

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

## Part 3: MPI

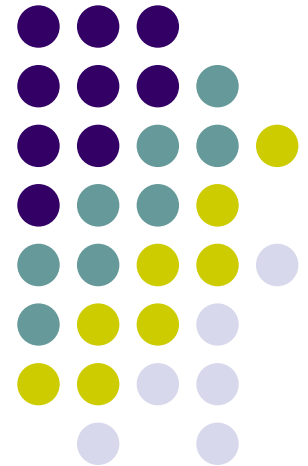
### No 3: Communication Costs

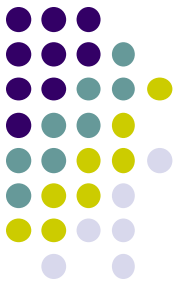
May 30, 2024

Toshio Endo

School of Computing & GSIC

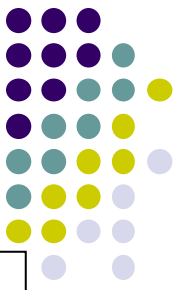
[endo@is.titech.ac.jp](mailto:endo@is.titech.ac.jp)





# 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
  - 4 classes      ← We are here (3/4)



# “mm” sample: Matrix Multiply

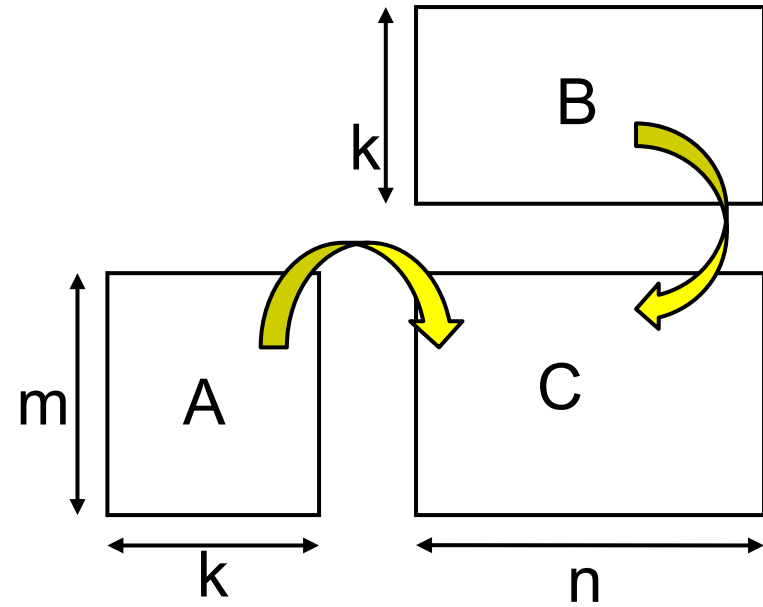
MPI version available at [/gs/bs/tga-ppcomp/24/mm-mpi/](https://github.com/tga-ppcomp/24/mm-mpi/)

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

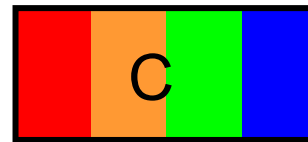
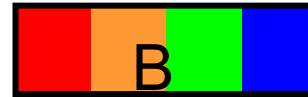


Execution: `mpiexec -n [#proc] ./mm [m] [n] [k]`

# Programming Data Distribution

(for mm-mpi sample)

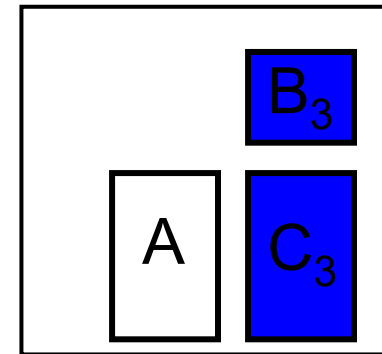
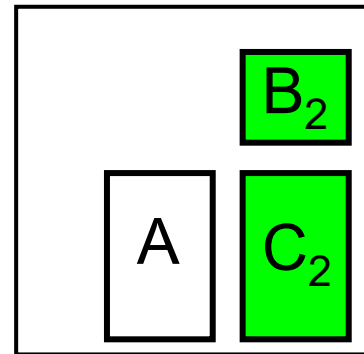
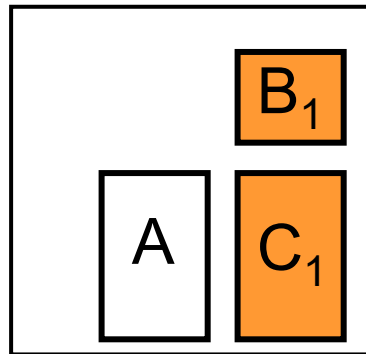
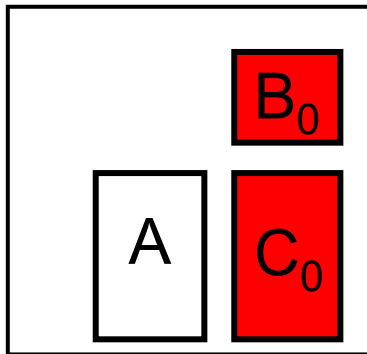
Design distribution method:



I will divide B, C vertically.

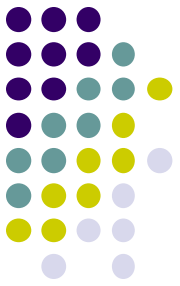
I will put replicas of A on every process...

Programming actual location:



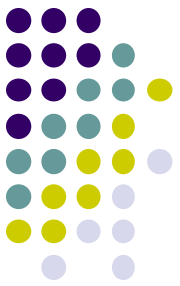
This is not a unique way. Let us discuss other ways

# Discussion on Considering Data Distribution

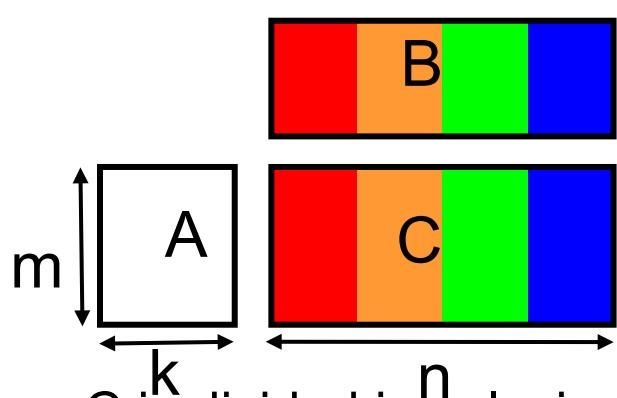


- Choice of data distribution have impact on
  - Communication cost
  - Memory consumption cost
    - In mm-mpi, every process has a copy of matrix  $A \rightarrow$  memory consumption is larger ☹️
- Smaller cost is better

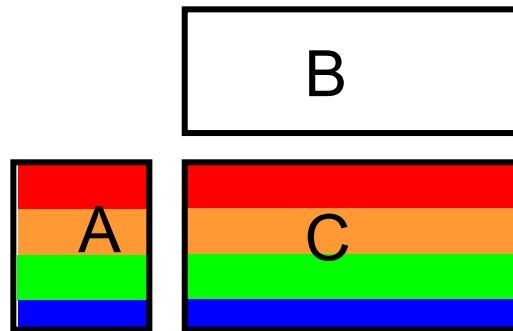
# Other Data Distribution Methods?



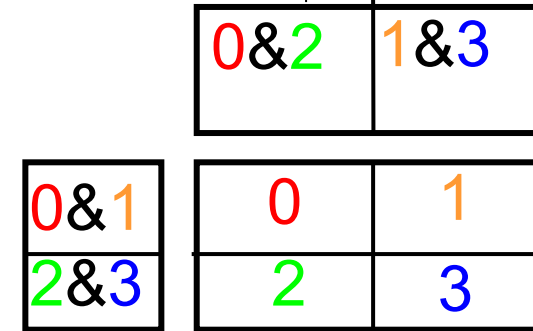
- $C_{i,j}$  requires  $i$ -th row of  $A$  and  $j$ -th column of  $B$



C is divided in col-wise  
 $\Rightarrow$  Similarly B  
 A is replicated



C is divided in row-wise  
 $\Rightarrow$  Similarly A  
 B is replicated



C is divided in 2D  
 $\Rightarrow$  A:row-wise + replica  
 B:col-wise + replica

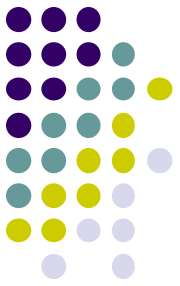
<b>Total Comm.</b>	0	0	0
<b>Total Mem.</b>	$O(mkp+nk+mn)$	$O(mk+nkp+mn)$	$O(mkp^{1/2}+nkp^{1/2}+mn)$

$p$ : the number of processes

Note: If initial matrix is owned by one process, we need communication before computation

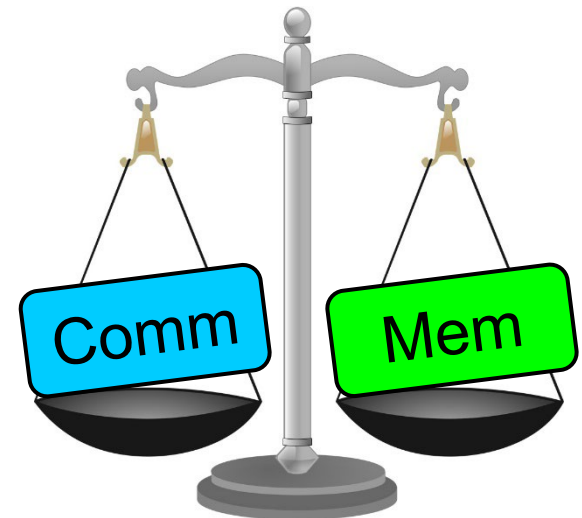
Among them, the third version has lowest memory consumption

