2018 Practical Parallel Computing (実践的並列コンピューティング) No. 7

Distributed Memory Parallel Programming with MPI (1)

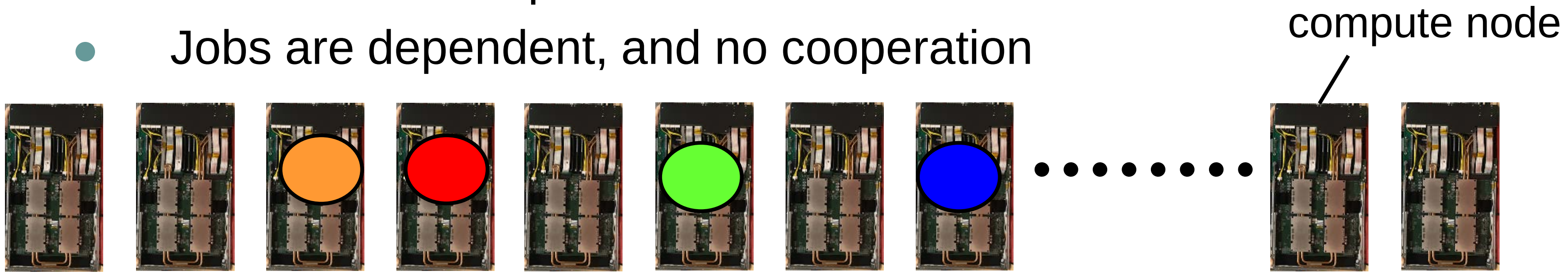
Toshio Endo

School of Computing & GSIC

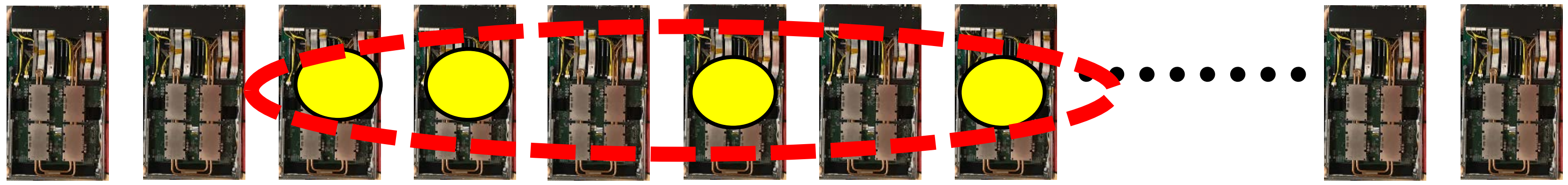
endo@is.titech.ac.jp

How Can We Use Many Nodes in Supercomputers?

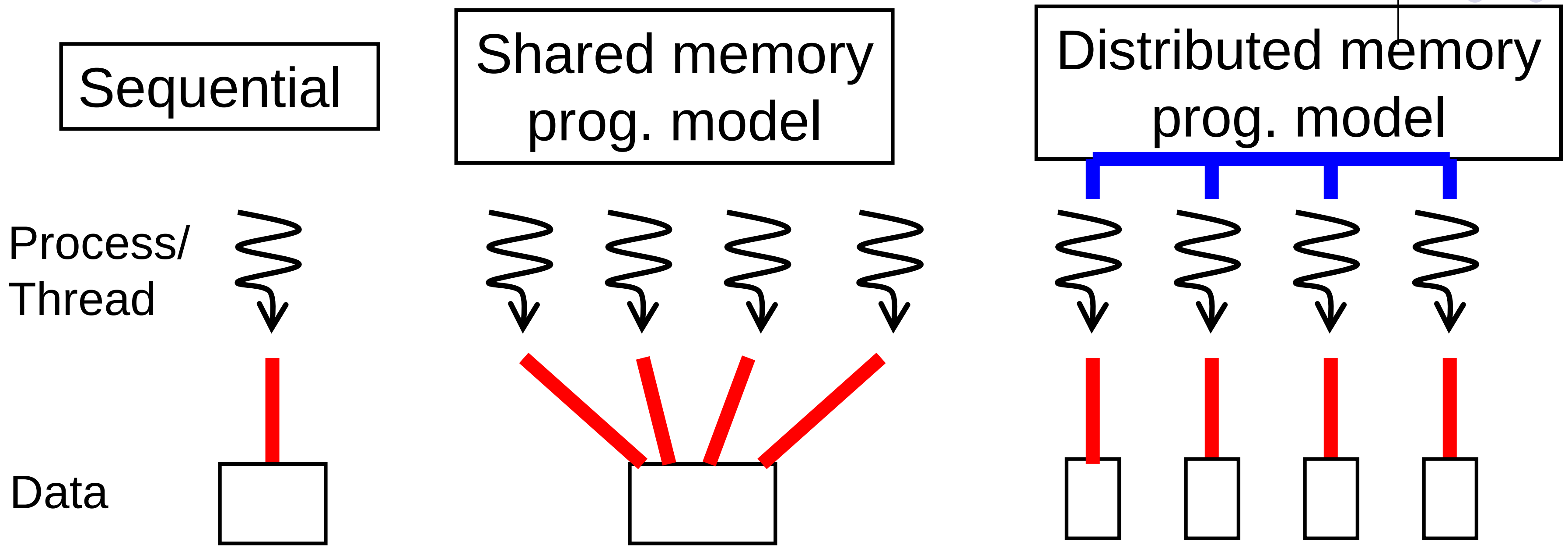
1. Throw several jobs into job scheduler
   - cf) Program executions with different parameters → Parameter Sweep
   - Jobs are dependent, and no cooperation



