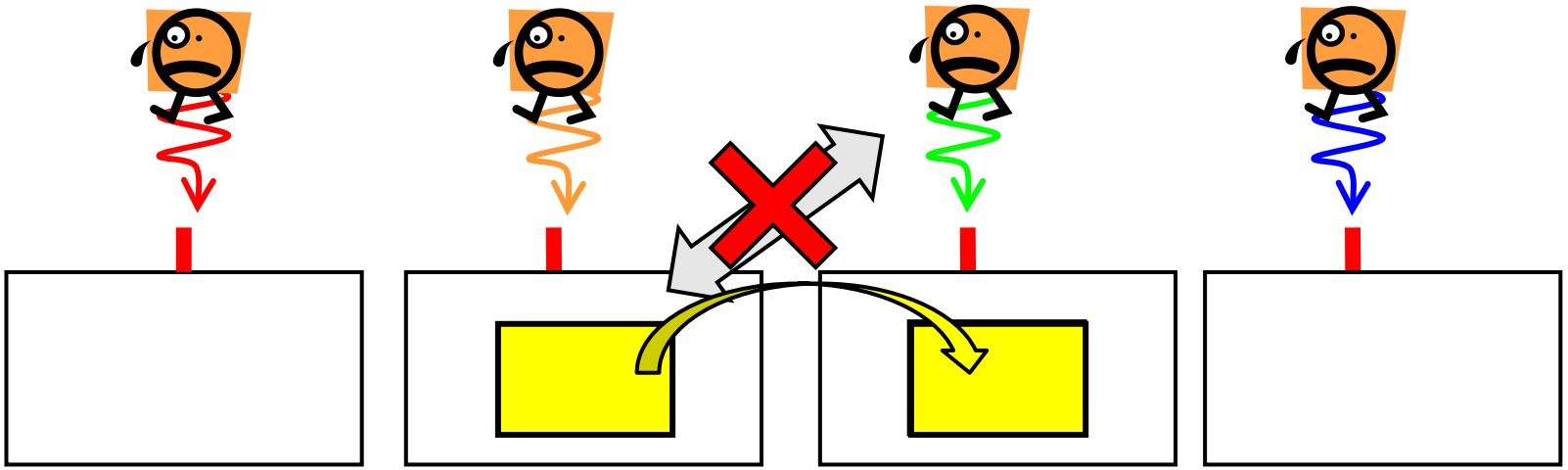


Basics of Distributed Memory Model

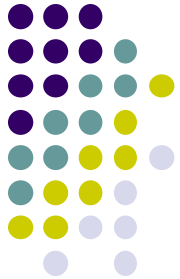


- Each process has its own memory
  - Strictly speaking, each has its address space
- A process cannot access others' memory

➔ Instead, processes can do communication for data transfer with others



mpi/test: A Simple Sample with Communication



[make sure that you are at a interactive node (rXn11)]
module load intel-mpi *[Do once after login]*
[please go to your ppcomp-ex directory]
cd mpi/test
make
[An executable file "test" is created]
mpiexec -n 2 ./test

This sample is for
2 processes

Basics of Message Passing: Peer-to-peer Communication

Example: ppcomp-ex/mpi/test/

Rank 0 computes contents of "int a[16]"

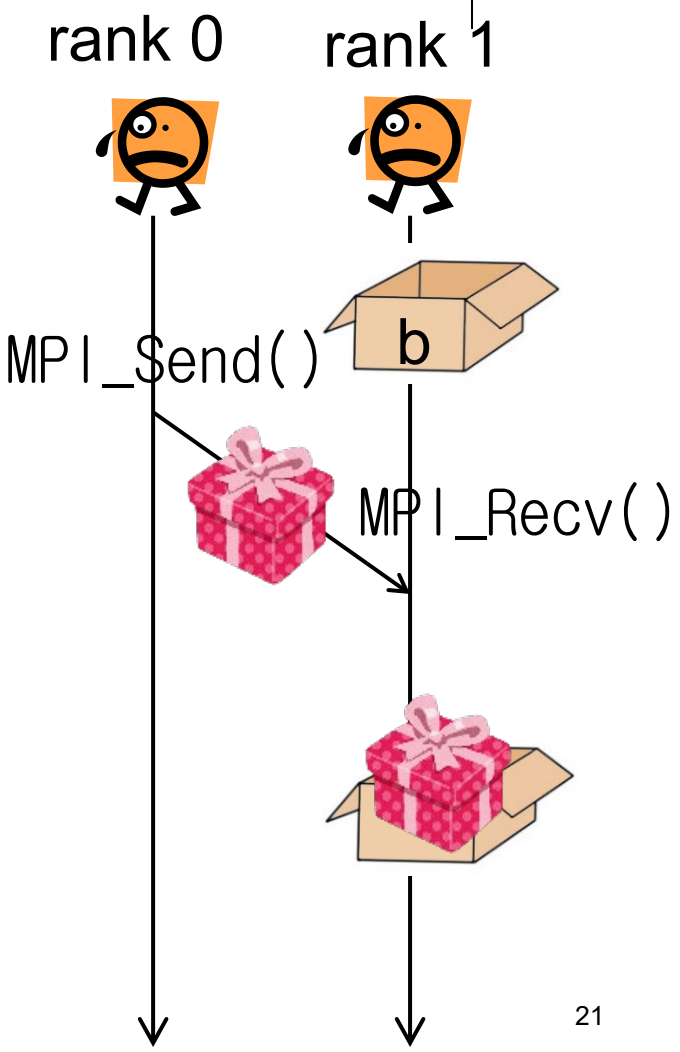
Rank 1 wants to see contents of a!