# Reducing Memory Consumption Further

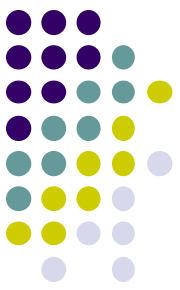


- Even in the third version, memory consumption is  $O(mkp^{1/2} + nkp^{1/2} + mn) > O(mk + nk + mn)$  (theoretical minimum)
- If  $p=10000$ , we consume 100x larger memory ☹️  
→ we cannot solve larger problems on supercomputers
- To reduce memory consumption, we want to **eliminate replica!**  
→ But this increases communication costs

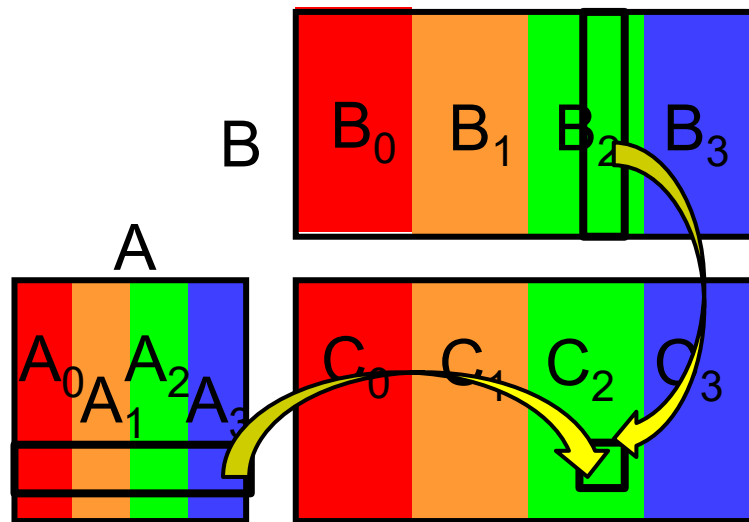
*Trade-off:* a balance achieved between two desirable but incompatible features



# Data Distribution of Memory Reduced “mm” (related to [M2])



- Not only B and C, but A is divided among all processes  
(In this example, column-wise)

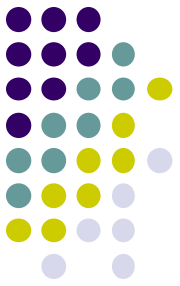


Memory consumption  
is smallest

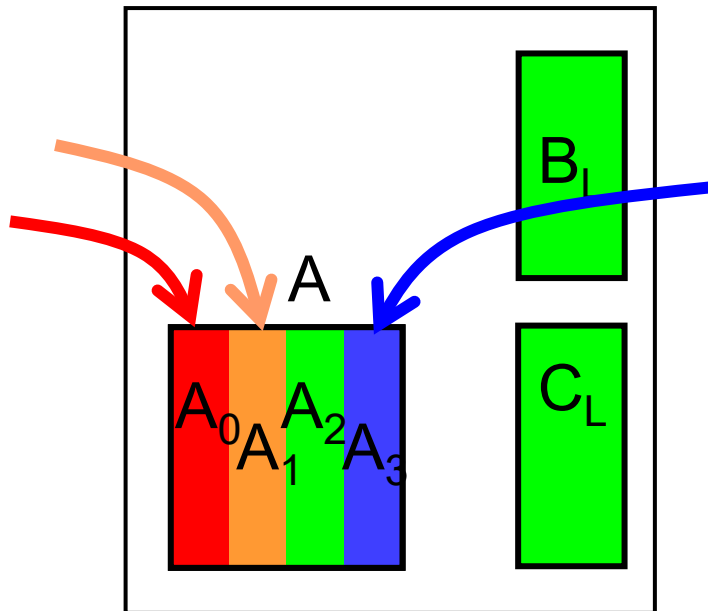
- But computing each C element requires data on other processes → We need **communication** !



# How We Proceed Computation with Others' Data



- The following algorithm is not good for memory consumption

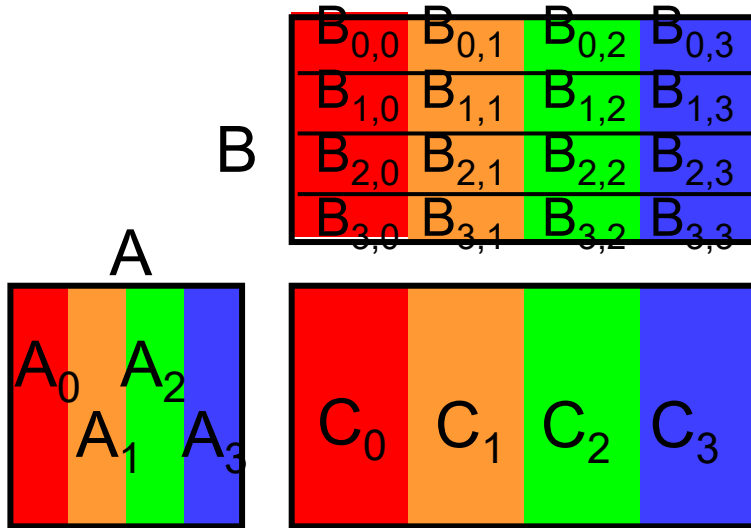


1. Collect entire A from other processes by communication
  2. Compute  $C_L = A \times B_L$
- Each process has (entire) A,  $B_L$ ,  $C_L$  → Same as mm-mpi ☹️

We should avoid computation of  $C_L = A \times B_L$  at once



# Algorithm of Memory Reduced “mm”



If we have A only partially,  
we can only do  $C_L = A \times B_L$   
partially

## Algorithm

### Step 0:

$P_0$  sends  $A_0$  to all other processes  
Every process  $P_r$  computes

$$C_r += A_0 \times B_{0,r}$$

### Step 1:

$P_1$  sends  $A_1$  to all other processes  
Every process  $P_r$  computes

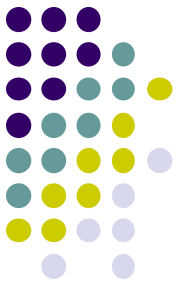
$$C_r += A_1 \times B_{1,r}$$

:

Repeat until Step (p-1)

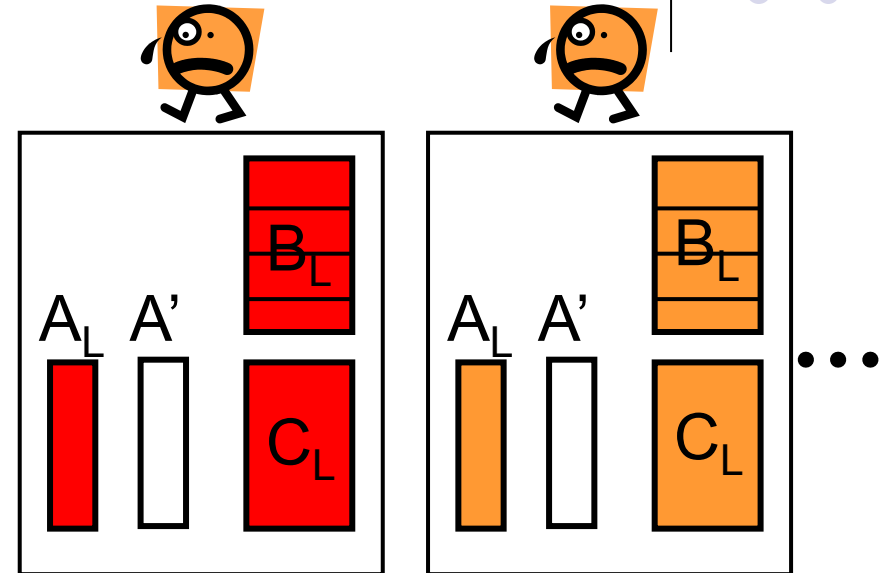
**Total Comm:**  $O(mkp)$     **Total Mem:**  $O(mk+nk+mn)$

# Actual Data Distribution



Every process has partial A, B, C

- $A_L$  on process  $r \Leftrightarrow A_r$
- $B_L$  on process  $r \Leftrightarrow B_r$
- $C_L$  on process  $r \Leftrightarrow C_r$



- Additionally, every process should prepare a receive buffer  $\rightarrow A'$  in the figure
  - $A'$  (instead of  $A_L$ ) is used for arguments of `MPI_Recv()`
  - On receivers,  $A'$  is used for computation

[Q] What if a process uses  $A_L$  for `MPI_Recv()`'s target ?

# Programming

## Memory Reduced mm



On every **process**  $r$ :

```
for (s = 0; s < p; s++) { // s: step no, p: number of processes
```

```
    if ( $r == s$ ) {
```

```
        for (dest = 0; dest < p; dest++)
```

```
            if (dest  $\neq$  r) MPI_Send( $A_L$ , ..., dest, ...);
```

$P_s$  sends its  $A_L$  to all other processes

```
    } else
```

```
        MPI_Recv( $A'$ , ..., s, ...);
```

Receives data ( $P_s$ 's  $A_L$ ) and stores it to  $A'$

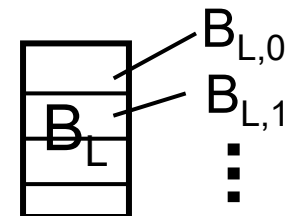
```
    if ( $r == s$ )
```

```
        Compute  $C_L += A_L \times B_{L,s}$ 
```

```
    else
```

```
        Compute  $C_L += A' \times B_{L,s}$ 
```

```
}
```



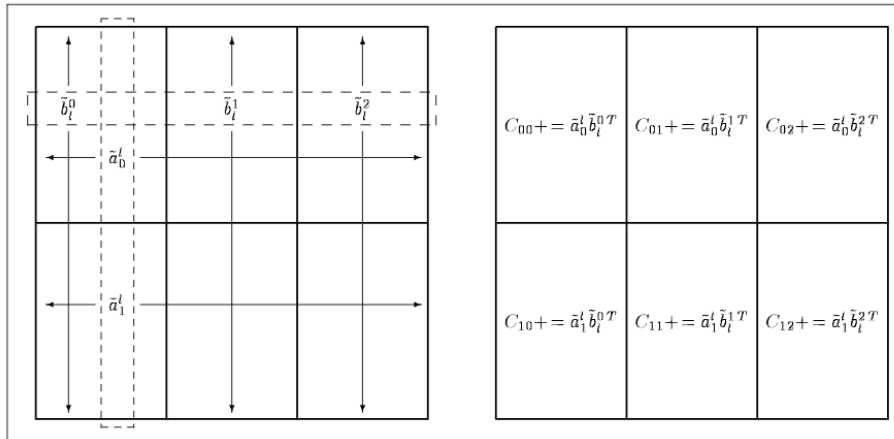
# [Options] Improvements of Memory Reduced Version



