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

2025 Class No.14
[MPI Part] (3)
Non-Blocking Communication,
Collective Communication

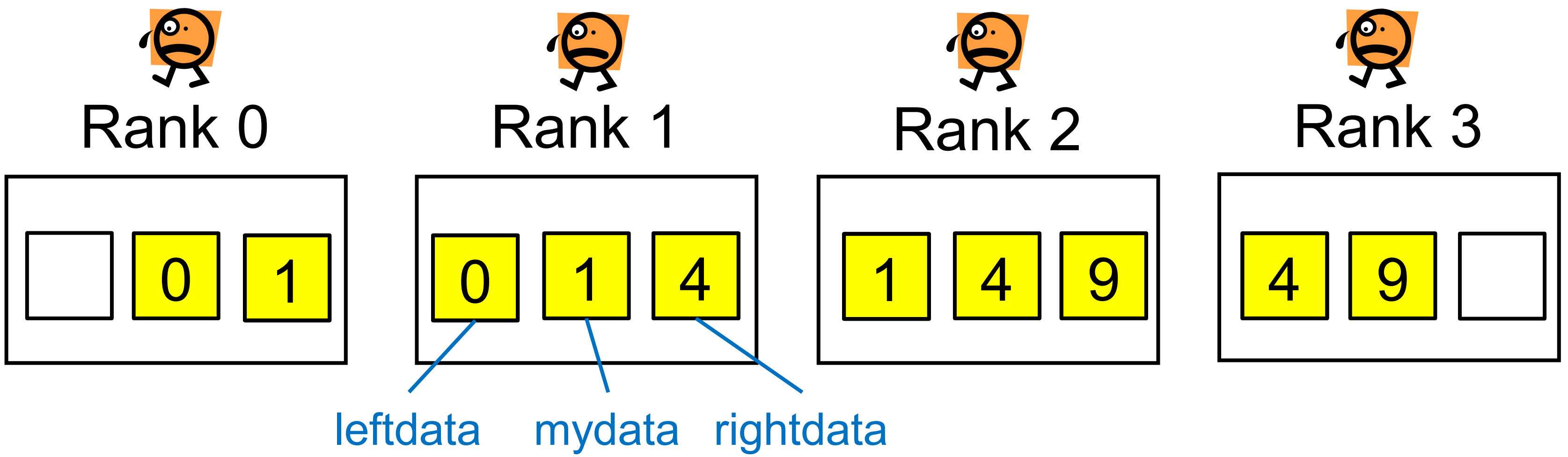
Toshio Endo
endo@scrc.iir.isct.ac.jp

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 ← We are here (3/3)
  - Report (elective)

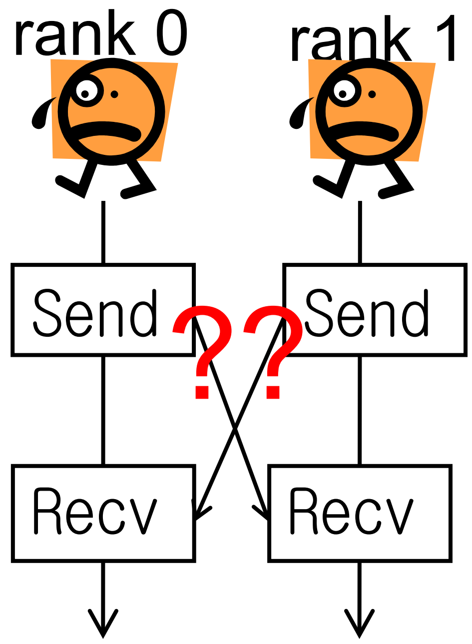
Review: Mutual Communication May Cause Deadlock Problem (related to [M1])

- In neicomm sample, the program does not finish under some conditions → Why?



Deadlock in MPI

- With “neicomm_unsafe()”, it “deadlocks” with 2 processes. Why?



One of reasons is usual MPI_Send and MPI_Recv uses “blocking communication”

- a process waits until “some event”

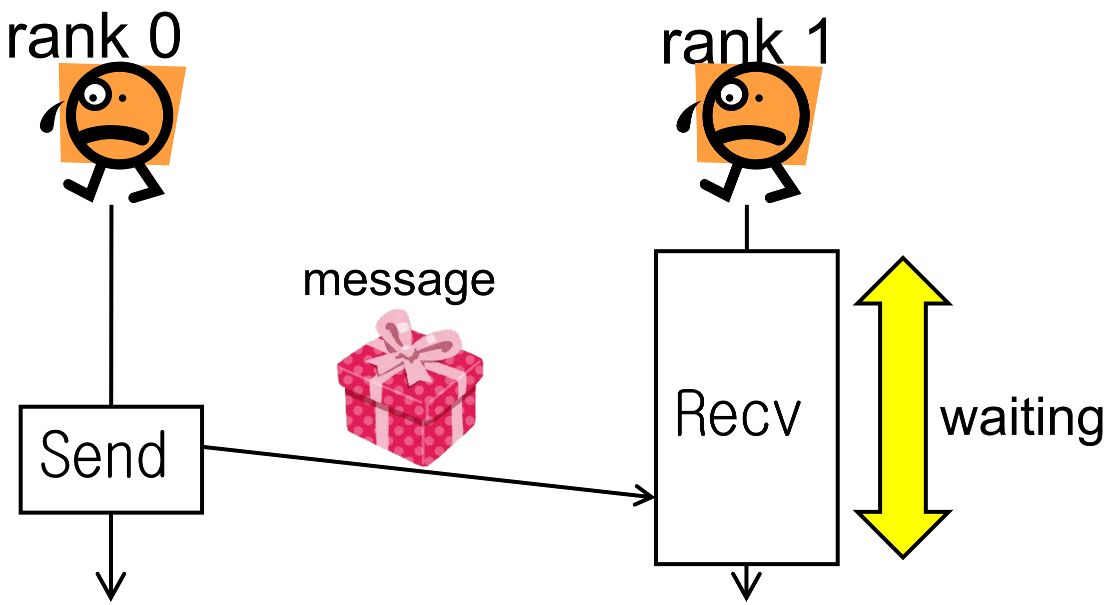
Behavior of MPI_Recv()

Example:

- MPI_Send is called by rank0, and MPI_Recv is called on rank1
    - Processes are running independently

If MPI_Recv is called earlier,

➔ MPI_Recv() waits until the message arrives (blocking)

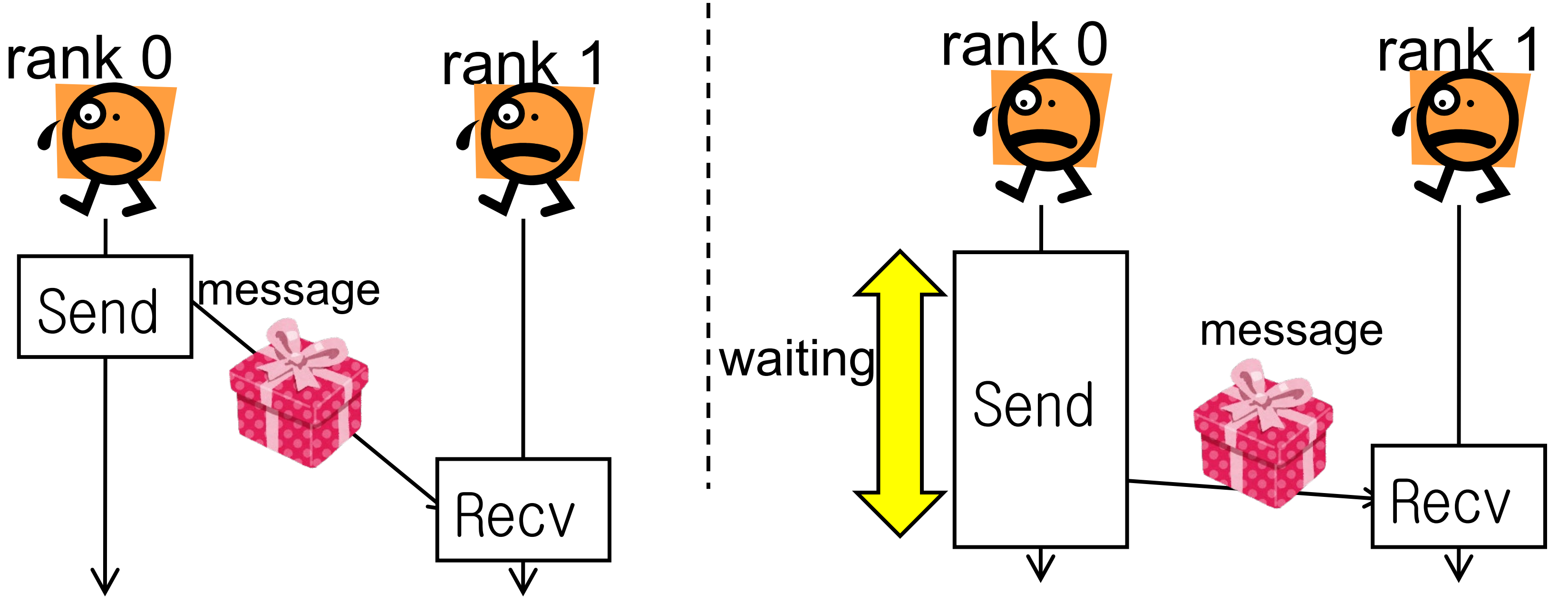


Behavior of MPI_Send()