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
  - Due to two reasons:
    - MPI is older than OpenMP
    - Distributed memory makes load balancing difficult

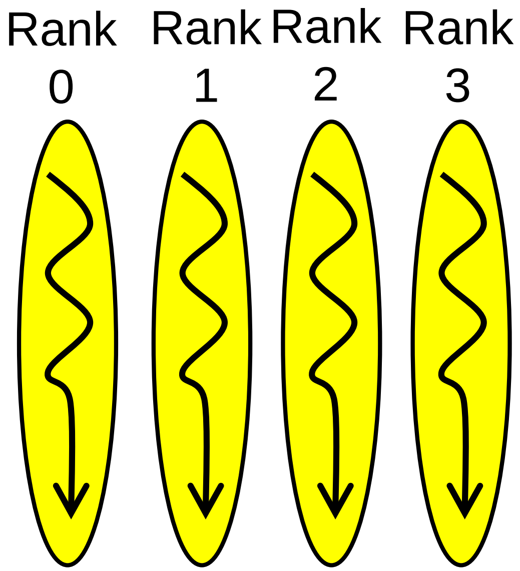
A MPI Program Looks Like

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

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

   (Some computation/communication)

   MPI_Finalize();        ← Finalize MPI
}
```



Sample Programs on TSUBAME2 (in case of OpenMPI)

Samples at ~endo-t-ac/ppcomp/18/mpitest/
~endo-t-ac/ppcomp/18/mm-mpi/

- Preparation for MPI environment
  - module load cuda ← for module dependency☹
  - module load openmpi
- MPI programs are compiled with mpicc command
  - In sample directories, “make” command will be ok
- Program execution (in case of qrsh)
  - $\text{mpirun –n } \underset{\text{Number of processes}}{4}\ \underbrace{\text{./myprog a b}}_{\text{program name and its options}}$

Throw an MPI Job

- Here program name is “myprog”. 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
module load openmpi

mpirun –n 8 –npernode 4 ./myprog a b
```

4core node x 2

Number of processes

Number of processes per node

(2) Throw the job with “qsub”

```
qsub job.sh
```

← no group

```
qsub –g tga-ppcomp job.sh
```

← with group name

Notes on Job Submission

- Please specify maximum run time (h_rt) properly
  - If h_rt is larger than 0:10:00, you need to specify “TSUBAME group name” for accounting (charged/有料)

    ```
    qsub –g tga-ppcomp job.sh
    ```

  - Use tga-ppcomp group only for this lecture / tga-ppcompグループは、本授業の課題とそのテスト専用に使ってください
- Please do not execute CPU intensive programs on login nodes
  - It is OK to edit programs, compile programs, and submit jobs, and so on
  - “qrsh” may help you. See Section 4.3 in User’s Guide
- [new!] Without TSUBAME group, you can only use ≦2 nodes / グループ無しの無料利用は2ノードまで
  - If number of nodes > 2, group name is required (Charged)
  - For the assignments, please use 256 cores or less