Rank0:

- Write data to an array a
- MPI_Send(a, 16, MPI_INT, 1, 100, MPI_COMM_WORLD);

Rank1:

- Prepares a memory region (array b here)
- MPI_Recv(b, 16, MPI_INT, 0, 100, MPI_COMM_WORLD, &stat);
- 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)

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)

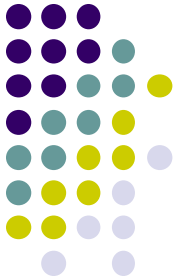
```
MPI_Recv(b, 16, MPI_INT, 2, 200, MPI_COMM_WORLD, &stat);
```

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

→ The message that matches the condition is delivered

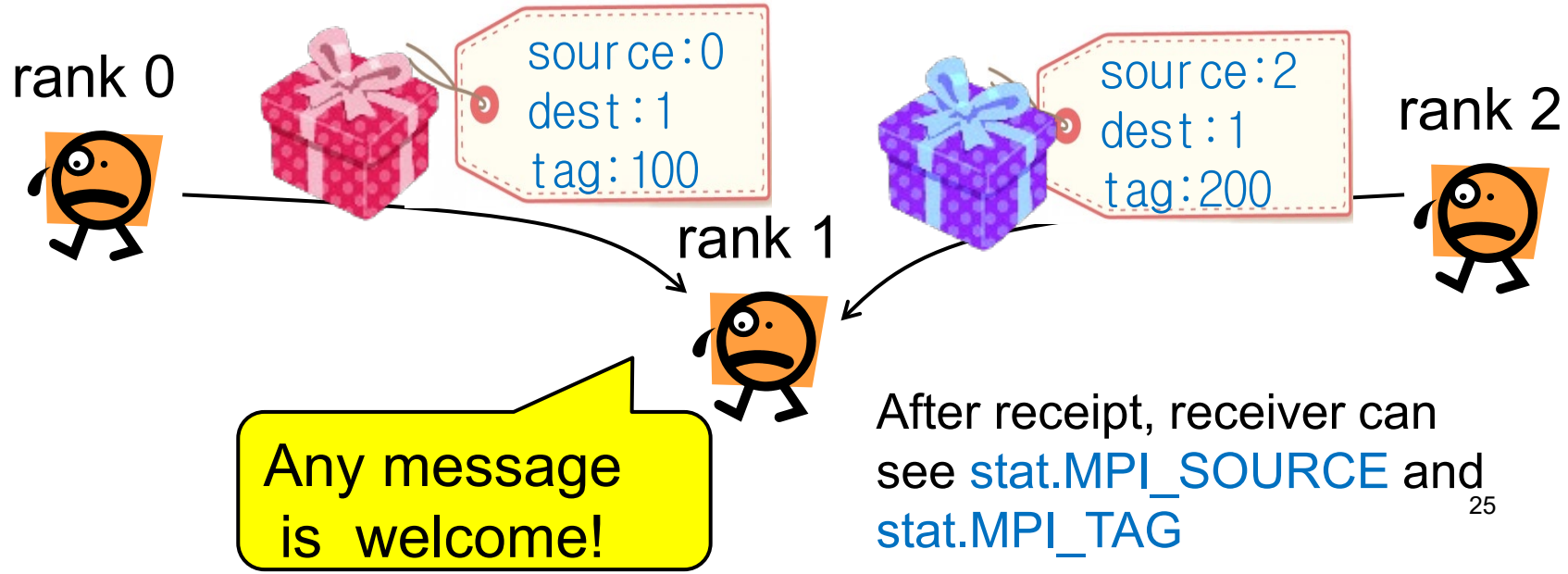
- Other messages should be received by other MPI_Recv calls later