Followings are options (NOT mandatory) in assignments [M2]

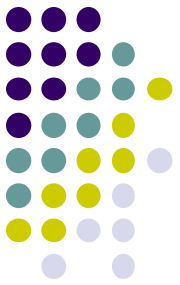
1. To use SUMMA: scalable universal matrix multiplication algorithm

- See <http://www.netlib.org/lapack/lawnspdf/lawn96.pdf>
- Replica is eliminated, and matrices are divided in 2D



2. To use **collective communications** (explained hereafter)

# Peer-to-peer Communications vs Collective Communications



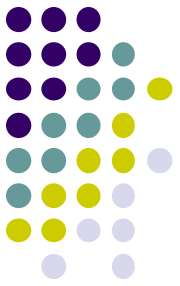
- Communications we have learned are called **peer-to-peer communications**
- A process sends a message. A process receives it



※ `MPI_Irecv`, `MPI_Isend` are also peer-to-peer communications

	Blocking	Non-Blocking
Peer-to-Peer	<code>MPI_Send</code> , <code>MPI_Recv...</code>	<code>MPI_Isend</code> , <code>MPI_Irecv...</code>
Collective	<code>MPI_Bcast</code> , <code>MPI_Reduce...</code>	( <code>MPI_Ibcast</code> , <code>MPI_Ireduce...</code> )

# Collective Communications (Group Communications)



- **Collective communications** involves many processes
  - MPI provides several collective communication patterns
    - Bcast, Reduce, Gather, Scatter, Barrier...
  - All processes must call the same communication function



→ Something happens for all of them

# One of Collective Communications: Broadcast by MPI\_Bcast

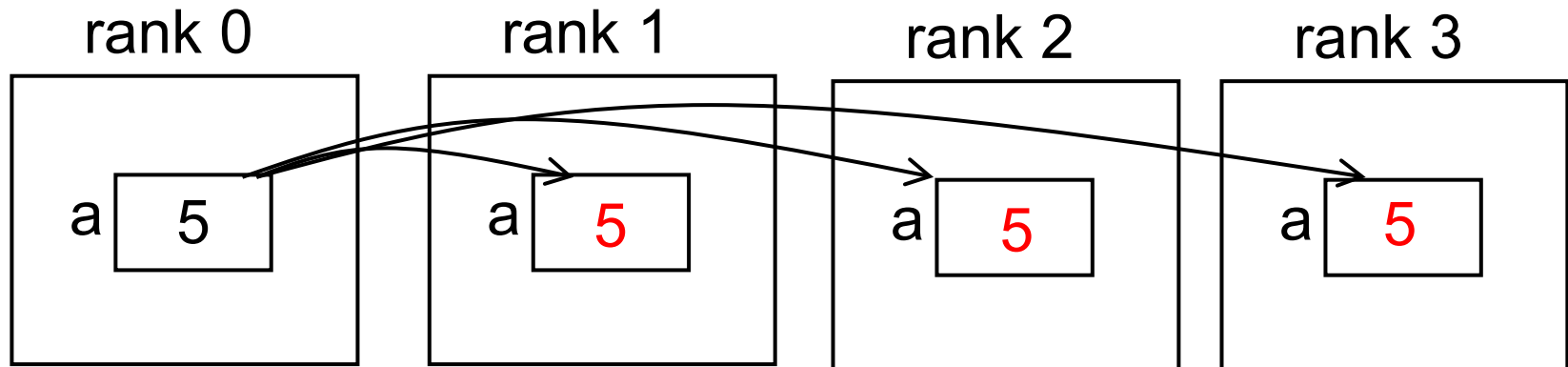


cf) rank 0 has “int a” (called **root process**). We want to send it to all other processes

```
MPI_Bcast(&a, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

- All processes (in the communicator) must call MPI\_Bcast(), including rank 0

→ All other process will receive the value on memory region **a**



✂ What is the role of 1<sup>st</sup> argument?

it is “input” on the root process, and “output” on other processes





- Note: Root wants to send  $A_L$ . Others want to receive data into  $A'$   
 → Different pointers

```

if (I am rank i) copies  $A_L$  to  $A'$ 
MPI_Bcast( $A'$ , ...);

```

```

if (I am rank i) {MPI_Bcast(AL, ...); }
else {MPI_Bcast(A', ...); }

```

17

# Reduction by MPI\_Reduce



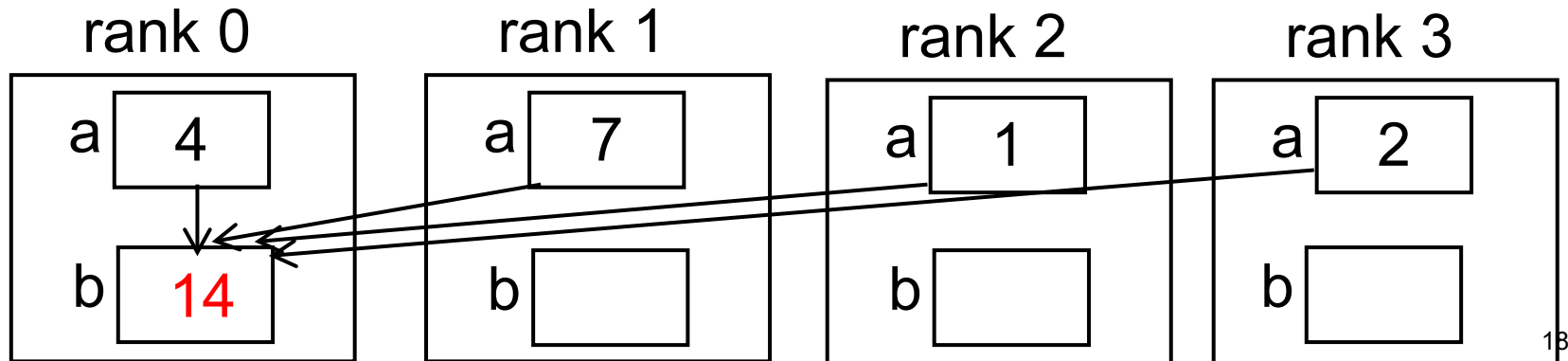
cf) Every process has “int a”. We want the sum of them

```
MPI_Reduce(&a, &b, 1, MPI_INT, MPI_SUM, 0,  
          MPI_COMM_WORLD);
```

*operation*

*root process*

- Every process must call MPI\_Reduce()  
→ The sum is put on b on root process (rank 0 now)
- Operation is one of MPI\_SUM, MPI\_PROD(product), MPI\_MAX, MPI\_MIN, MPI LAND (logical and), etc.

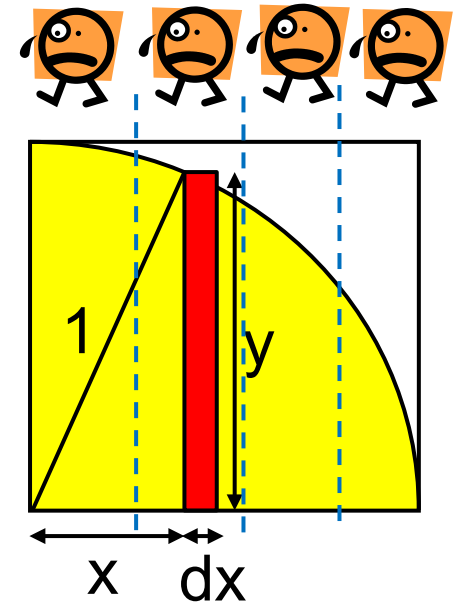


# MPI Version of “pi” Sample

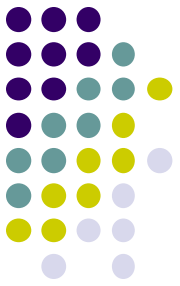


[/gs/bs/tga-ppcomp/24/pi-mpi/](#)

- Execution: `mpiexec -n [#procs] ./pi [n]`
  - n: Number of division
  - Cf) `./pi 100000000`
- We divide  $n$  tasks among processes and calculate total yellow area
  1. Each process calculates local sum
  2. Rank 0 obtains the final sum by **MPI\_Reduce**



$$dx = 1/n$$
$$y = \sqrt{1-x^2}$$



# Using pi-mpi sample

- </gs/bs/tga-ppcomp/24/pi-mpi>

*[make sure that you are at a interactive node (r7i7nX) ]*

`module load intel-mpi` *[Do once after login]*

`cd ~/ppc24` *[In web-only route]*

`cp -r /gs/bs/tga-ppcomp/24/pi-mpi .`

`cd pi-mpi`

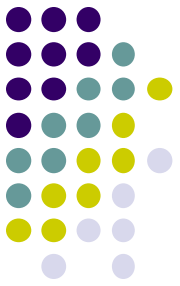
`make`

*[An executable file “pi” is created]*

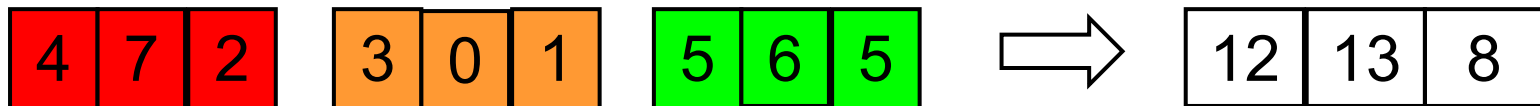
`mpiexec -n 4 ./pi 1000000000`

Number of division

# Note: Differences with “omp for reduction” in OpenMP



- Syntaxes are completely different
- Computations are also different
  - `#pragma omp for reduction(...)` in OpenMP
    - Do “`sum += a[i]`” in parallel for loop with `reduction(+:sum)`
- `MPI_Reduce(...)` in MPI
  - If each input is an array, output is also an array
  - Operations are done for each index



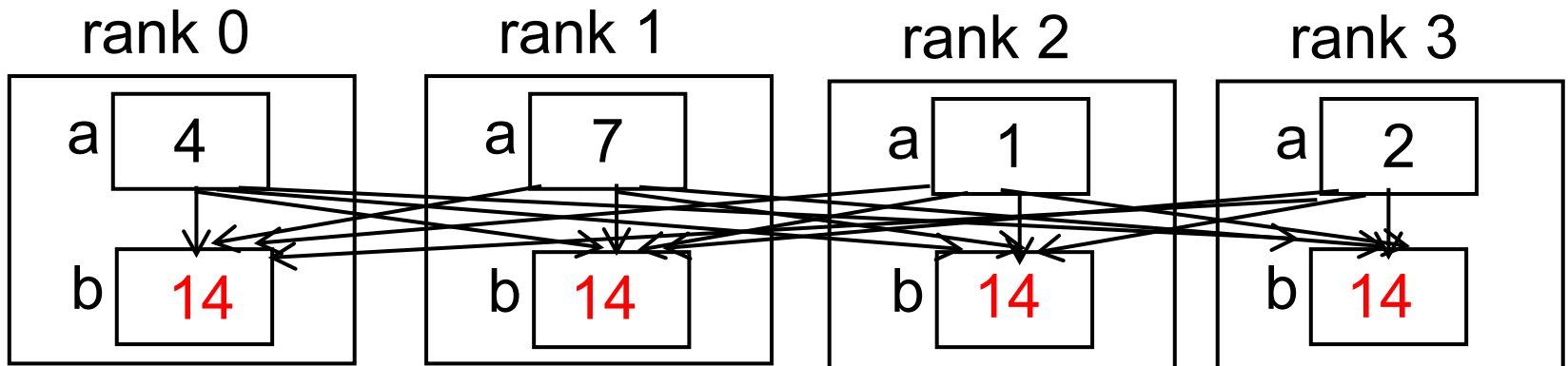


# MPI\_Allreduce

- Allreduce = Reduction + Bcast