ID of Each 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

Basics of MPI: Send and Receive of a message

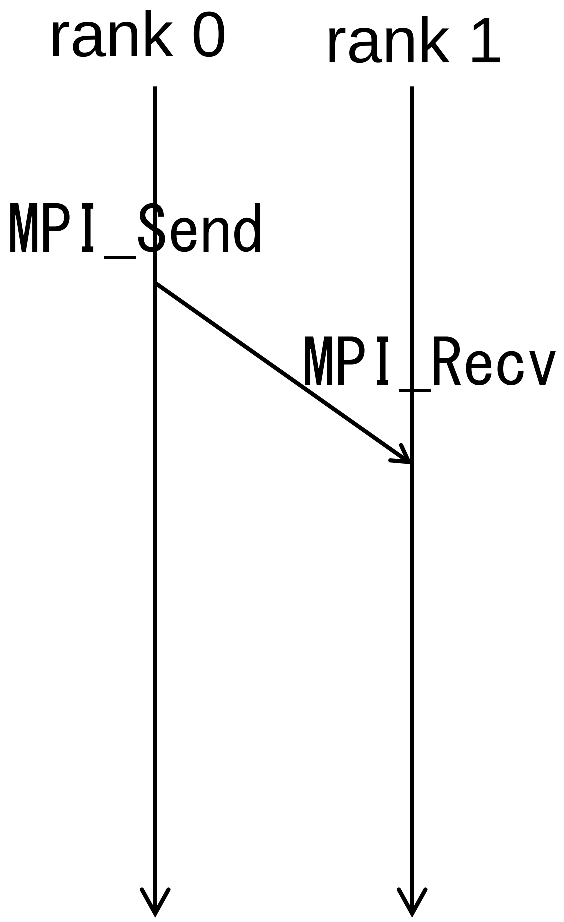
In order to send contents of "int a[16]" from rank 0 to rank1