Notes on MPI_Recv: Message Matching (2)

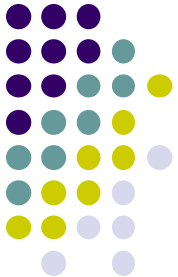


- In some algorithms, the sender may not be known beforehand
  - cf) client-server model
- For such cases, MPI_ANY_SOURCE / MPI_ANY_TAG may be useful

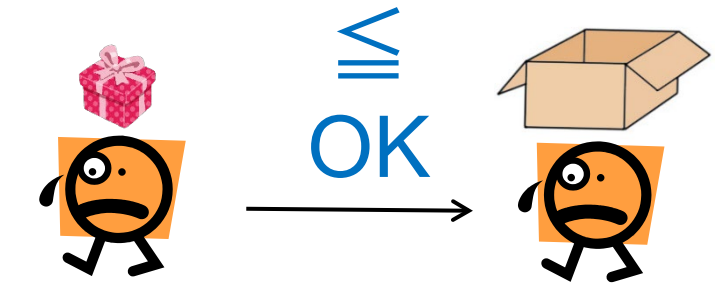
```
MPI_Status stat;
MPI_Recv(b, 16, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
   MPI_COMM_WORLD, &stat);
```



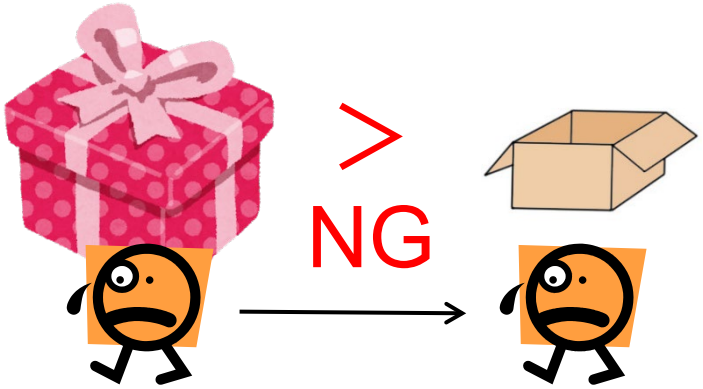
Notes on MPI_Recv: What If Message Size is Unmatched



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



If message is smaller than expected, it's ok
→ Receiver can know the actual size by
MPI_Get_Count(&stat, MPI_INT, &s);



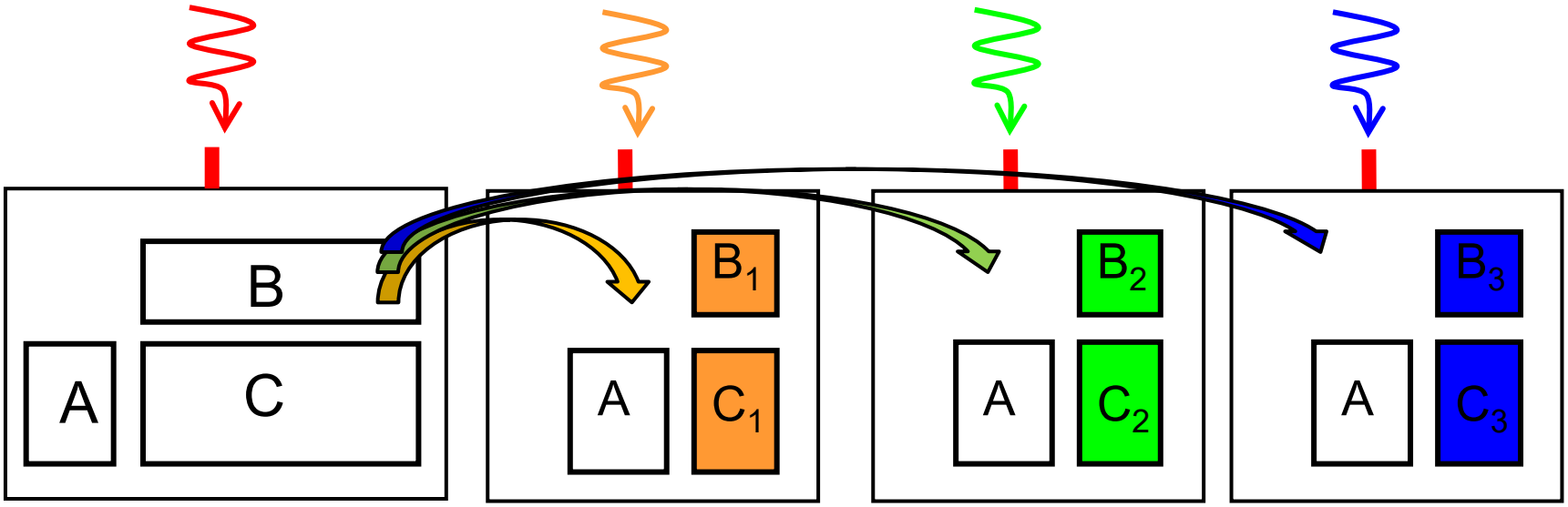
If message is larger than expected, it's an error (the program aborts)