If MPI_Send is called earlier, there are two possibilities

(case 1) MPI_Send() finishes soon (non-blocking)

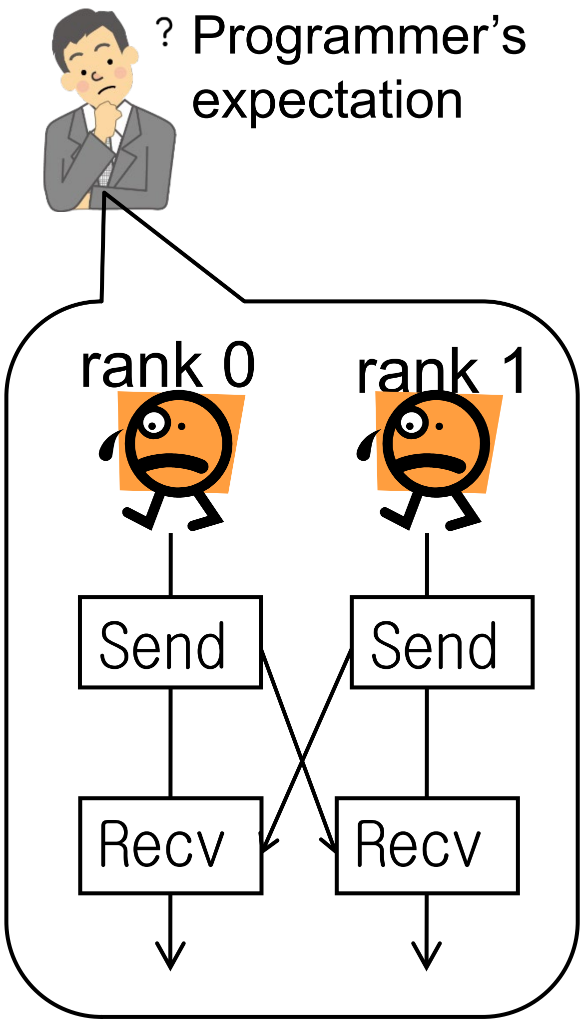
(case 2) MPI_Send() waits until the message arrives to destination (blocking)



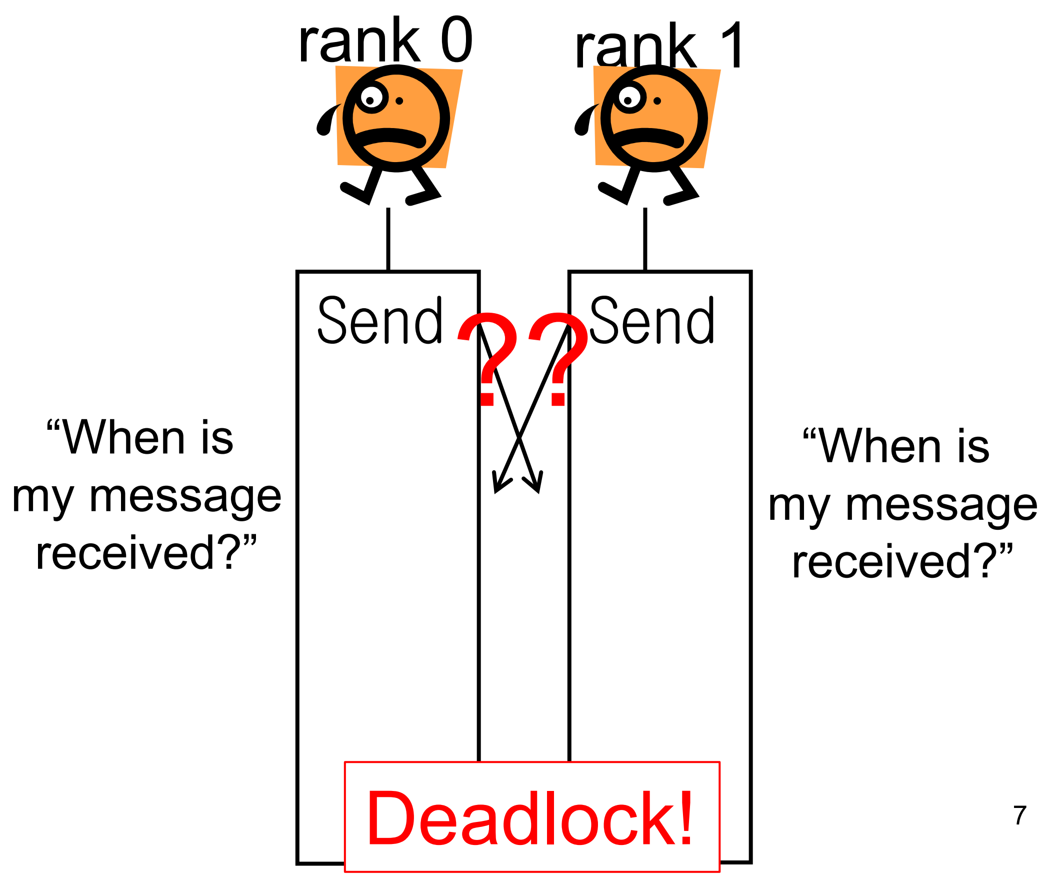
Which occurs?
It depends on MPI library, message size, etc. ➔ Unknown

And Deadlock Happens

? Programmer's expectation

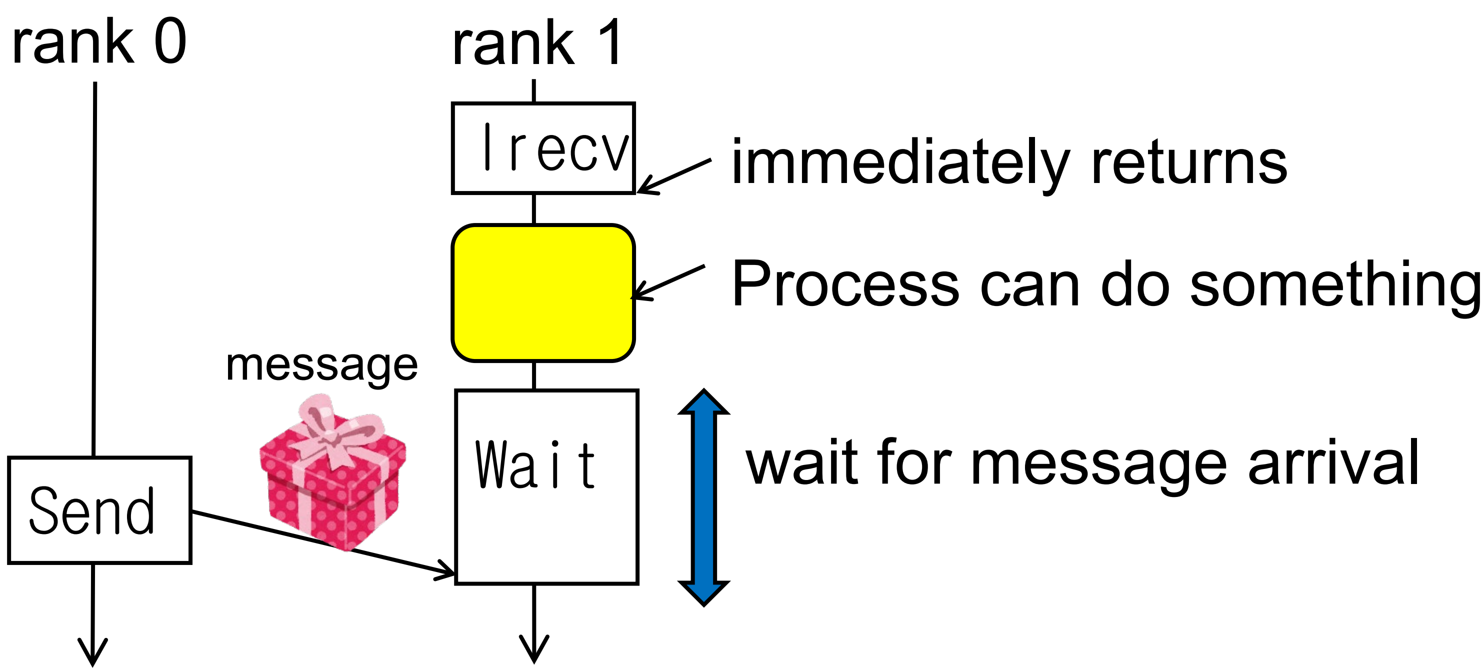


If MPI_Send is blocked until arrival in destination …



Introduction of Non-Blocking Communication

- Non-blocking communication: starts a communication (send or receive), but does not wait for its completion
  - MPI_Recv is blocking communication, since it waits for message arrival
- Program must wait for its completion later: MPI_Wait()