- rank0 calls

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

- rank1 calls

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



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

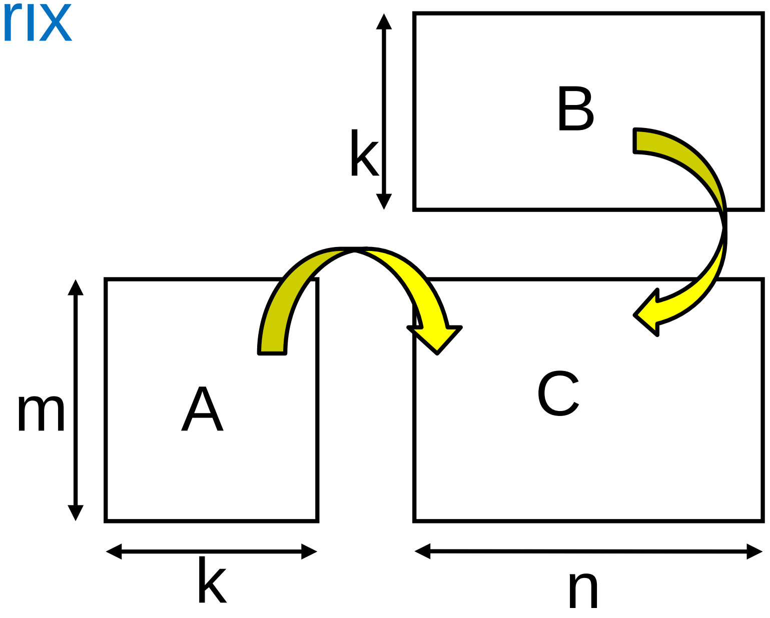
“mm” sample: Matrix Multiply

MPI version available at ~endo-t-ac/ppcomp/18/mm-mpi/

A: a $(m \times k)$ matrix, B: a $(k \times n)$ matrix

C: a $(m \times n)$ matrix

$C \leftarrow A \times B$



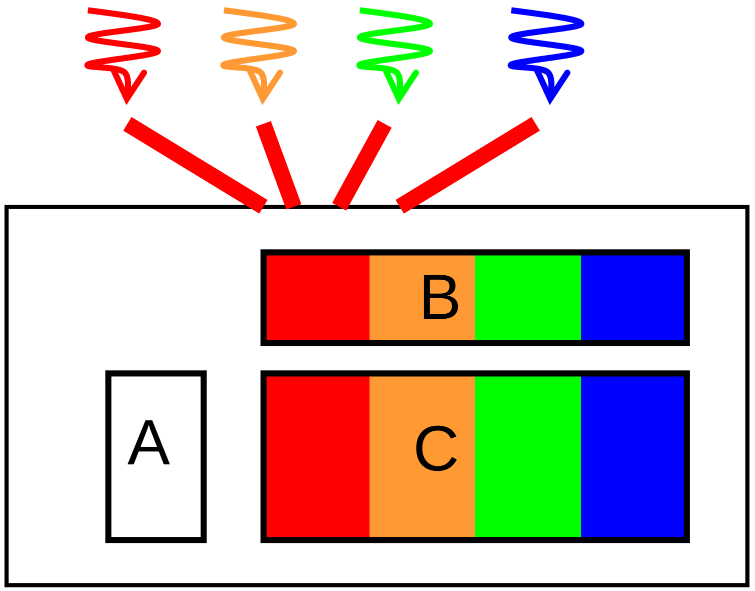
- Algorithm with a triple for loop
- Supports variable matrix size.
  - Each matrix is expressed as a 1D array by *column-major* format

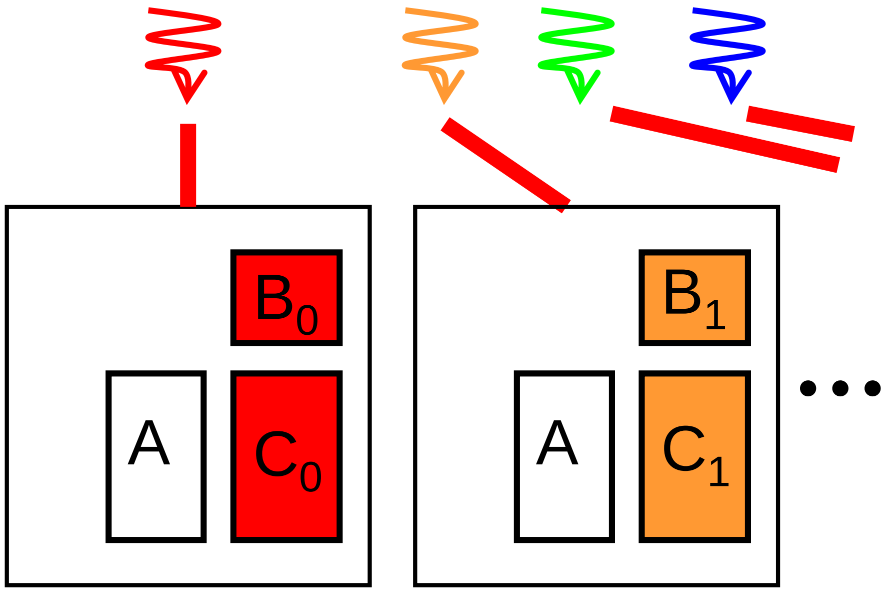
- Execution: mpirun –n [np] –npernode [nn] ./mm [m] [n] [k]

Why Distributed Programming is More Difficult

Programming matrix multiplication

- Shared memory: Programmers consider how computations are divided
- Distributed memory: Programmers consider how data and computations are divided