If the message size is UNKNOWN beforehand, the receiver should prepare enough memory

Details of mm-comm Sample (1)

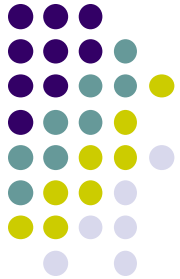
We want to do:

- First, Only rank 0 has all data of A, B, C
- Other ranks receives required data from Rank 0
- Every process does its computation (same as mm sample)
- Rank 0 receives all results C from other processes



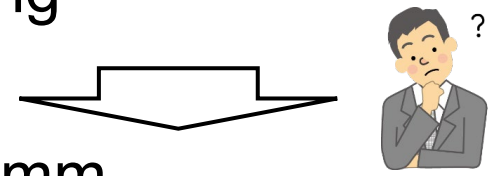
※ This figure is corrected in next discussion

Details of mm-comm Sample (2)



We need to consider rules of MPI:

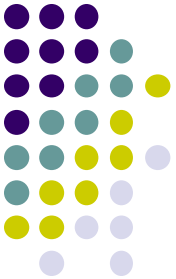
- Sender calls MPI_Send(), receiver calls MPI_Recv()
- The receiver needs to prepare memory buffer before receiving



In mm-comm,

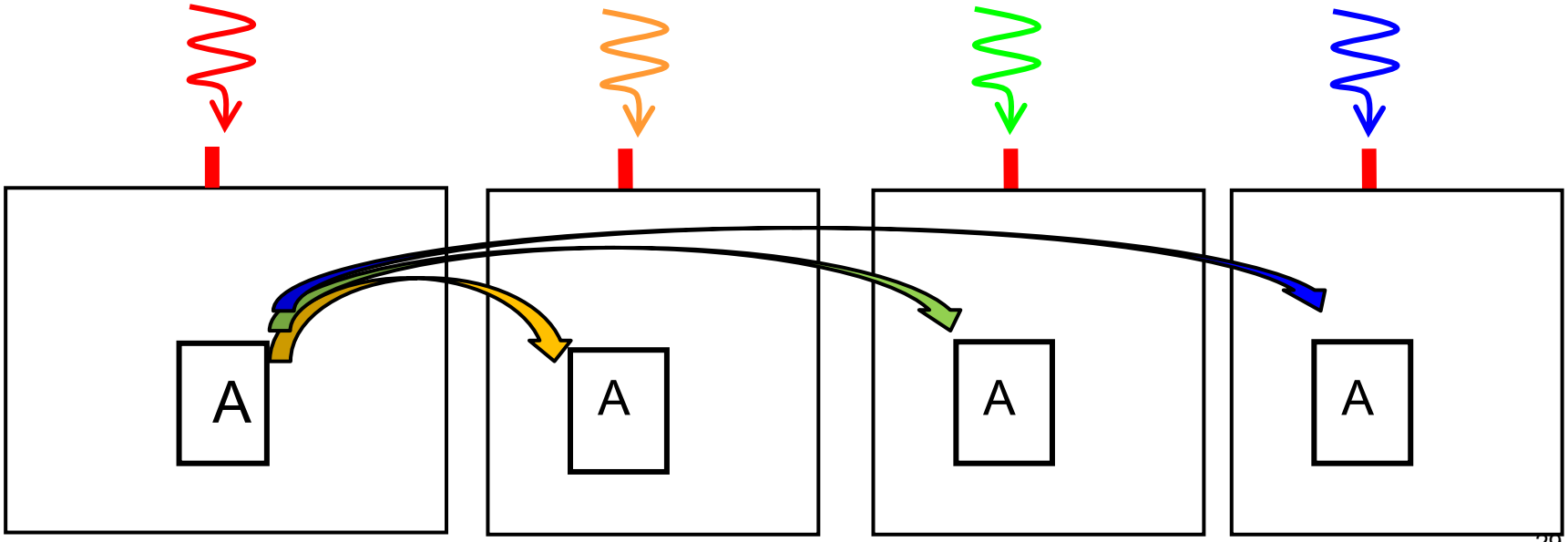
- Rank 0 has
  - A, B, C: initialized ← Only rank 0 has data at first!
  - LB, LC: empty at first
- Each other rank has
  - A, LB, LC: empty at first