Non-Blocking Receive

```
MPI_Status stat;
MPI_Recv(buf, n, type, src, tag, comm, &stat);
```

```
MPI_Status stat;
MPI_Request req;
MPI_Irecv(buf, n, type, src, tag, comm, &req);←start recv
   :  (Do domething)
MPI_Wait(&req, &stat);    ←wait for completion
```

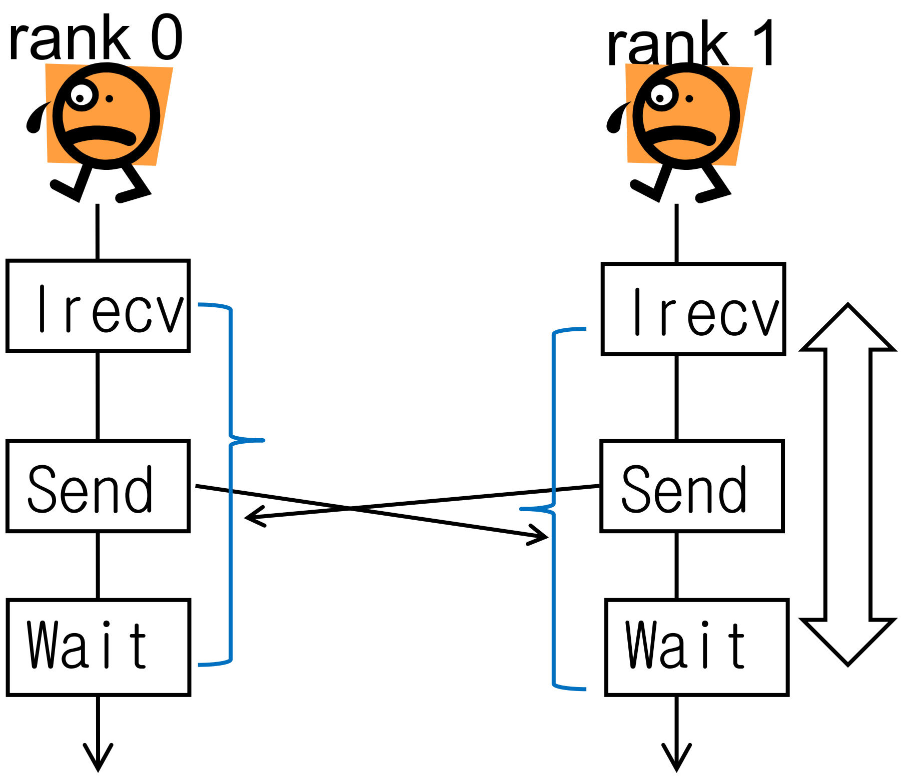
MPI_Irecv: starts receiving, but it returns Immediately

MPI_Wait: wait for message arrival

MPI_Request is like a “ticket” for the communication

Avoiding Deadlock with Non-Blocking Communication

On each process, Recv is divided into Irecv & Wait

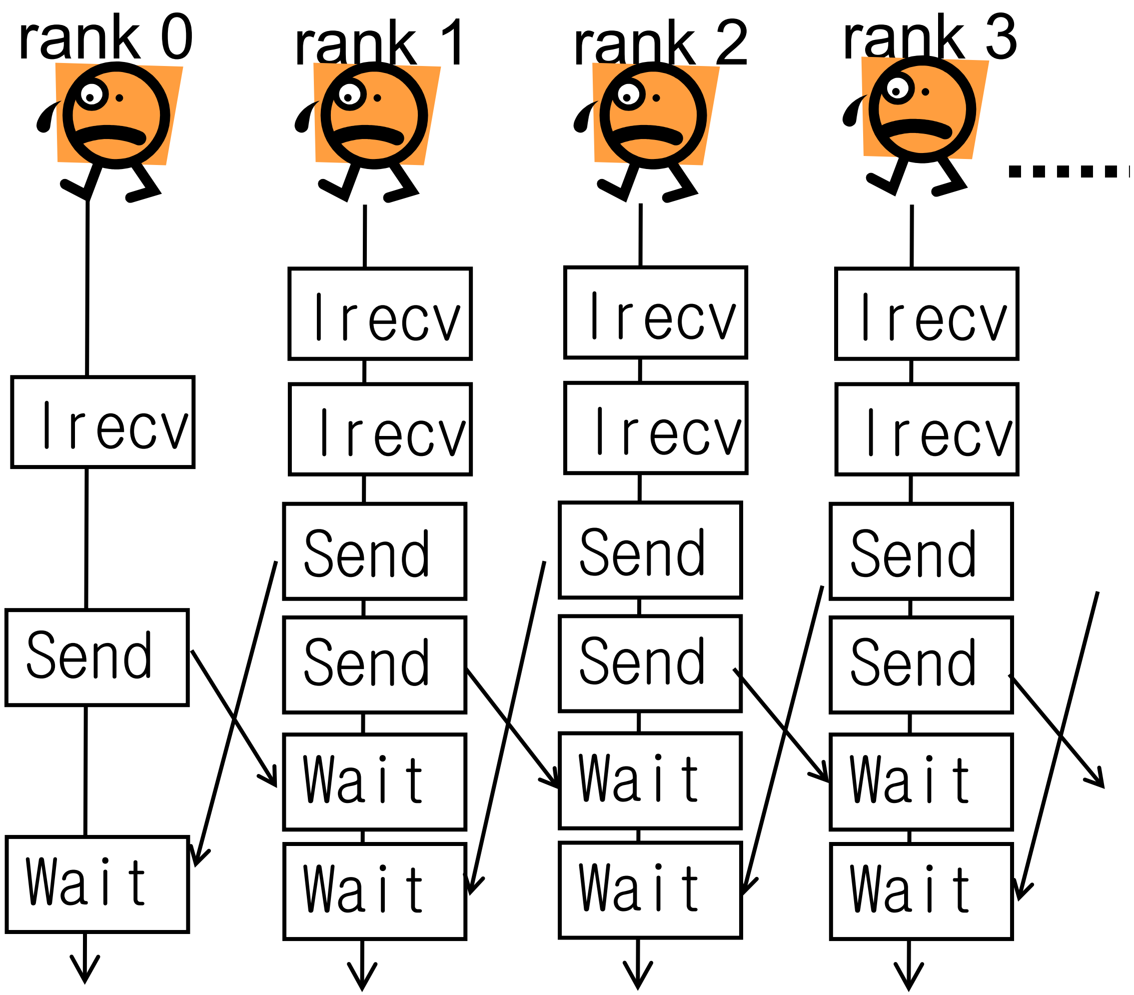


What's difference than before?

A message can be (internally) received during Irecv and Wait
➔ MPI_Send can finish in finite time

Cases for Multiple Processes

- See neicomm_safe() in neicomm.c



Each Irecv has to use distinct MPI_Request

Functions Related to Non-blocking Communication

- MPI_Isend(buf, n, type, dest, tag, comm, &req); ←start send
  - MPI_Isend must be followed by MPI_Wait (or alternatives)
- MPI_Wait(&req, &stat); ←wait for completion of one communication
- MPI_Test(&req, &flag, &stat); ←check completion of one communication
- MPI_Waitall, MPI_Waitany, MPI_Testall, MPI_Testany…

Avoiding Deadlock with Non-Blocking Communication (2)

- The following patterns are also Ok
- Each process does
  - Irecv, Irecv, Send, Send, Wait, Wait
    - Use in neicomm_safe()
  - Isend, Isend, Recv, Recv, Wait, Wait
  - Isend, Isend, Irecv, Irecv, Wait, Wait, Wait, Wait
    - 4 MPI_Request required
  - Irecv, Irecv, Send, Send, Wait, Wait, Wait, Wait
    - 4 MPI_Request required

"Send, Send, Irecv, Irecv, Wait, Wait" is NG. Why?