```
MPI_Allreduce(&a, &b, 1, MPI_INT, MPI_SUM,  
             MPI_COMM_WORLD);
```

- The sum is put on **b** on all processes



Important communication pattern for distributed deep learning → Try Google “allreduce deep learning”



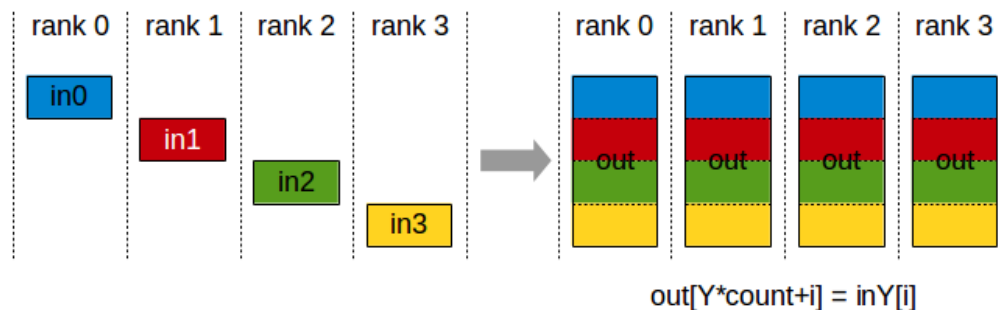
# MPI\_Barrier

- **Barrier synchronization:** processes are stopped until all processes reach the point  
`MPI_Barrier(MPI_COMM_WORLD);`
  - Used in sample programs, to measure execution time more precisely

# Other Collective Communications

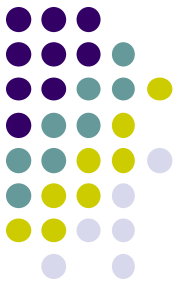


- **MPI\_Scatter**
  - An array on a process is “scattered” to all processes
  - cf) Process 0 has an array of length 10,000. There are 10 processes. The array is divided to parts of length 1,000 and scattered
- **MPI\_Gather**
  - Data on all processes are “gathered” to the root process.
  - Contrary to MPI\_Scatter
- **MPI\_Allgather**
  - Similar to MPI\_Gather. Gathered data are put on all processes



*From NCCL manual at  
[docs.nvidia.com](https://docs.nvidia.com)*

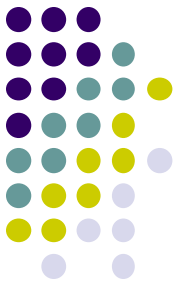




# Performance of Collective Communication



# “Do I Really Need to Learn New Functions?”

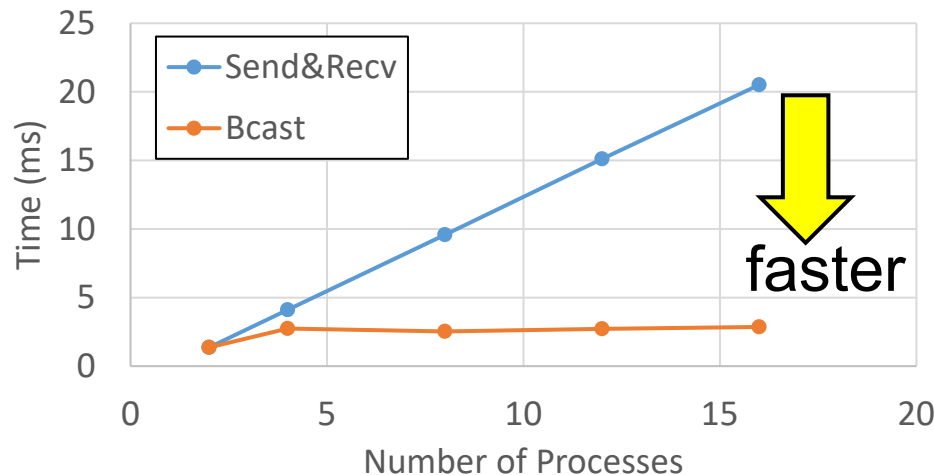


- You can still use MPI\_Send/MPI\_Recv multiple times, but **collective functions are often faster**

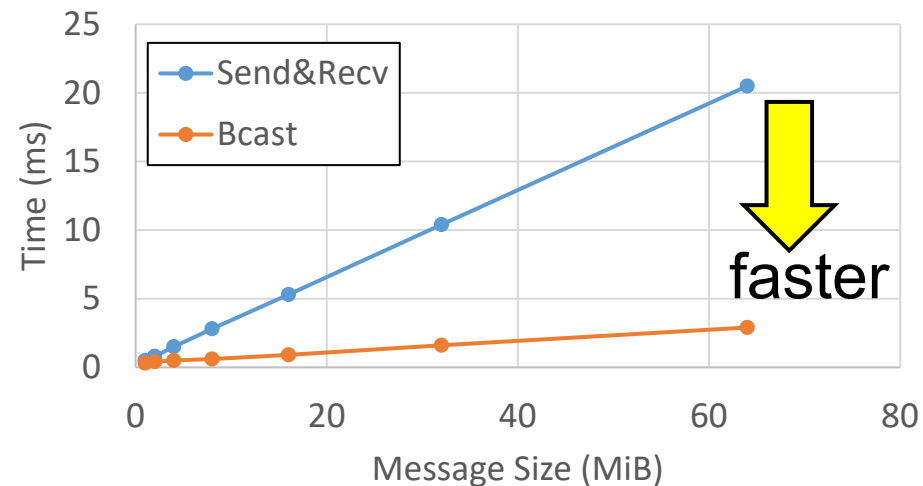
On TSUBAME4  
1 proc / node