Communication of Matrix A in mm-comm

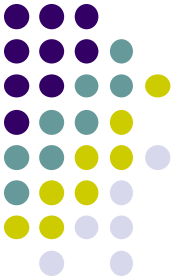


See comm1() in mpi/mm-comm/mm.c

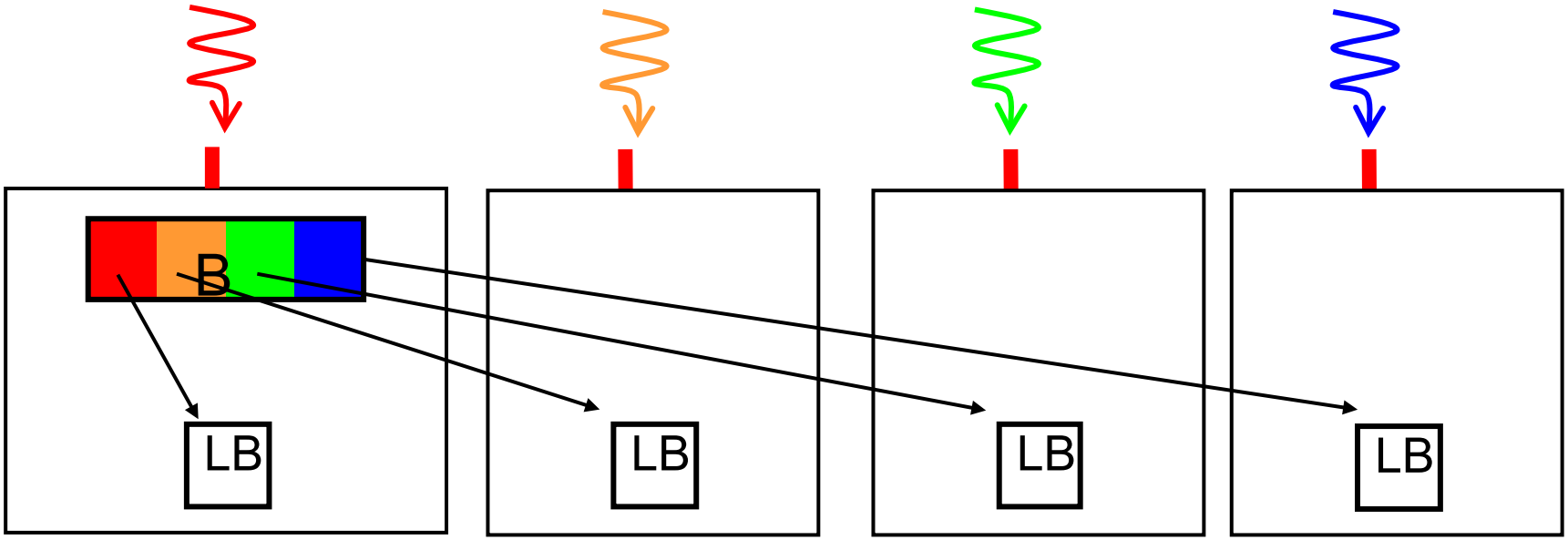
- Data of the entire A is transferred
- Rank 0 calls MPI_Send(A, m*n....) for (p-1) times
- Each of other processes calls MPI_Recv(A, m*n...) once



Communication of Matrix B in mm-comm



- B is “scattered” to processes (unlike A)
- Rank 0 sends partial B to other processes
- Each of other processes receives partial B into its LB array