Next topic: "mm" sample again (related to [M3])

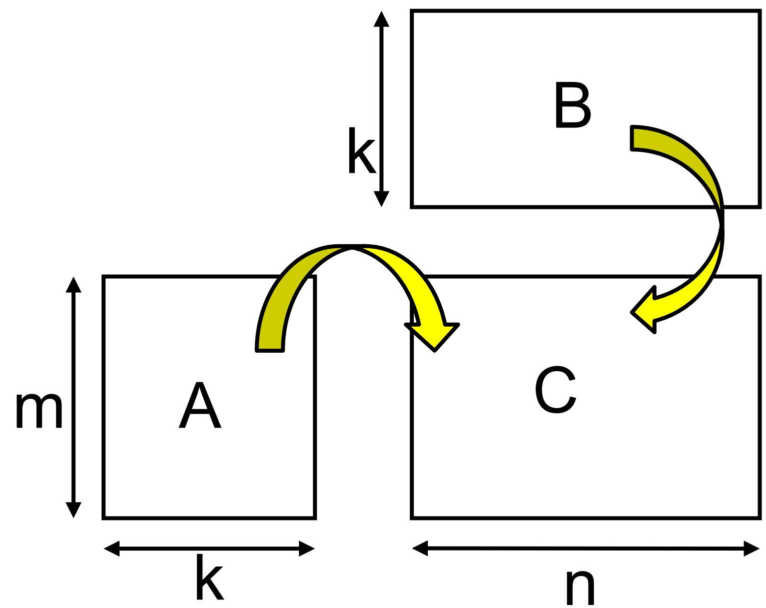
ppcomp-ex/mpi/mm, ppcomp-ex/mpi/mm-comm

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

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

$C \leftarrow A \times B$

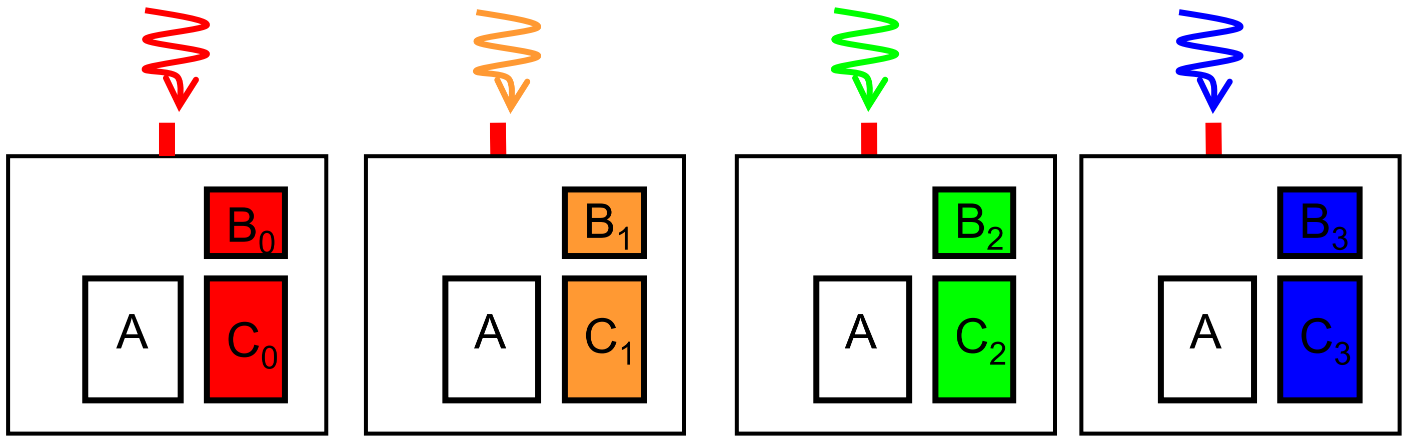
- Algorithm with a triple for loop
- Supports variable matrix size.
  - Each matrix is expressed as a 1D array by *column-major* format



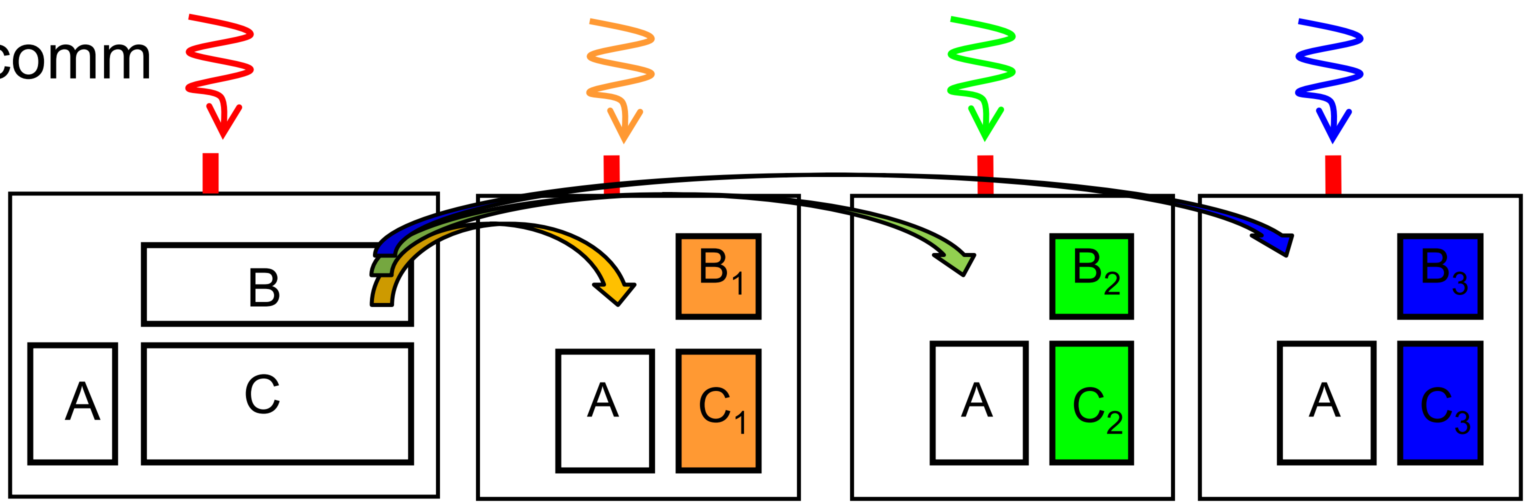
Execution: mpiexec -n [#proc] ./mm [m] [n] [k]

Data Distribution in mm, mm-comm

mm



mm-comm



Introduction of Group Communication

- mm-comm does communication:
  - Matrix A is sent from rank 0 to others
  - Matrix B/C are split and sent from rank 0 to others
  - Partial Cs are sent from other processes to rank 0
- In mm-comm, all are written by combination of MPI_Send/MPI_Recv
  - MPI have specialized functions for above purposes

➔ Collective communication functions

Peer-to-peer Communications vs Collective Communications

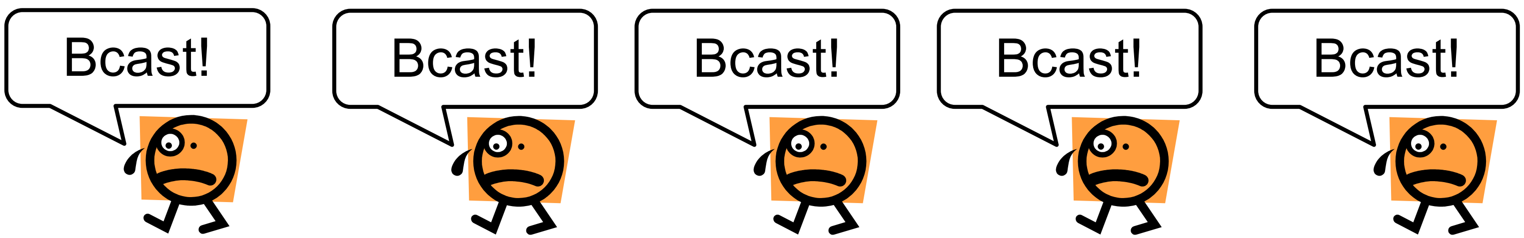
- Communications we have learned are called peer-to-peer communications
- A process sends a message. A process receives it

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

	Blocking	Non-Blocking
Peer-to-Peer	MPI_Send, MPI_Recv…	MPI_Isend, MPI_Irecv…
Collective	MPI_Bcast, MPI_Reduce…	(MPI_Ibcast, MPI_Ireduce…)

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

Several communication patterns:

- MPI_Bcast, MPI_Scatter, MPI_Gather, MPI_Reduce, MPI_Barrier...