In the graph, rank 0 called MPI\_Send for p-1 times to other processes

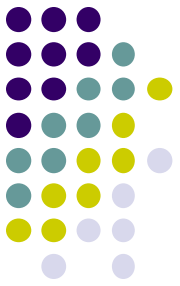
64MB message



16 Processes



- MPI\_Bcast are faster, especially when p is larger !
- The reason is MPI uses “scalable” communication algorithms:  
cf) <http://www.mcs.anl.gov/~thakur/papers/mpi-coll.pdf>



# FYI: Measurement Method

- Measurement in the previous page was done with [/gs/bs/tga-ppcomp/24/mpibcast/](#)
- intel-mpi is used
- NOTE: job\*.sh in this directory need to consume TSUBAME points
  - job\*.sh use > 2nodes

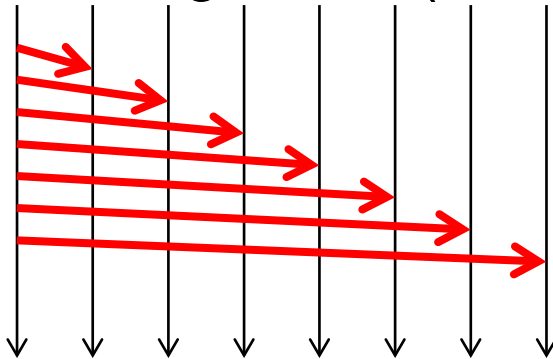
# Why are Collective Communications Fast?



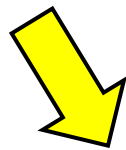
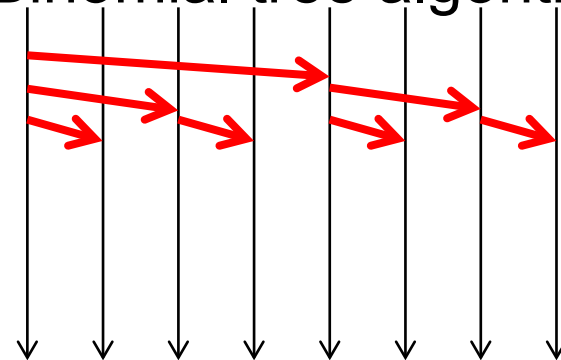
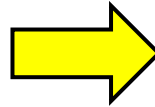
- Since MPI library uses *scalable* communication algorithms

- Case of **broadcast**:

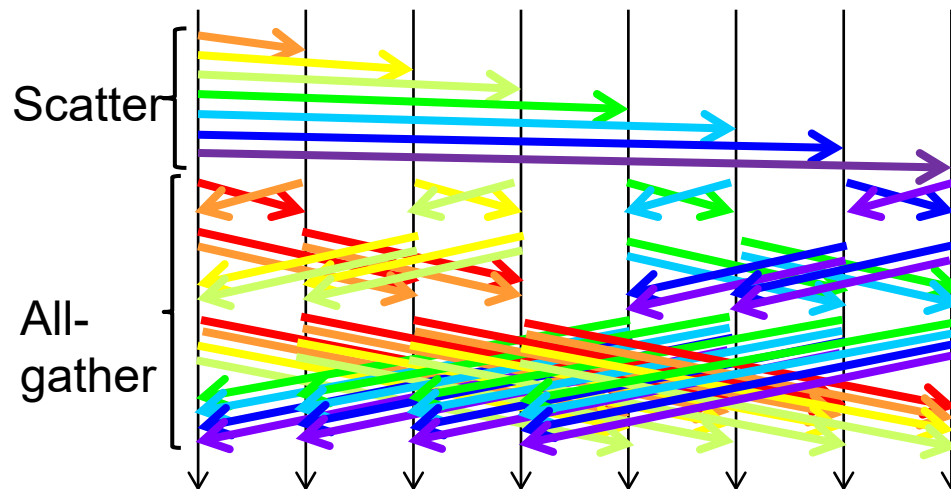
Flat tree algorithm (slow)

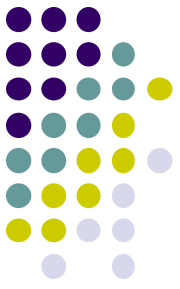


Binomial tree algorithm



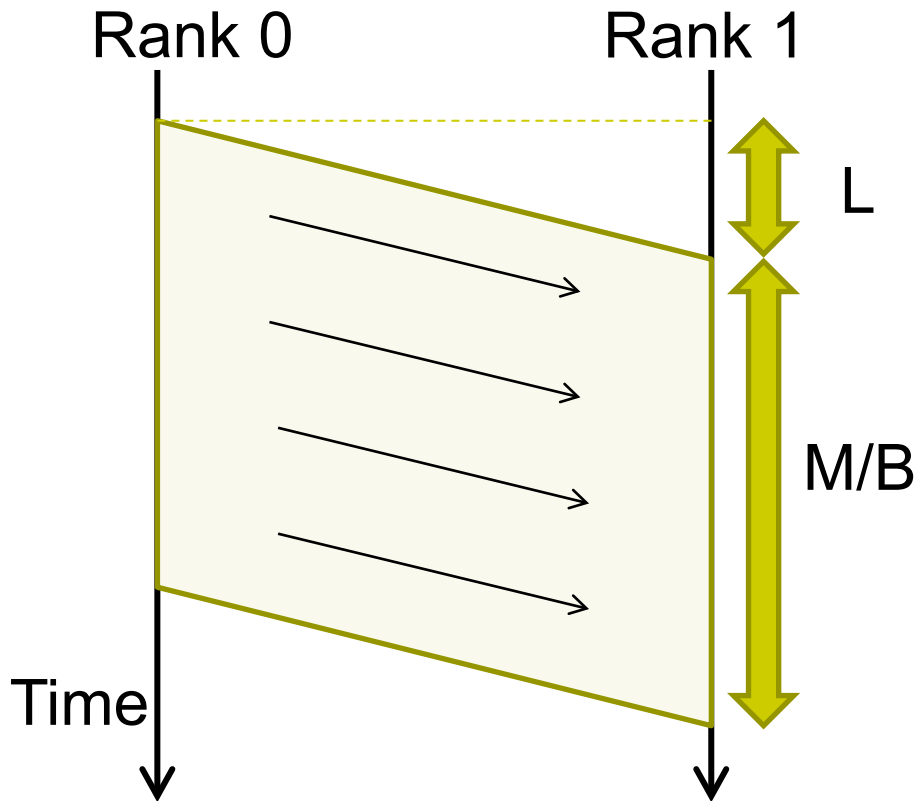
Scatter&Allgather  
algorithm





# Model of Communication Time

Illustration of peer-to-peer communication of data size  $M$



$$T = M / B + L$$

$T$ : Communication time

$M$ : Data size

$B$ : Bandwidth

$L$ : Network latency

※ Be aware of difference between “Byte” and “bit”: 1Byte=8bit

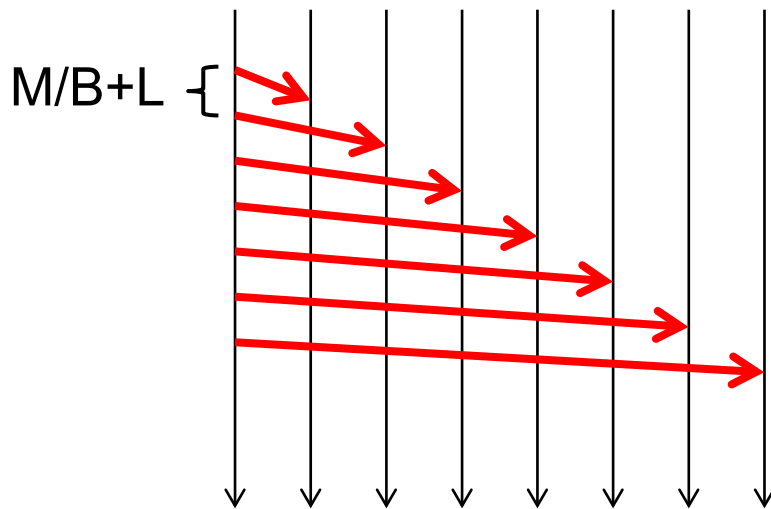
※ Actually it is more complex for process's place, effects of network topology, congestion, packet size...