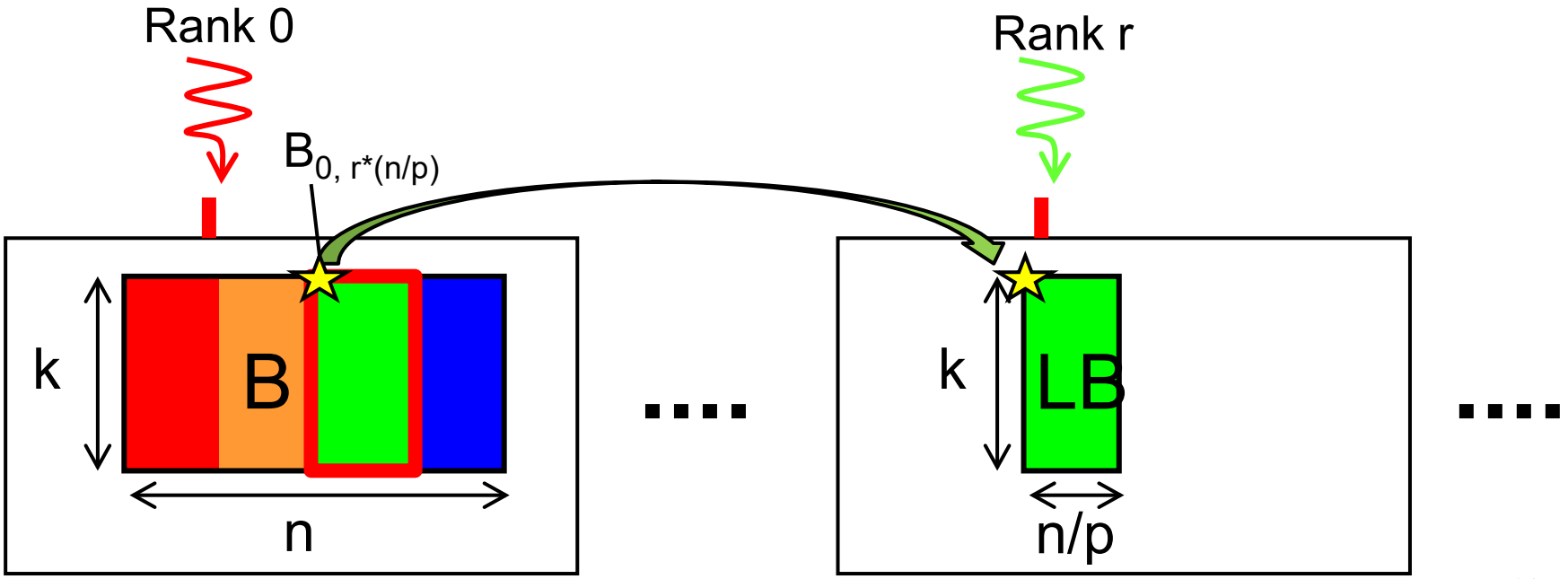
※ On Rank 0, we use memcpy() from B to LB

How to Communicate Partial B

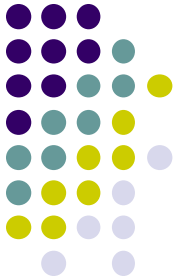
When rank 0 sends partial B to rank r,

- Rank 0 sends data, from $B_{0, r*(n/p)}$ ➔ ★ in B
- Rank ip receives data, from is LB
- Communication size is k*(n/p)