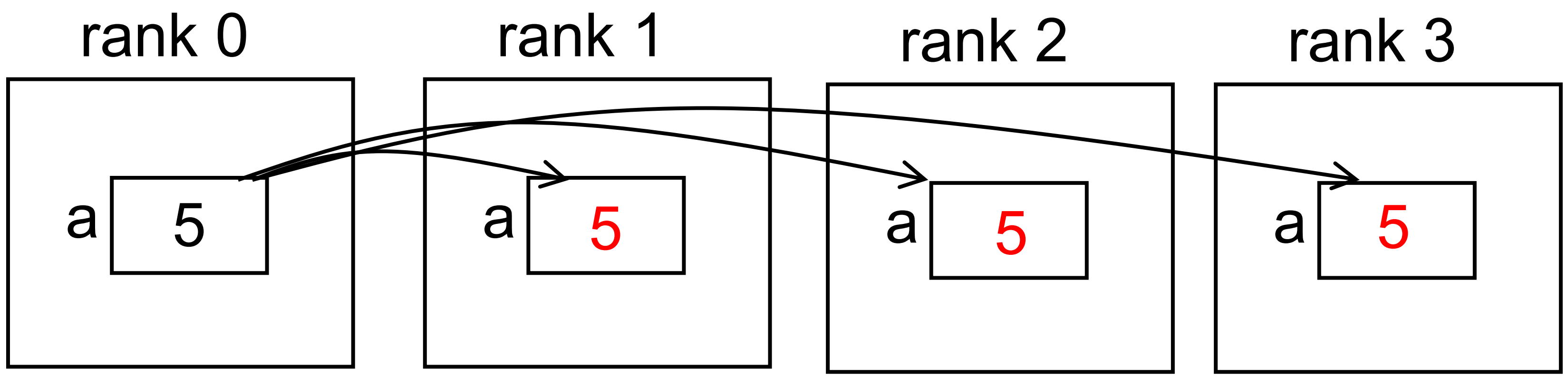
One of Collective Communications: Broadcast by MPI_Bcast

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);
```

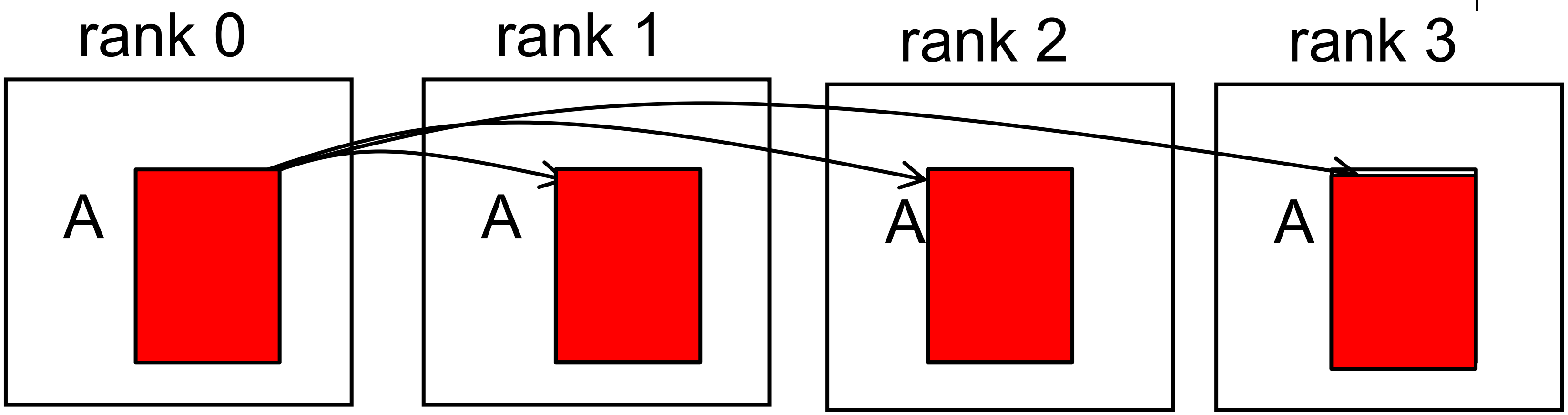
- All processes (in the communicator) must call MPI_Bcast(), including rank 0

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



※ What is the role of 1st argument?
it is “input” on the root process, and “output” on other processes

MPI_Bcast Can Be Used in "Communication of A" in mm-comm [M3]



- Rank 0 has contents of A
- All other processes require *all* contents of A

→ This is "broadcast" pattern. We can use MPI_Bcast instead!
Root 0 becomes rank

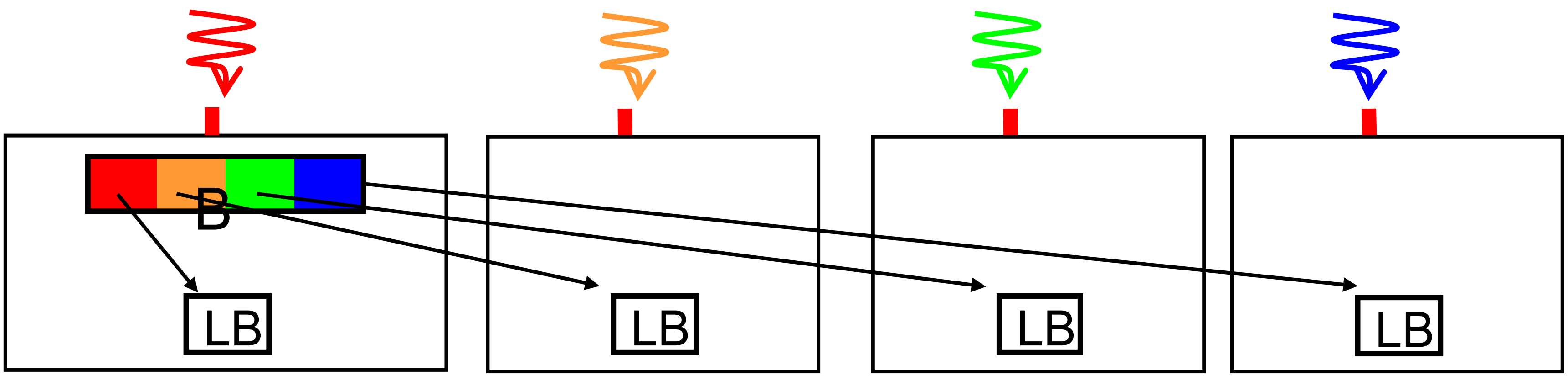
Communication of A in comm1() in mm_comm can be modified

```
MPI_Bcast(A, m*k, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

root

Another Pattern: MPI_Scatter

- In mm-comm, we do not want to broadcast B/C
- Instead, we want to distribute partial B/Cs ➔ MPI_Scatter



send data

send count, # of data per process

```
MPI_Scatter(B,   k*n/nprocs, MPI_DOUBLE,
                LB, k*n/nprocs, MPI_DOUBLE,
                0, MPI_COMM_WORLD);
```

} sender (root) information. valid only in root

} receiver information

recv data

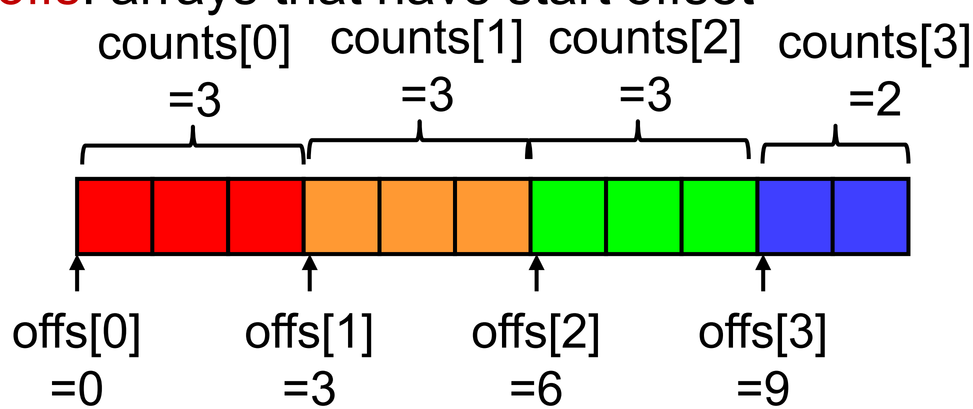
root

Non-uniform Scatter

- MPI_Scatter only supports uniform division
- → If n (width of B) is indivisible by nprocs, each process may take non-uniform counts of data
- → MPI_Scatterv would be useful

```
MPI_Scatterv(B,  counts, offs, MPI_DOUBLE,
                 LB, (e-s), MPI_DOUBLE,
                 0, MPI_COMM_WORLD);
```

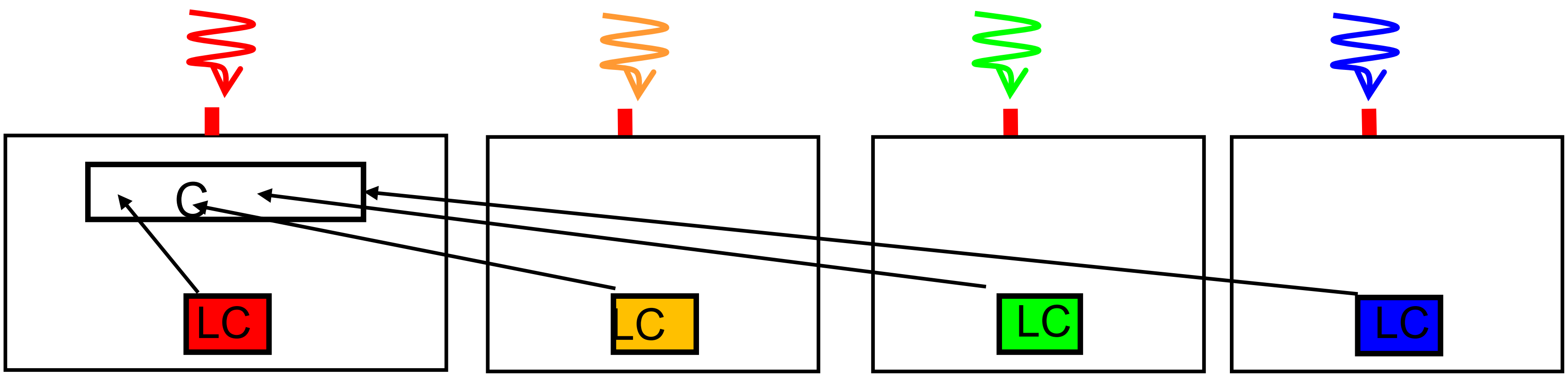
counts: arrays that have data counts for each process
offs: arrays that have start offset