# Cost Model of Broadcast Algorithms



- Case of “broadcast” of size  $M$  data
  - $p$ : number of processes,  $B$ : network bandwidth,  $L$ : network latency

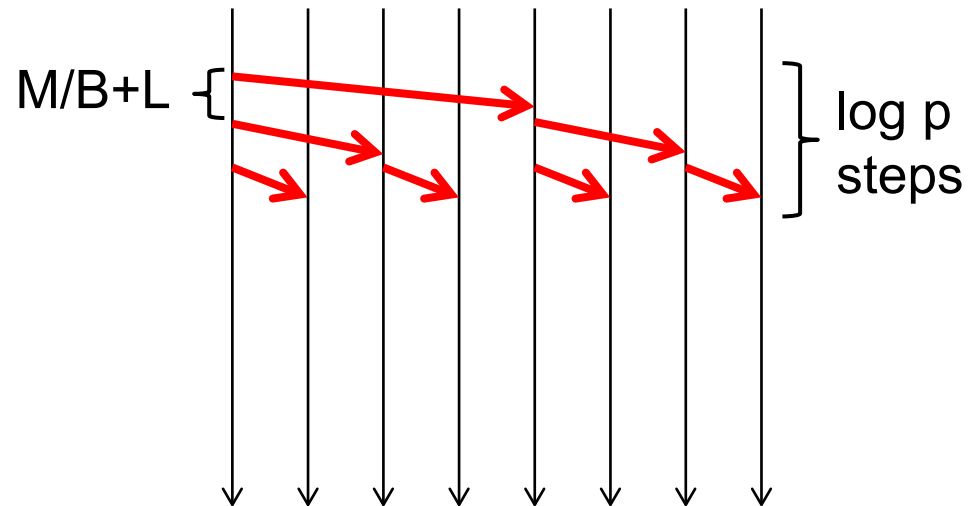
Flat tree algorithm



$$p(M/B + L)$$

→ *Slow*

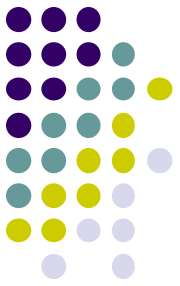
Binomial tree algorithm



$$(\log p)(M/B + L)$$

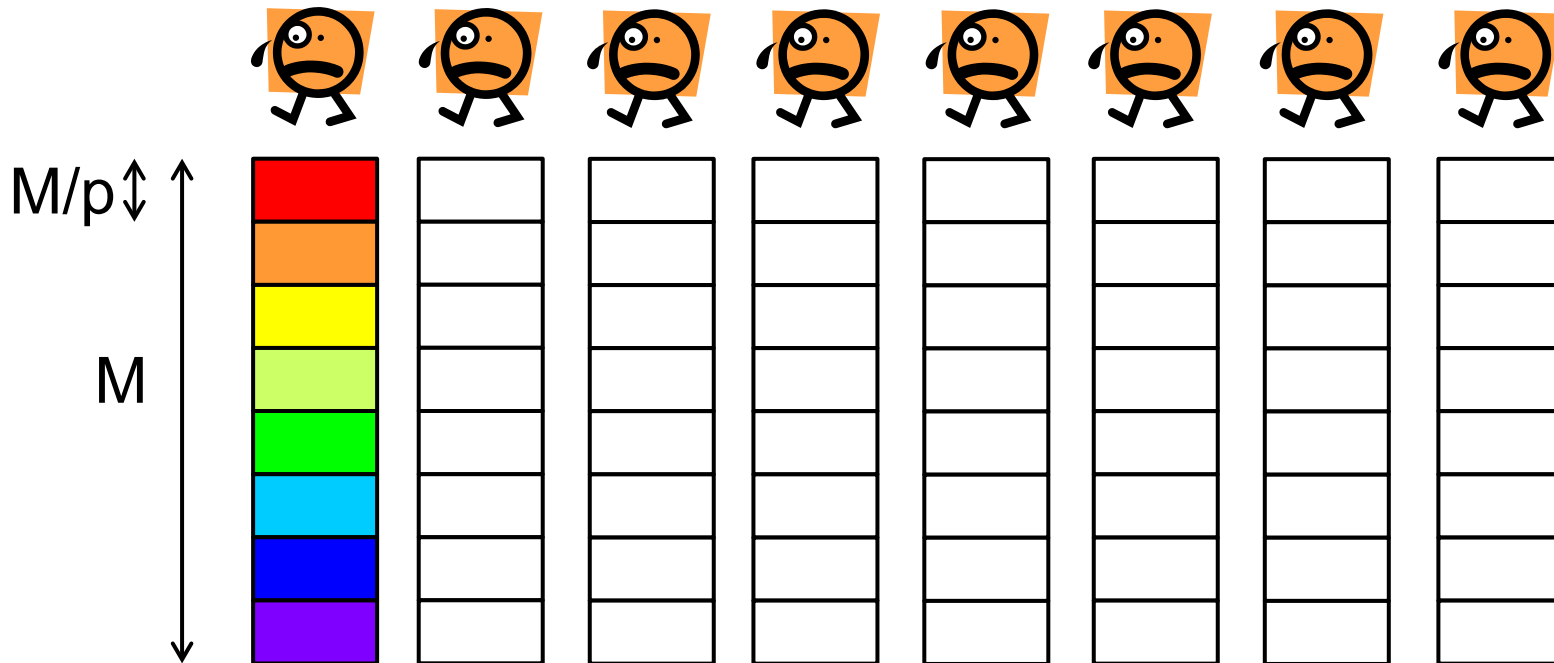
※ difference between  $p-1$  and  $p$  is ignored

# Broadcast by Scatter&Allgather Algorithm (1)



- (1) The root process divide the message into  $p$  parts
- (2) Scatter
- (3) Allgather

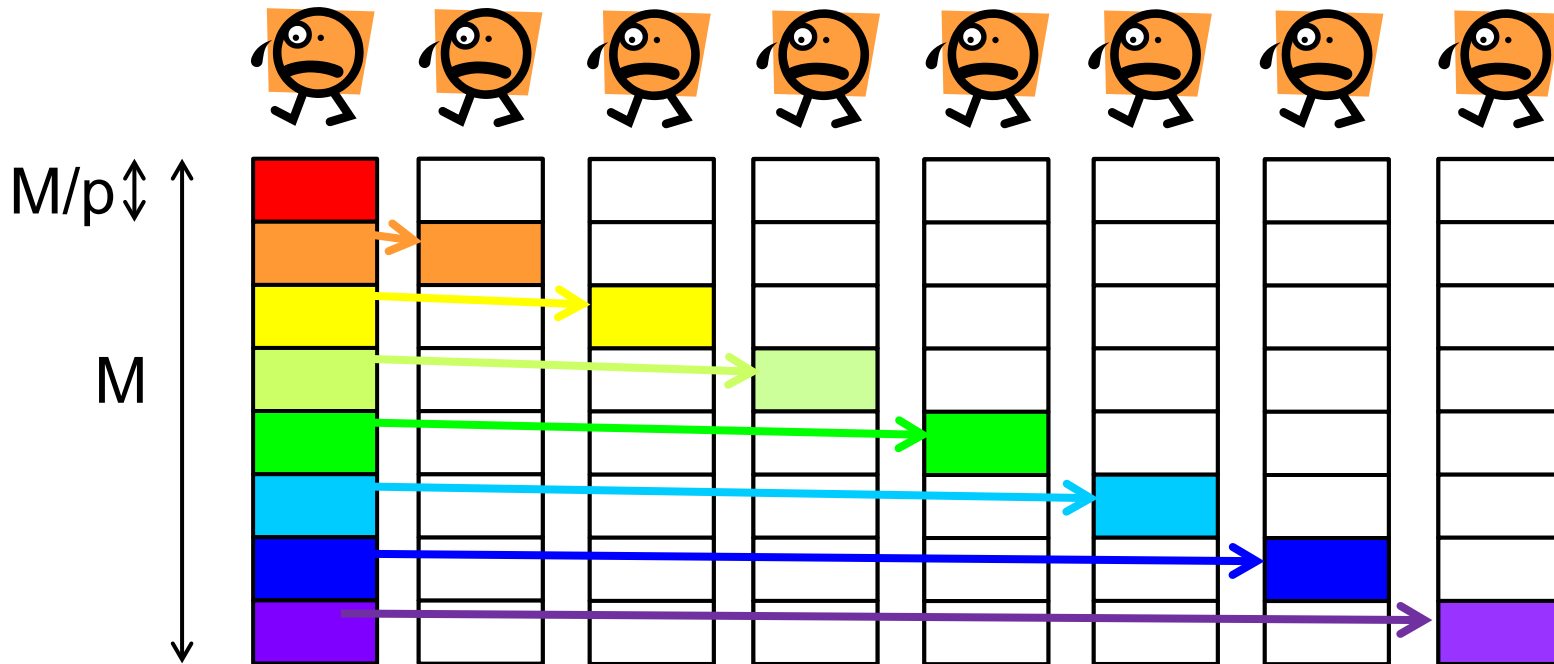
R. Thakur and W. Gropp. Improving the performance of collective operations in mpich. EuroPVM/MPI conference, 2003.



# Scatter&Allgather Algorithm (2)

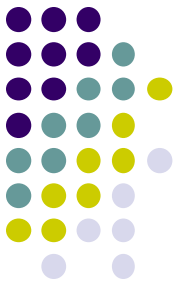


- (1) The root process divide the message into  $p$  parts
- (2) Scatter:  $i$ -th part goes to process  $i$
- (3) Allgather

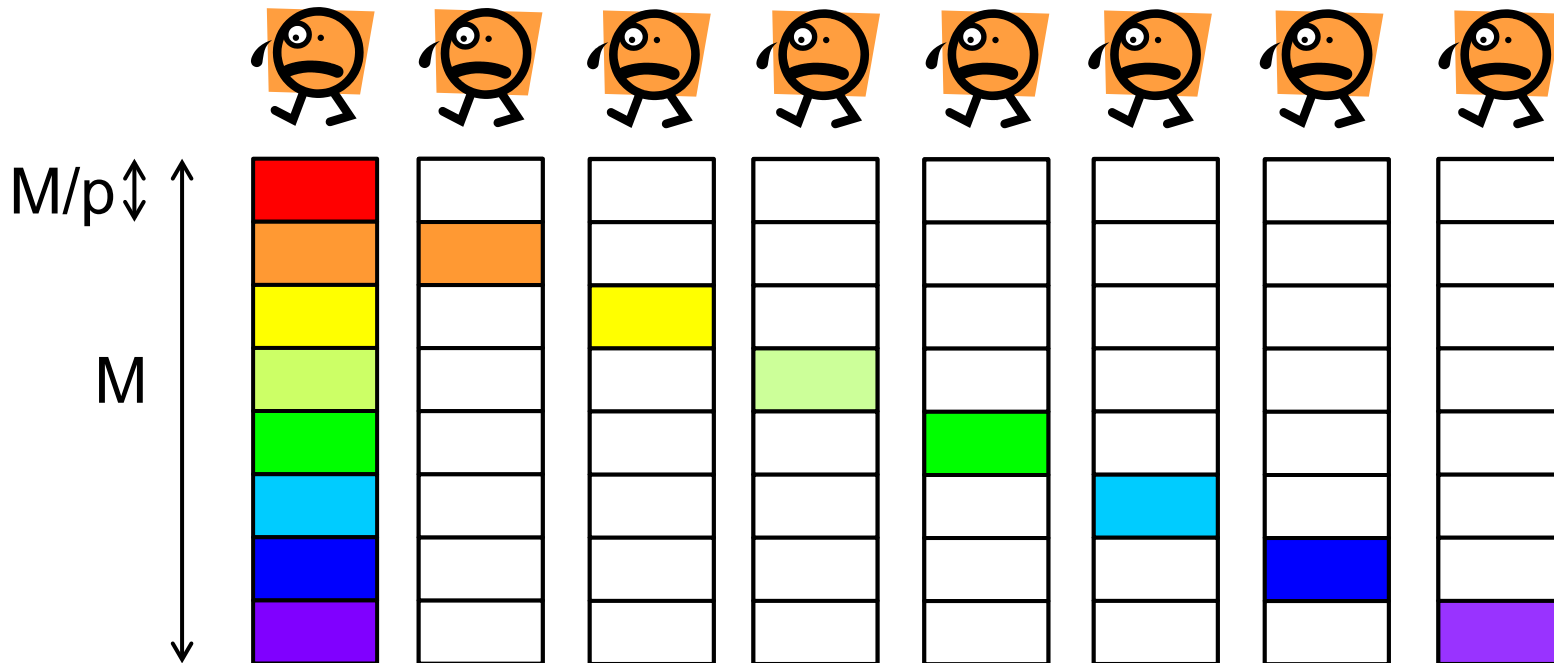




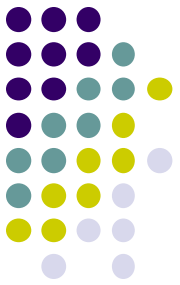
# Scatter&Allgather Algorithm (3)



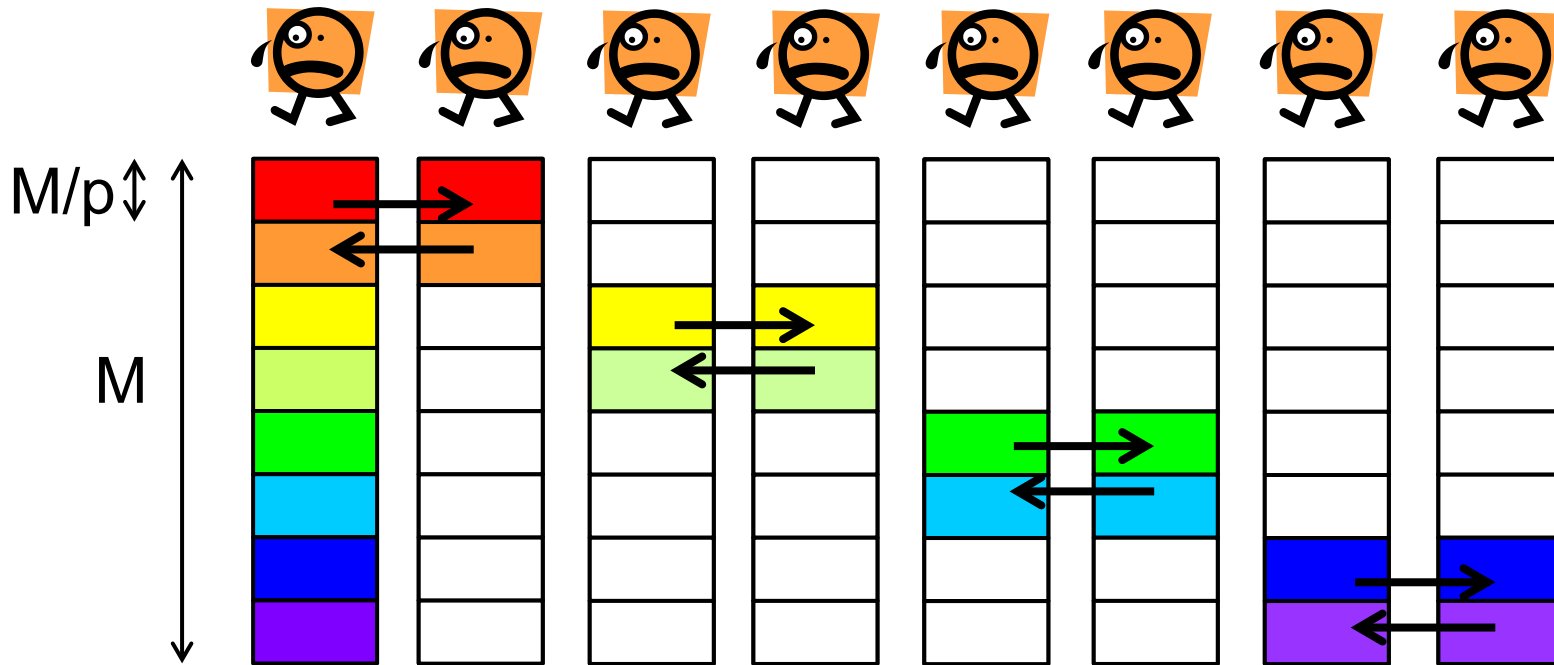
- (1) The root process divide the message into  $p$  parts
- (2) Scatter
- (3) Allgather in  $\log p$  steps
  - If  $p=8$ , we use 3 steps



# Scatter&Allgather Algorithm (3)



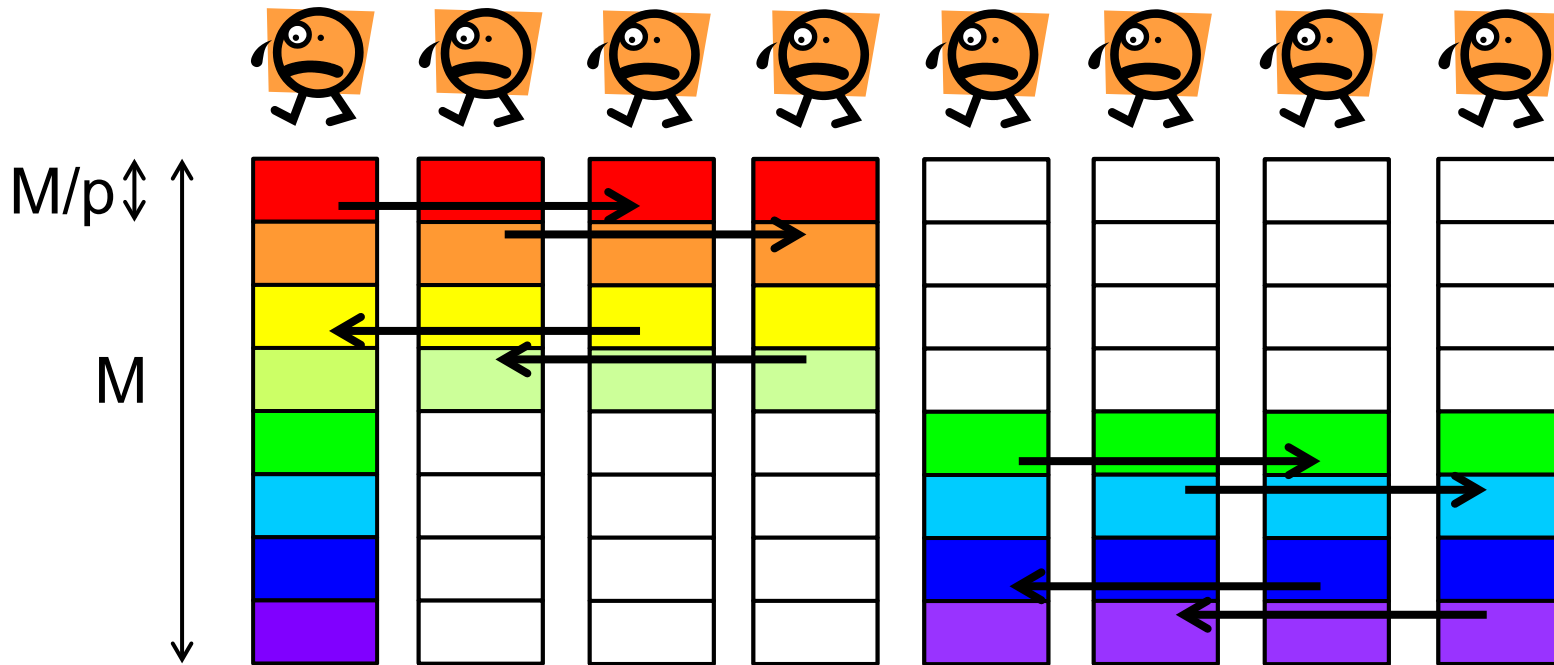
- (1) The root process divide the message into  $p$  parts
- (2) Scatter