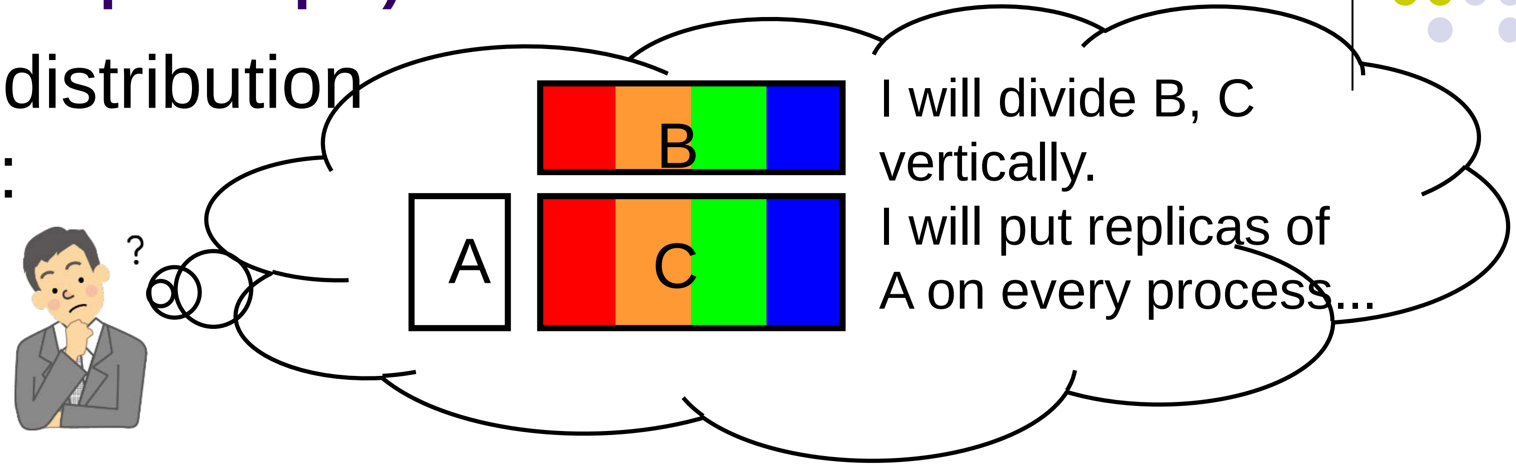
In this case, matrix A is accessed by all threads
→ Programmers do not have to know that

Programmers have to design which data is accessed by each process

Programming Data Distribution

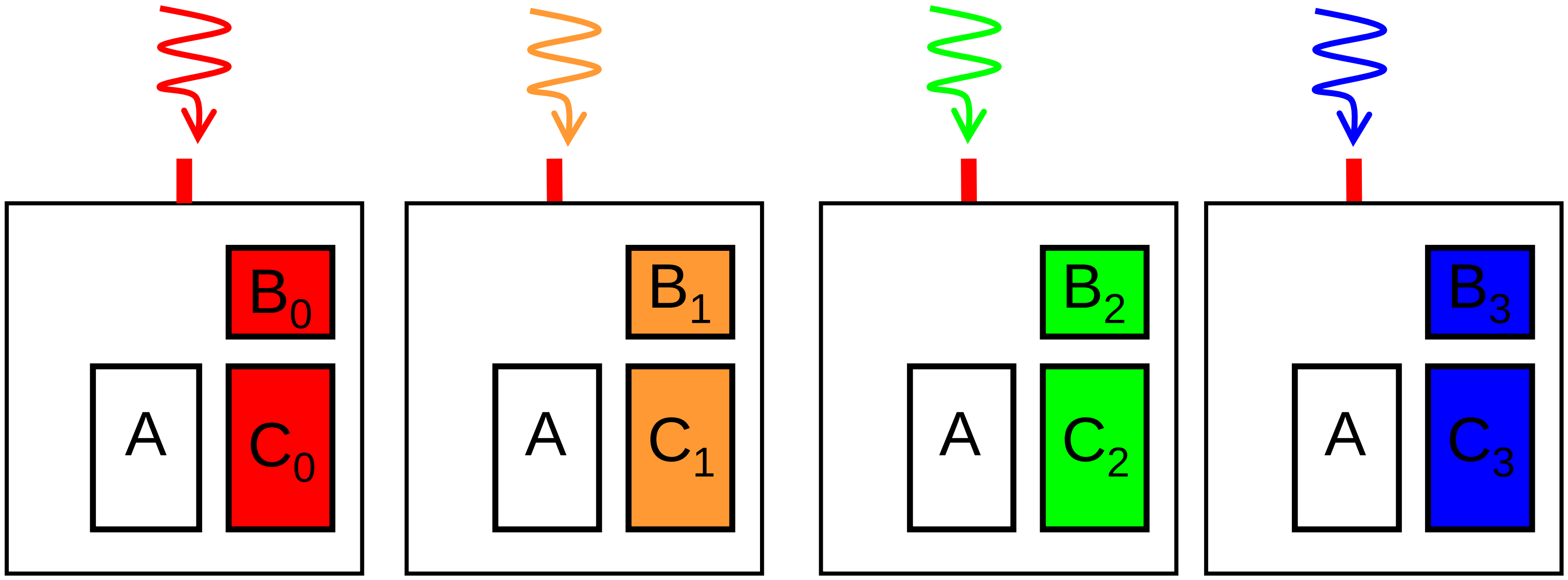
(for mm-mpi sample)