Next Pattern: MPI_Gather

- After computation, we want to gather partial Cs on process 0

➔ MPI_Gather/MPI_Gatherv may be used



send data

send count, # of data per process

```
MPI_Gather(LC,   k*n/nprocs, MPI_DOUBLE,
           C,    k*n/nprocs, MPI_DOUBLE,
           0, MPI_COMM_WORLD);
```

sender information

recv data

receiver (root) information
valid only in root

root

Non-uniform Gather

- MPI_Gather gathers data from each process uniformly
- MPI_Gatherv can gather data of uniform sizes

```
MPI_Gatherv(LC, (e-s), MPI_DOUBLE,
            C, counts, offs, MPI_DOUBLE,
            0, MPI_COMM_WORLD);
```

} sender information

} receiver (root) informati

valid only in root

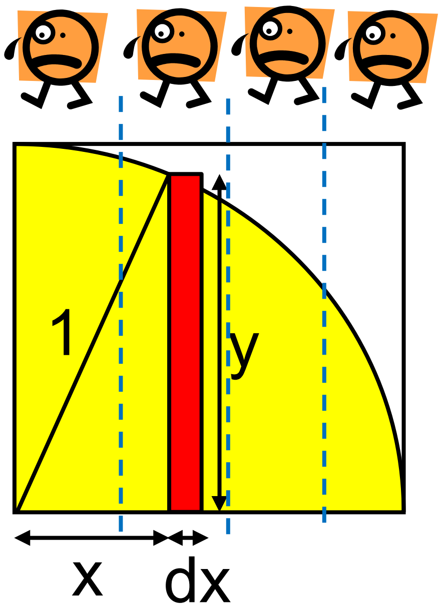
Meaings of counts and offs arrays are similar to MPI_scatterv

Next Pattern: MPI_Reduce with "mpi/pi" Sample

ppcomp-ex/mpi/pi/

- 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*x)

Using pi-mpi sample

- ppcomp-ex/mpi/pi

```
[make sure that you are at a interactive node (r7i7nX) ]
module load intel-mpi    [Do once after login]
[please go to your ppcomp-ex directory]
cd mpi/pi
make
[An executable file "pi" is created]
mpiexec -n 4 ./pi 100000000
```

Number of division

Reduction Operation by MPI_Reduce

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

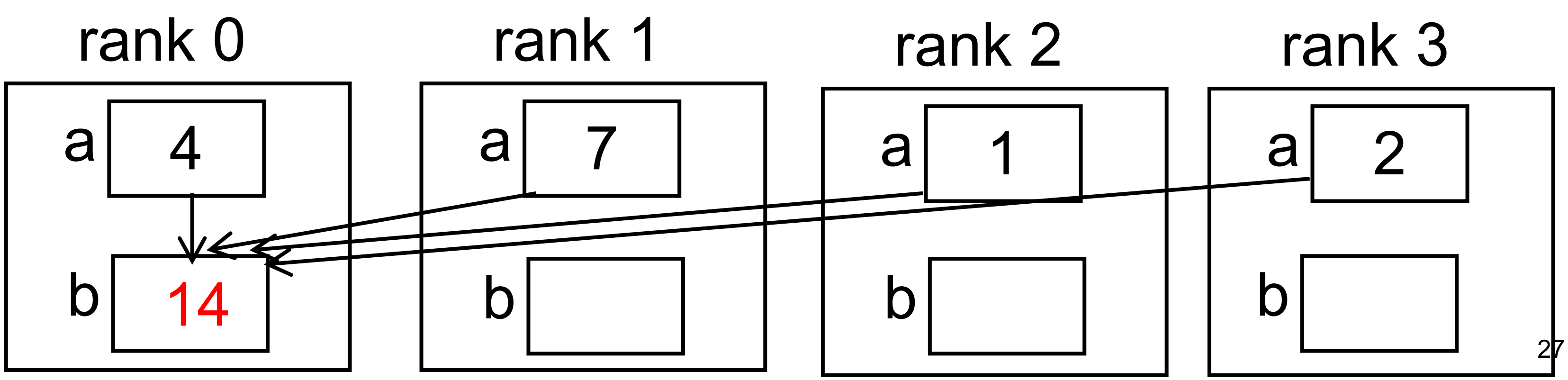
```
MPI_Reduce(&a, &b, 1, MPI_INT, MPI_SUM, 0,
                MPI_COMM_WORLD);
```

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.



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)

4	7	2	3	0	1	5	6	5

⇨ sum=33

  - MPI_Reduce(…) in MPI
    - If each input is an array, output is also an array
    - Operations are done for each index

4	7	2

3	0	1

5	6	5

⇨

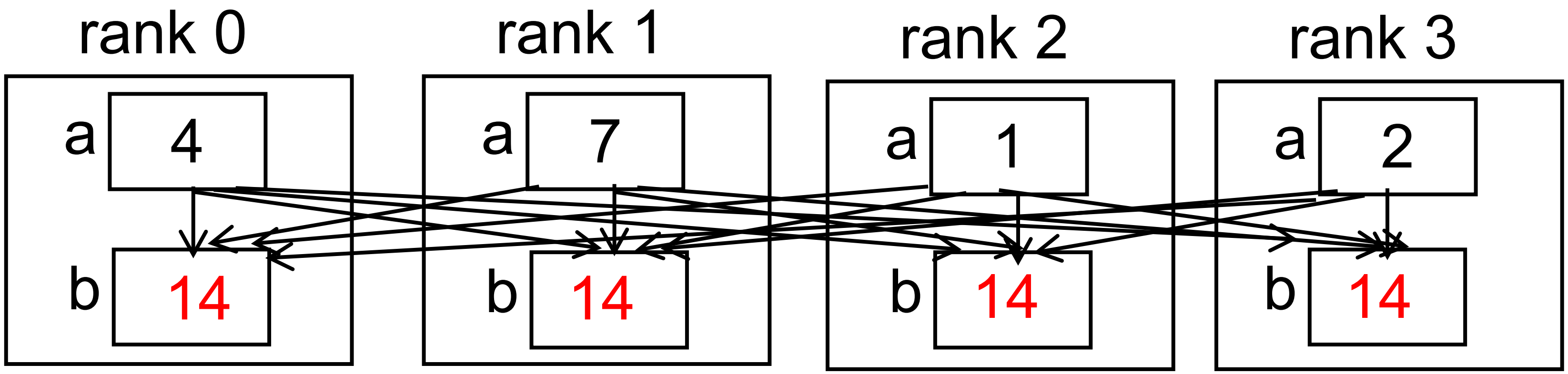
12	13	8

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

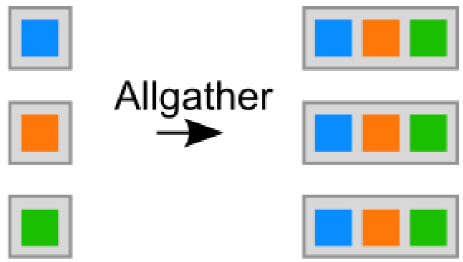
- Barrier synchronization: processes are stopped until all processes reach the point

```
MPI_Barrier(MPI_COMM_WORLD);
```

  - Used in sample programs, to measure execution time more precisely

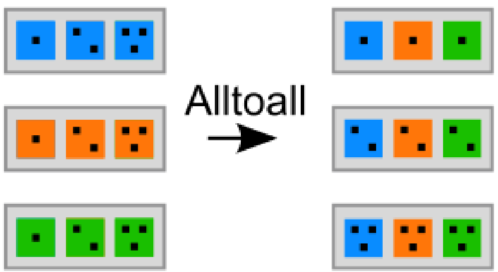
Other Collective Communications

- MPI_Allgather, MPI_Allgatherv
  - Similar to MPI_Gather. Gathered data are put on all processes