※ mm-comm/mm.c is more complex to support indivisible cases

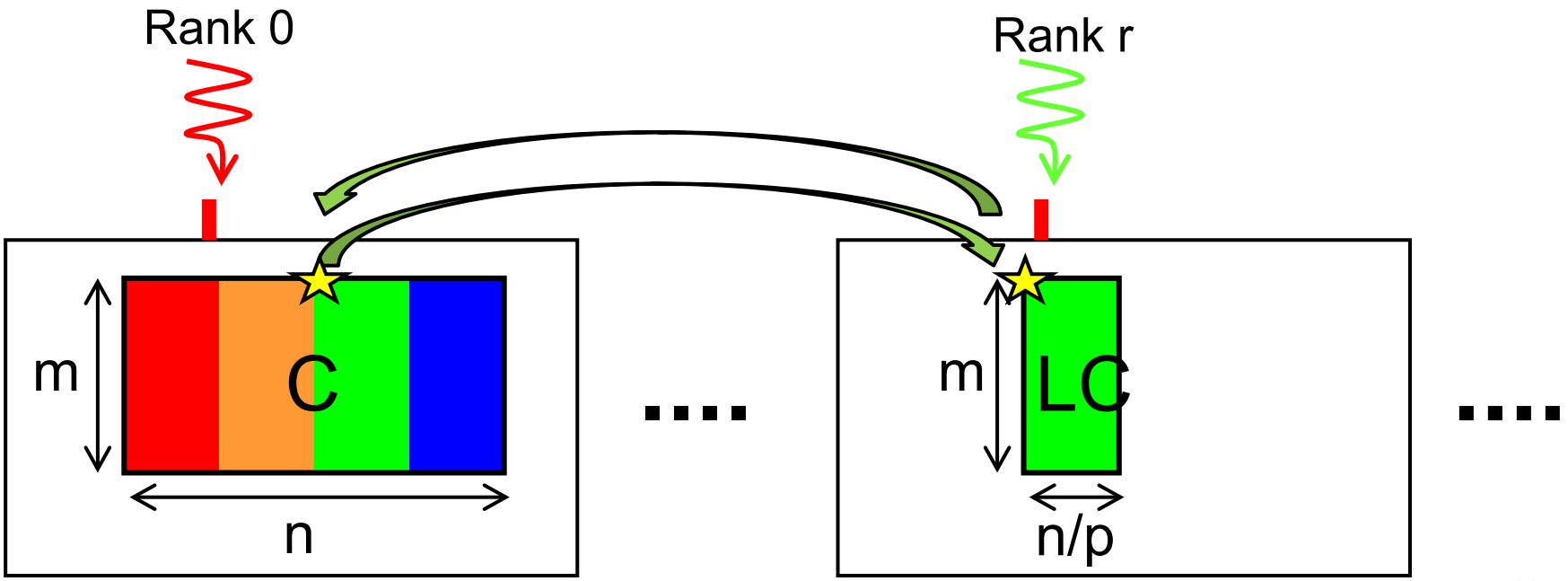


How to Communicate Partial C

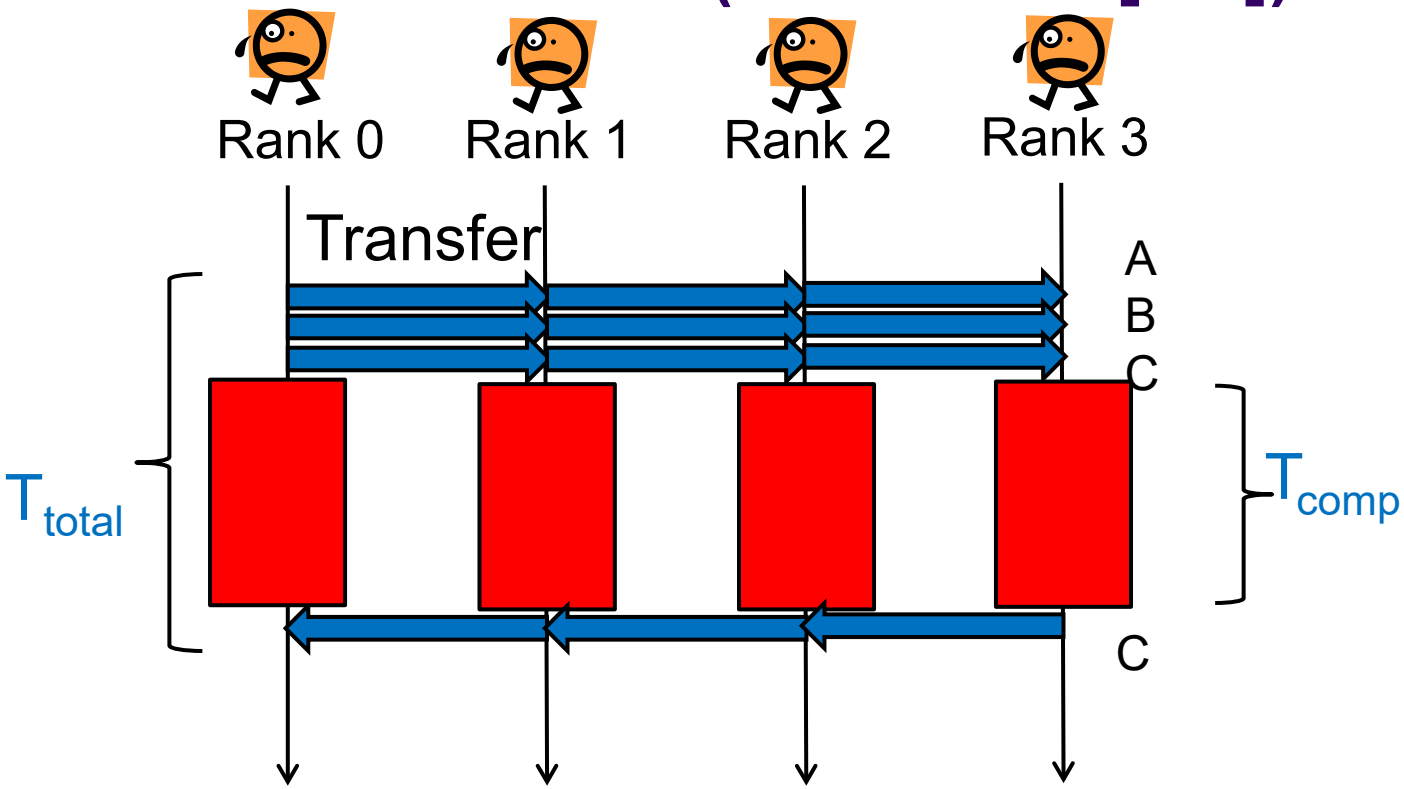


C is similar to B, but

- Matrix height is m
- After computation, LCs are gathered to C in rank 0 (see comm2())



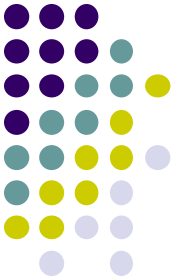
mm-comm Suffers from Transfer Cost (Related to [M3])



- Computation cost (per process): O(mnk/p)
- Transfer cost:
  - O(mkp) for A, O(kn) for B, O(mn) for C

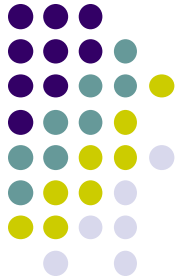
    ↑ entire A is sent for (p-1) times

Assignments in this Course



- There is homework for each part.

OMP Part	[O1] diffusion [O2] bsort [O3] qsort [O4] free	*Select 1 problem*
ACC Part	[A1] diffusion [A2] bsort [A3] mm speed [A4] free	*Select 1 problem*
CUDA Part	[C1] diffusion [C2] bsort [C3] mm speed [C4] free	*Select 1 problem from 2 parts*
MPI Part	[M1] diffusion [M2] bsort [M3] mm speed [M4] free	



Assignments in MPI Part (1)

Choose one of [M1]—[M4], and submit a report

Due date: June 9 (Monday)

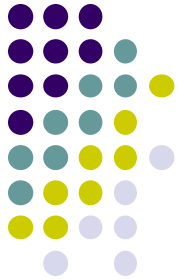
[M1] Parallelize “diffusion” sample program by MPI.

- You can start from /ppcomp-ex/mpi/diffusion
- 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)



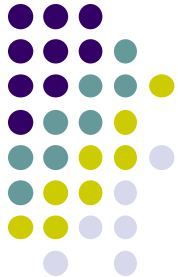
[M2] Parallelize “bsort” sample program by MPI

- You can start from /ppcomp-ex/mpi/bsort
- You can assume
  - array length is power of 2
  - number of processes is power of 2