- (3) Allgather in  $\log p$  steps
  - If  $p=8$ , we use 3 steps



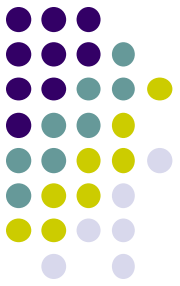
# Scatter&Allgather Algorithm (3)



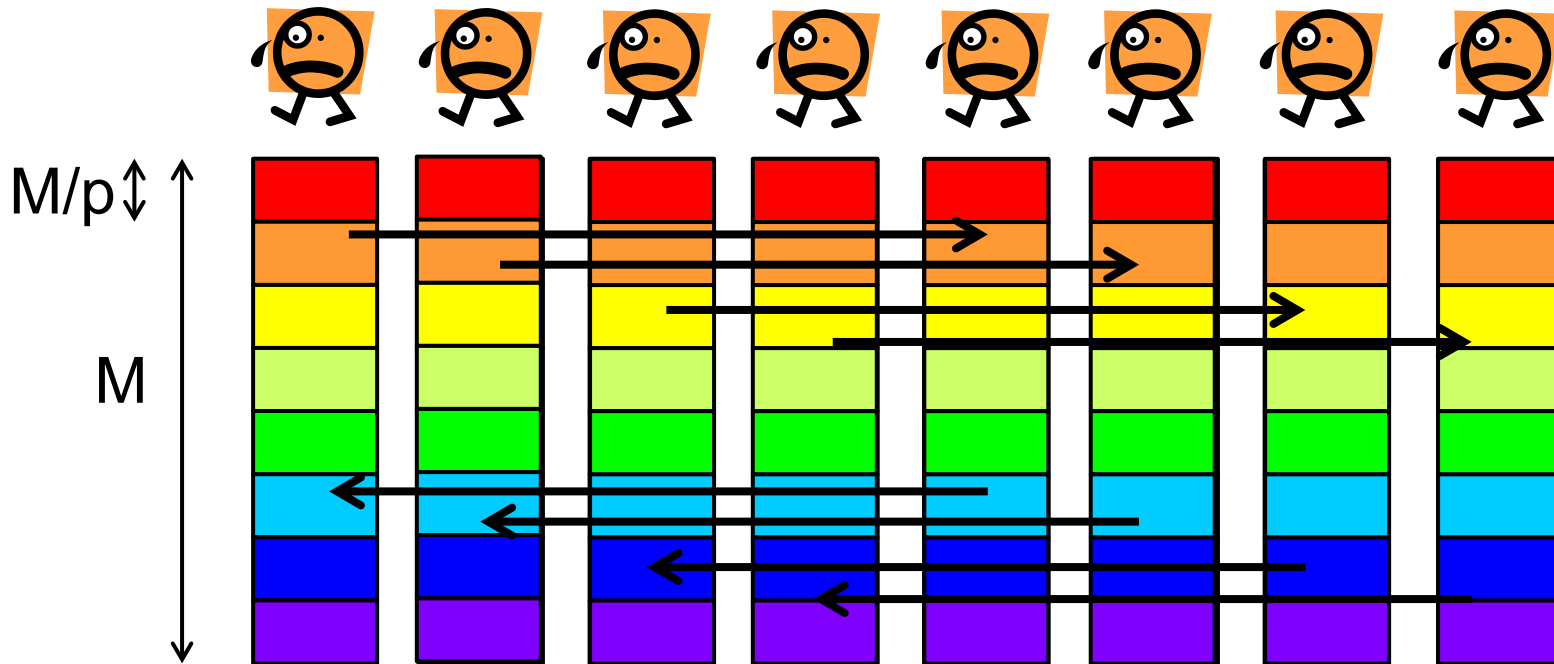
- (1) The root process divide the message into  $p$  parts
- (2) Scatter
- (3) Allgather in  $\log p$  steps
  - If  $p=8$ , we use 3 steps



# Scatter&Allgather Algorithm (3)



- (1) The root process divide the message into  $p$  parts
- (2) Scatter
- (3) Allgather in  $\log p$  steps
  - If  $p=8$ , we use 3 steps

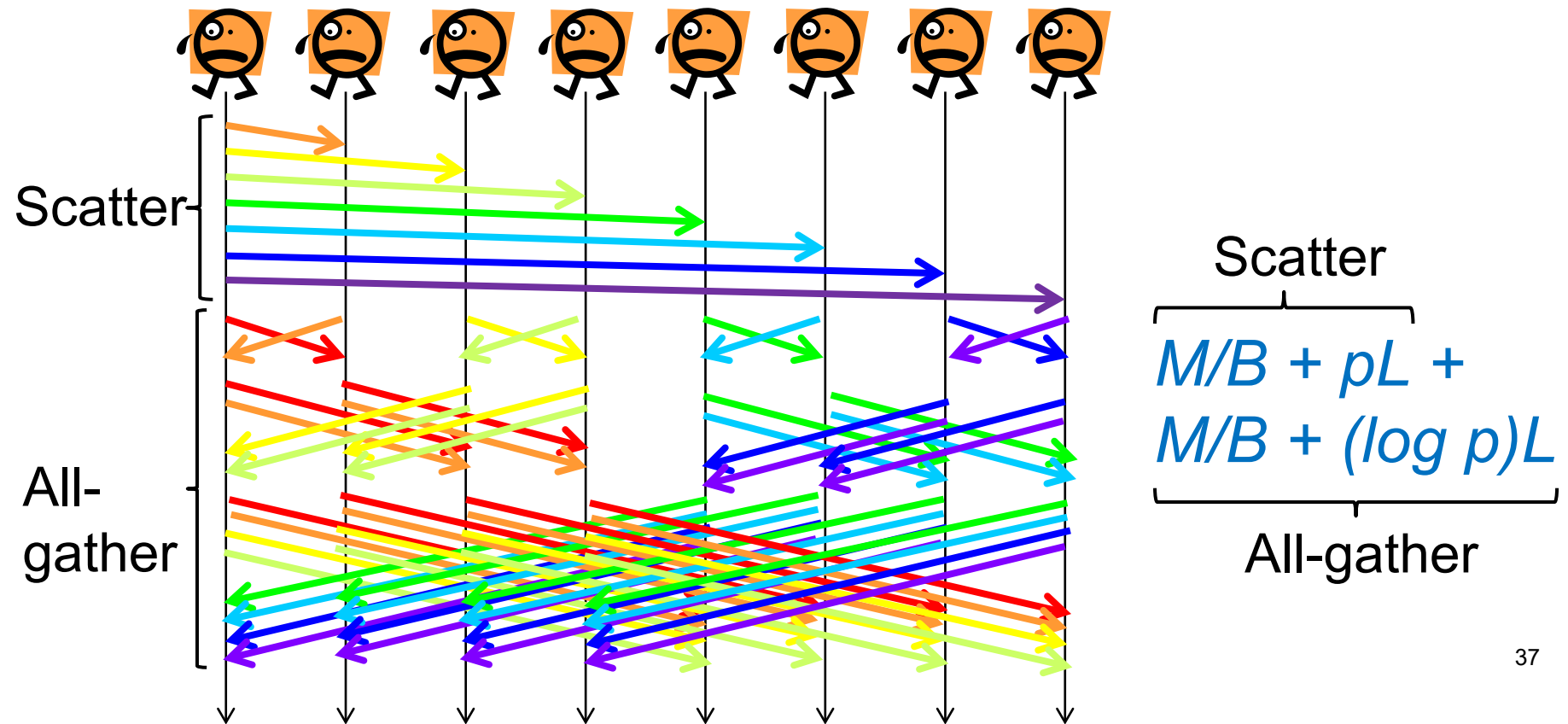


# Cost of Scatter&Allgather Algorithm



- Scatter&Allgather algorithm

- (1) The root process divide the message into  $p$  parts
- (2) Scatter
- (3) Allgather



# Comparison of Broadcast Algorithms



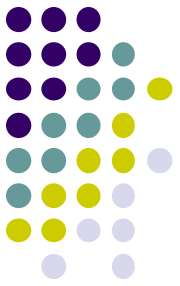
- Consider two extreme cases
  - If  $M$  is sufficiently large:  $M/B+L \rightarrow M/B$
  - If  $M$  is close to zero:  $M/B+L \rightarrow L$

	Flat Tree	Binomial Tree	Scatter&Allgather
General Cost	$p(M/B+L)$	$(\log p) (M/B+L)$	$2M/B + (p + \log p)L$
Cost with very large $M$ ( $L$ is ignored)	$p M/B$	$(\log p) M/B$	$2 M/B$ $\rightarrow$ Fastest
Cost with very small $M$ ( $M$ is ignored)	$p L$	$(\log p) L$ $\rightarrow$ Fastest	$(p + \log p) L$

Many MPI libraries implement multiple algorithms

They switch them automatically according to message size  $M$  😊

# Assignments in MPI Part (Abstract)



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