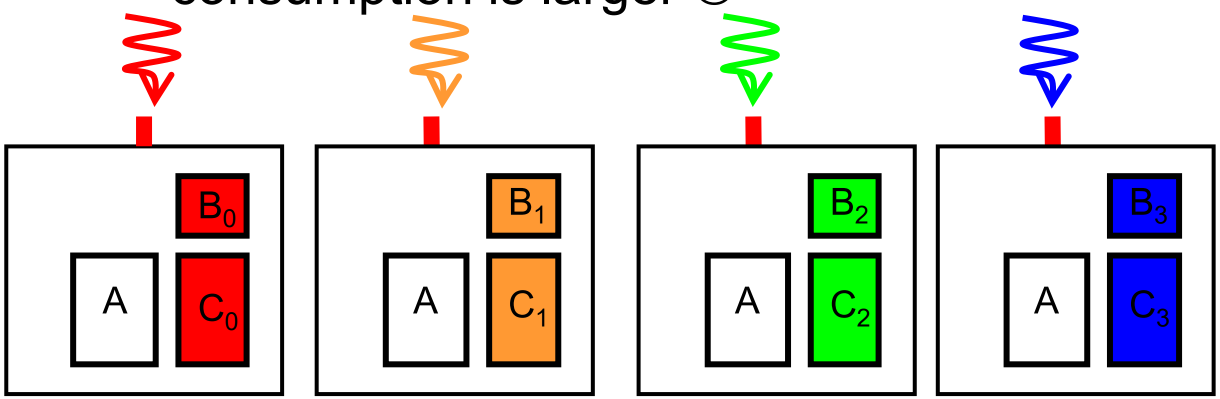
from Wikipedia

- MPI_Alltoall, MPI_Alltoallv
  - Each process has data. Each of them are scattered



[Advanced Topic] Re-considering Data Distribution of mm (Option of [M3])

- Consider memory consumption cost:
  - In mm, every process has whole matrix A → memory consumption is larger ☹



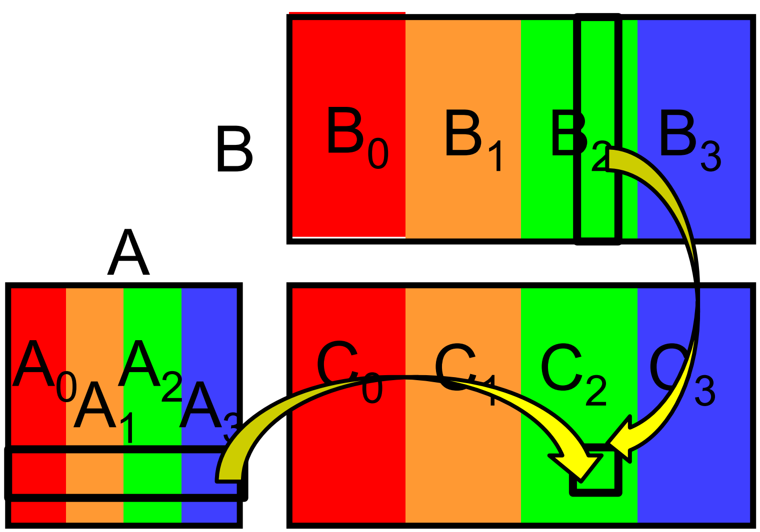
Memory amount of each process: O(mk+km/p+mn/p)

Each process has whole A

  - Even mm-comm has the same problem
- Can we reduce memory consumption?
  - The ideal case would be O(mk/p+km/p+mn/p)

Data Distribution of Memory Reduced "mm"

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

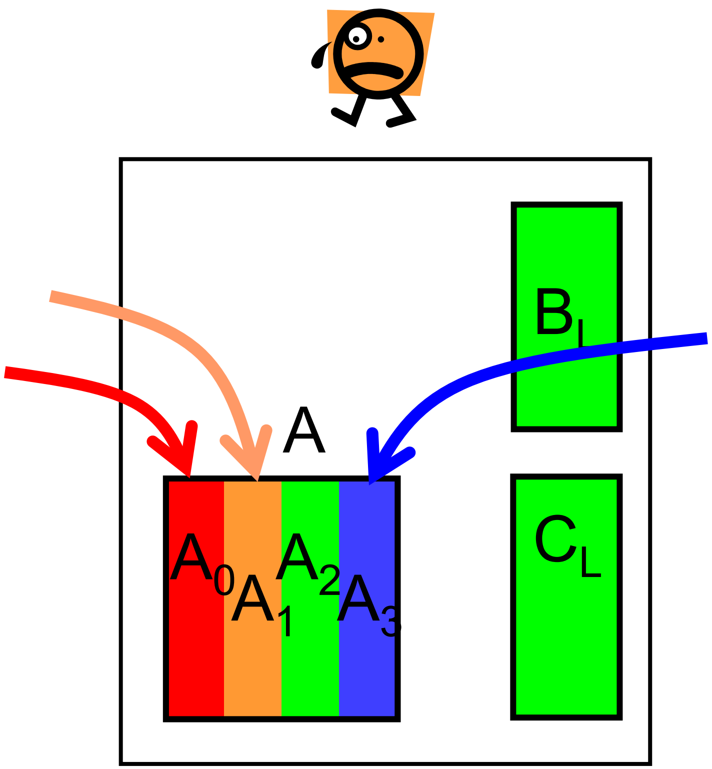


Memory consumption is smallest

- 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

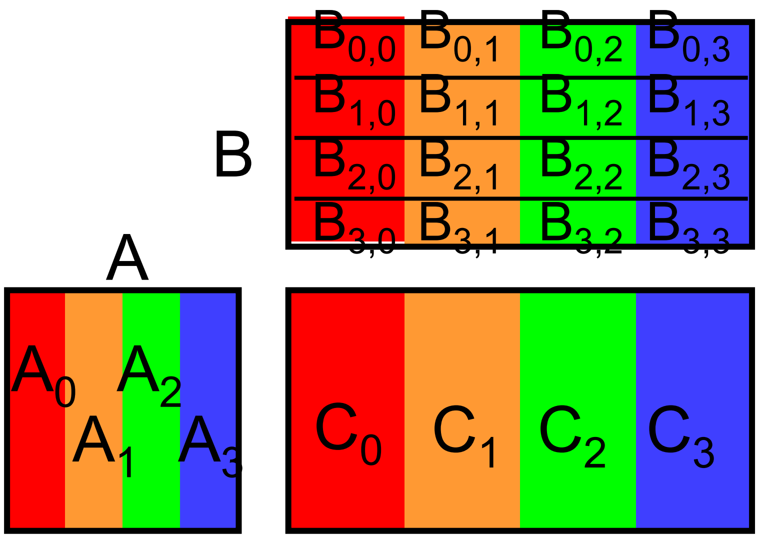


1. Collect entire A from other processes by communication
2. Compute $C_L = A \times B_L$

- Each process has (entire) A, B_L, C_L ➔ Same as mm ☹

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

<u>Step 0 :</u>

P_0 sends A_0 to all other processes

Every process P_r computes

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

<u>Step 1 :</u>

P_1 sends A_1 to all other processes

Every process P_r computes

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

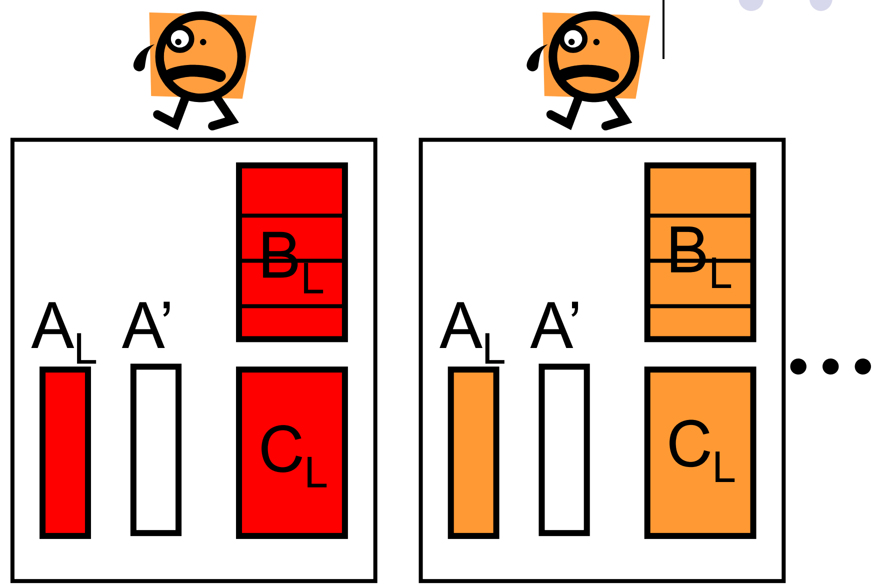
:

Repeat until *<u>Step (p-1)</u>*

Actual Data Distribution

Every process has partial A, B, C

- A_L on process r ⇔ A_r
- B_L on process r ⇔ B_r
- C_L on process r ⇔ C_r



- Additionally, every process should prepare a receive buffer ➔ 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 (1)

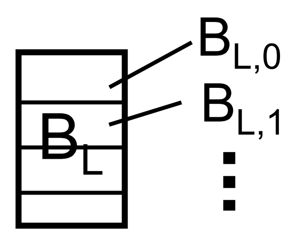
On every process r:

```
for (s = 0; s < p; s++) {   // s: step no, p: number of processes
  if (r == s) {
    for (dest = 0; dest < p; dest++)
      if (dest != r) MPI_Send(A_L, …, dest, …);
  } else
    MPI_Recv(A', …, s, …);

  if (r == s)
    Compute C_L += A_L × B_L,s
  else
    Compute C_L += A' × B_L,s
}
```

P_s sends its A_L to all other processes

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



Using Collective Communication

- Communication part of the previous code is same as MPI_Bcast pattern!