Optional:

- Comparison with other sort algorithms
  - Quick sort (qsort), Heap sort, Merge sort, …
- Comparison with OpenMP or GPU...

Assignments in MPI Part (3)



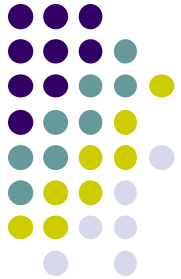
[M3] Evaluate speed of “mpi/mm” and “mpi/mm-comm” samples in detail

- Compare speed of mpi/mm and mpi/mm-comm
- Use various matrices sizes
- Evaluate effects of data transfer cost

Optional:

- To improve mm-comm with group communication
- To improve the algorithm to reduce memory consumption
- To try Advanced algorithm, such as SUMMA
  - the paper “*SUMMA: Scalable Universal Matrix Multiplication Algorithm”* by Van de Geijn
  - http://www.netlib.org/lapack/lawnspdf/lawn96.pdf

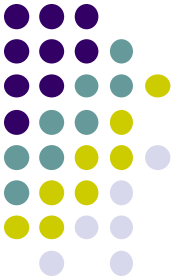
Assignments in MPI Part (4)



[M4] (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)

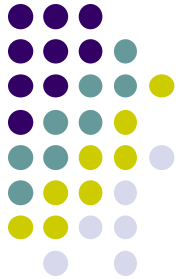


- Submit the followings via T2SCHOLA

  (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
    - [MPI] Up to 4 processes on an interactive node is enough
    - [MPI] To use more processes or multiple nodes, you need to do “job submission” (optional)
  - With varying problem sizes
  - Discussion with your findings
  - Other machines than TSUBAME are ok, if available

Plan of MPI Part

- Class #12
  - Introduction to MPI, message passing
- Class #13
  - Non-blocking communication, group communication
- Class #14
  - Performance improvement
- Class #15 (May 29, Optional)
  - TSUBAME supercomputer tour