Design distribution method:



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

Programming actual location:



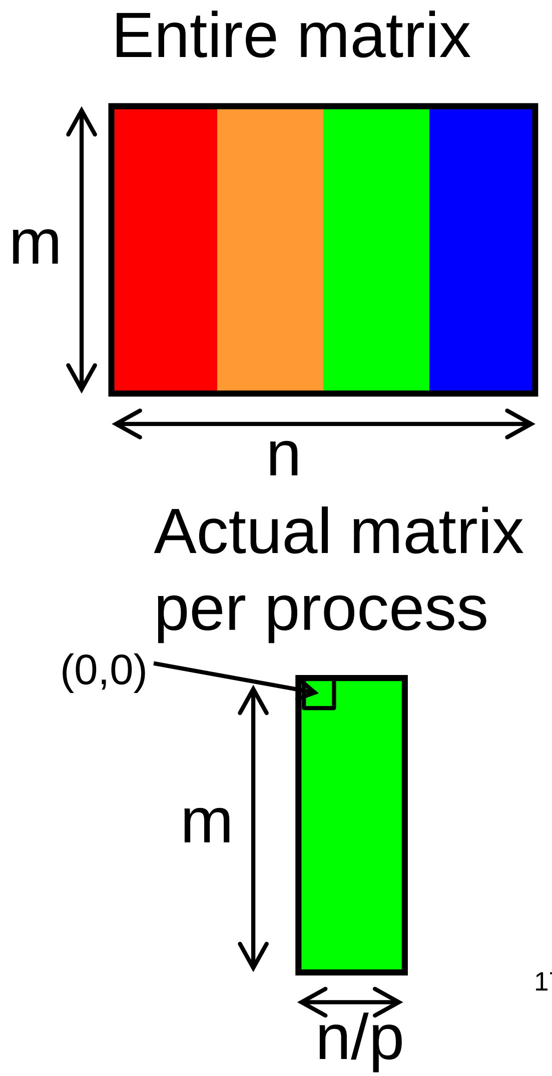
Programming Actual Data Distribution

- 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 of partial matrix owned by Process r ⇔

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

local index

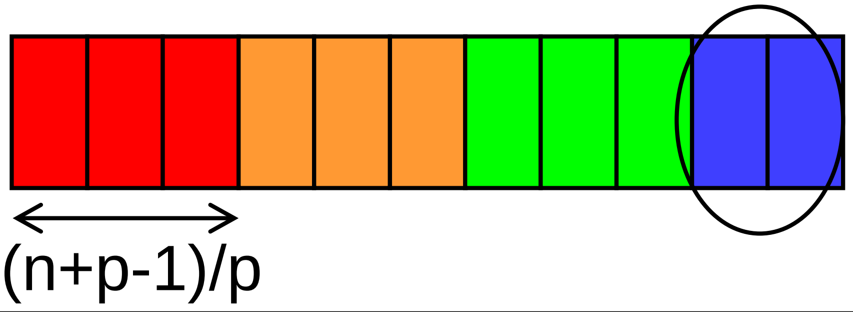
global index



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)

Assignments in this Course

- There is homework for each part. Submissions of reports for 2 parts are required
- Also attendances will be considered

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

Assignments in MPI Part (1)

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

Due date: May 28 (Monday)

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

Optional:

- Make array sizes variable parameters
- Improve performance further. Blocking, SIMD instructions, etc, may help
- Considering fractions, in the case with NY is not divisible by the number of processes

Assignments in MPI Part(2)

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

Optional:

- Considering fractions
- Trying advanced algorithms, such as SUMMA (Scalable Universal Matrix Multiplication Algorithm)[Van de Geijn 1997] is good

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 Submission

- Submit the followings via OCW-i

  (1) A report document

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

  (2) Source code files of your program

  - If you use multiple files, you can use “.zip” or “.tgz”

- Report 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 processor cores
    - 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