Due date: **June 14 (Thursday)**

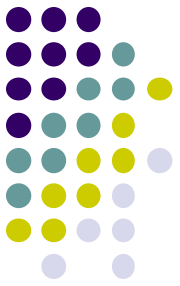
[M1] Parallelize “diffusion” sample program by MPI.

- Be careful for deadlock

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

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

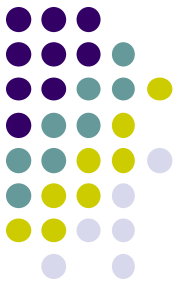
For more detail, please see 3-1 slides



# Notes in Part 3 Assignments

- It is reported that MPI programs may be sometimes very slow on TSUBAME4 CPUs
  - $\frac{1}{2}$  –  $\frac{1}{4}$  speeds of OpenMP programs
  - Reasons are still unclear, but considered to be related to architecture and multi-processes
- It is ok to report such slow performance





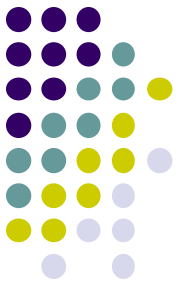
# Next Class

- MPI (4)
  - Other topics about MPI and parallel computing
- Planned schedule
  - June 3: Part 3 (4): (Short) class + TSUBAME4 tour
    - If you come to the class room at 10:45, you can see TSUBAME4
    - Room 202, 2F, G2 building, Suzukake-dai campus

G2 bulding: No 35 in Campus map

<https://www.titech.ac.jp/english/0/maps/suzukakedai>

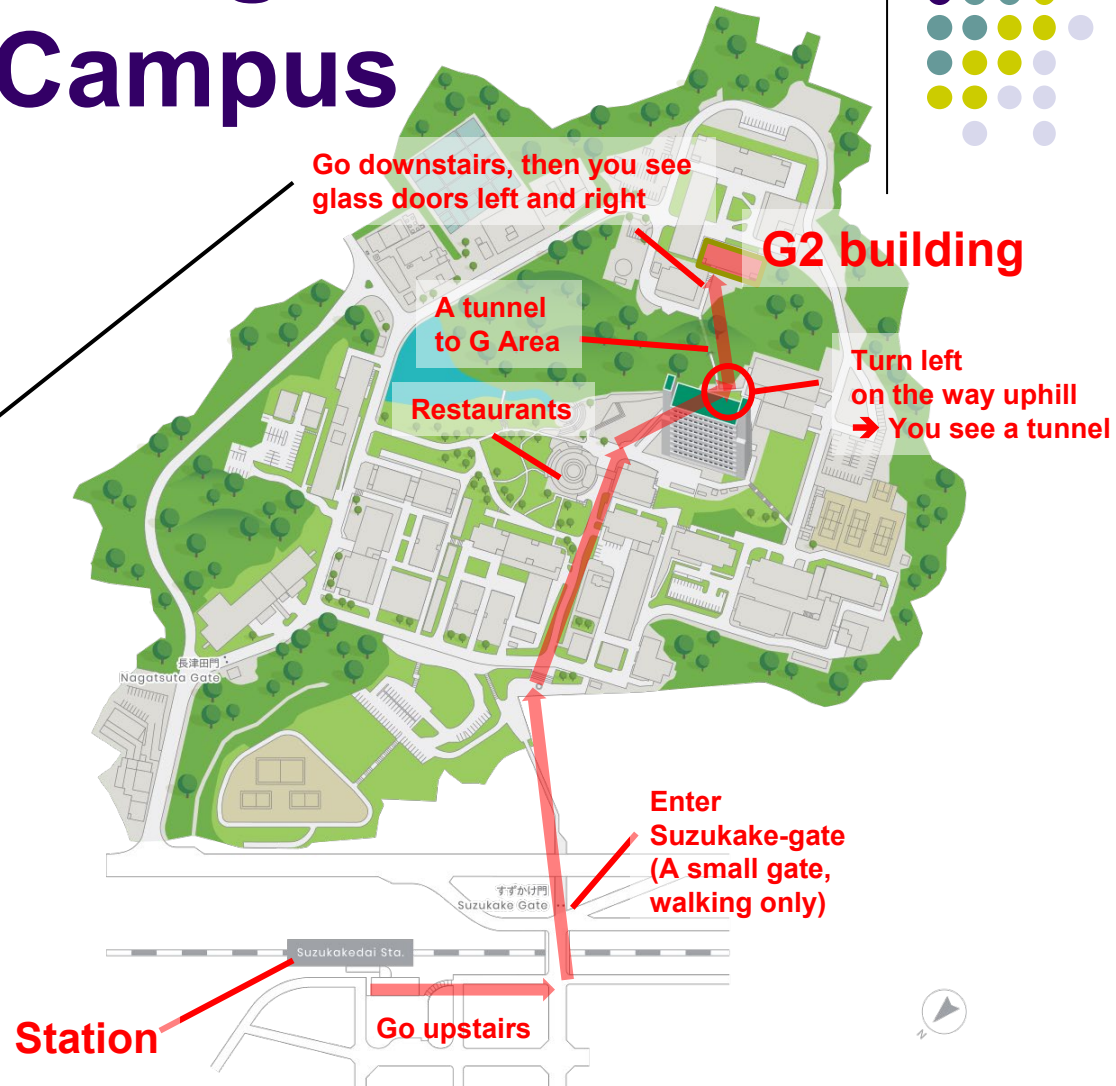
# Way to G2 Building at Suzukakedai Campus



Suzukakedai Campus:

10 minutes walk from  
Suzukakedai Station,  
Denentoshi Line

Enter the right door and  
go up to the 2<sup>nd</sup> floor  
→ Room 202



# すずかけ台G2棟 への道

すずかけ台キャンパス：

田園都市線すずかけ台駅から  
約徒歩10分

202講義室：  
右のドアに入り二階に上る