  Note:

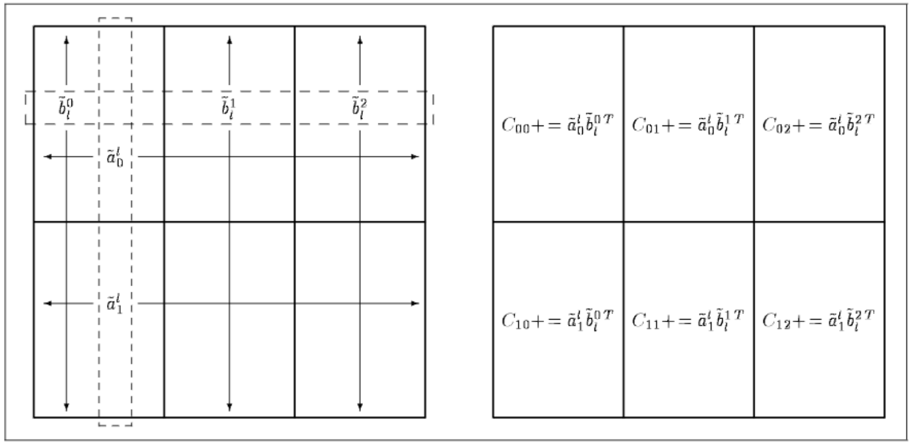
  1. The different "roots" are used for different steps
  2. Sent data is A_L, while received data is A'

One of solutions:

```
if (r == s) {
    // on root
    MPI_Bcast(A_L, ...., s, MPI_COMM_WORLD);
} else {
    // on non-root
    MPI_Bcast(A', ...., s, MPI_COMM_WORLD);
}
```

[More Advanced] Improvements of Memory Reduced Version

- 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



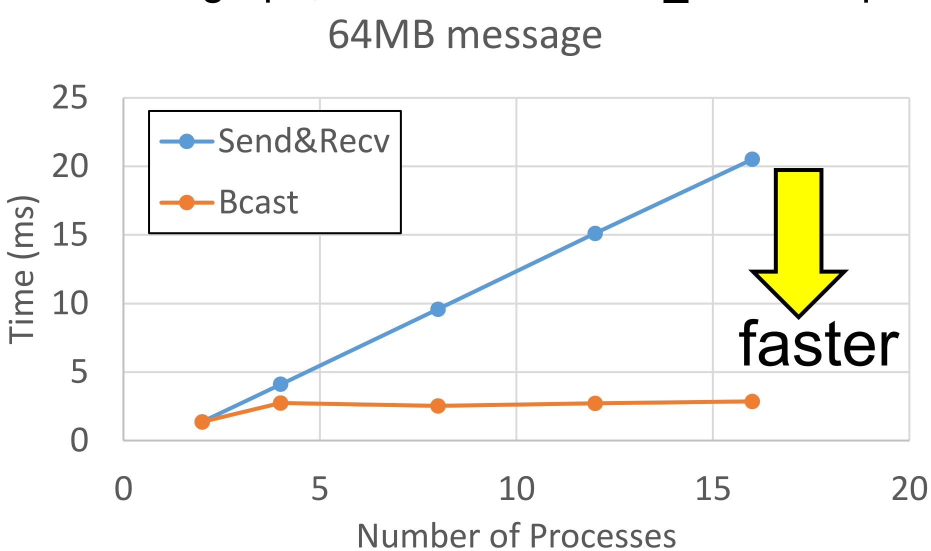
Performance of Collective Communication

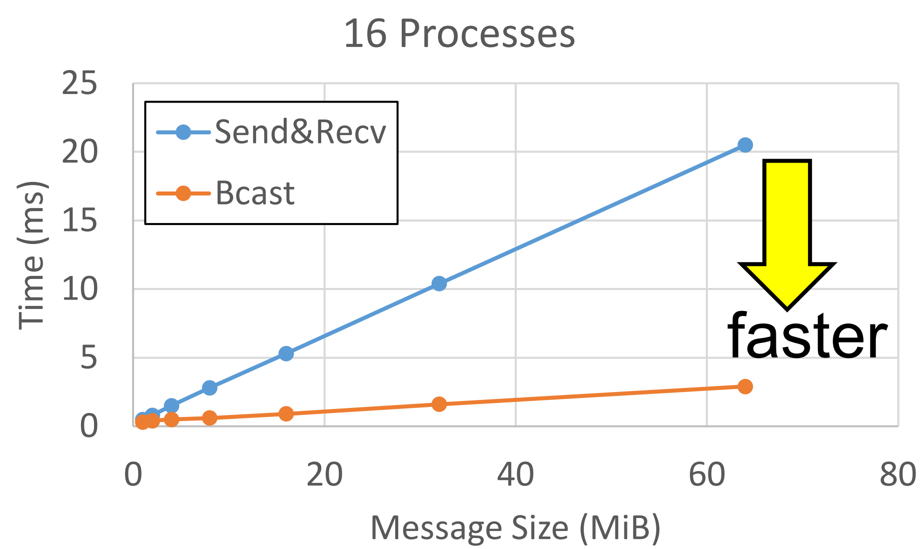
“Do I Really Need to Learn New Functions?”

- You can still use MPI_Send/MPI_Recv multiple times, but collective functions are often faster

In the graph, rank 0 called MPI_Send for p-1 times to other processes

On TSUBAME4
1 proc / node





- 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

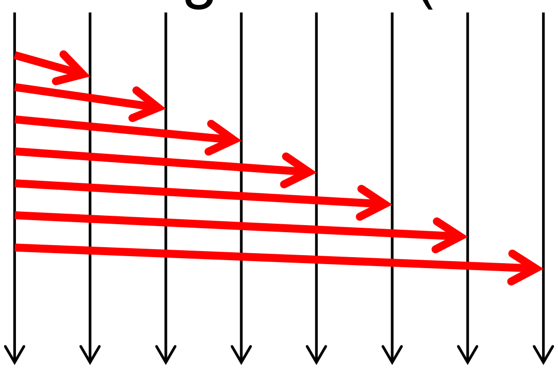
FYI: Measurement Method

- Measurement in the previous page was done with ppcomp-ex/mpi/mpibcast/
- intel-mpi is used
- NOTE: job*.sh in this directory need to consume TSUBAME points
  - job*.sh use > 2nodes

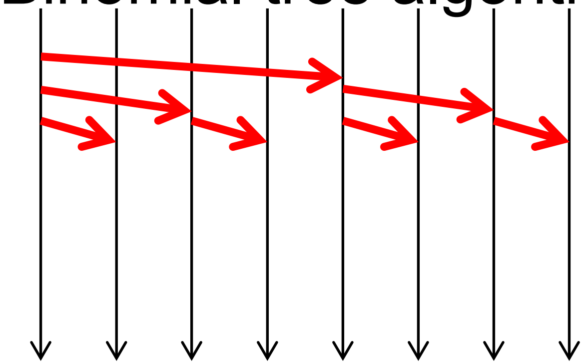
Why are Collective Communications Fast?

- Since MPI library uses *scalable* communication algorithms
  - Case of broadcast:

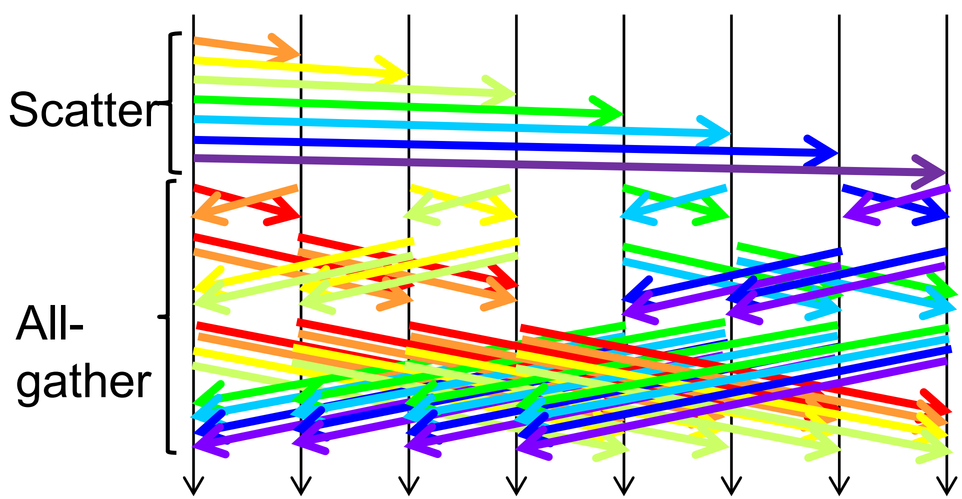
Flat tree algorithm (slow)



Binomial tree algorithm

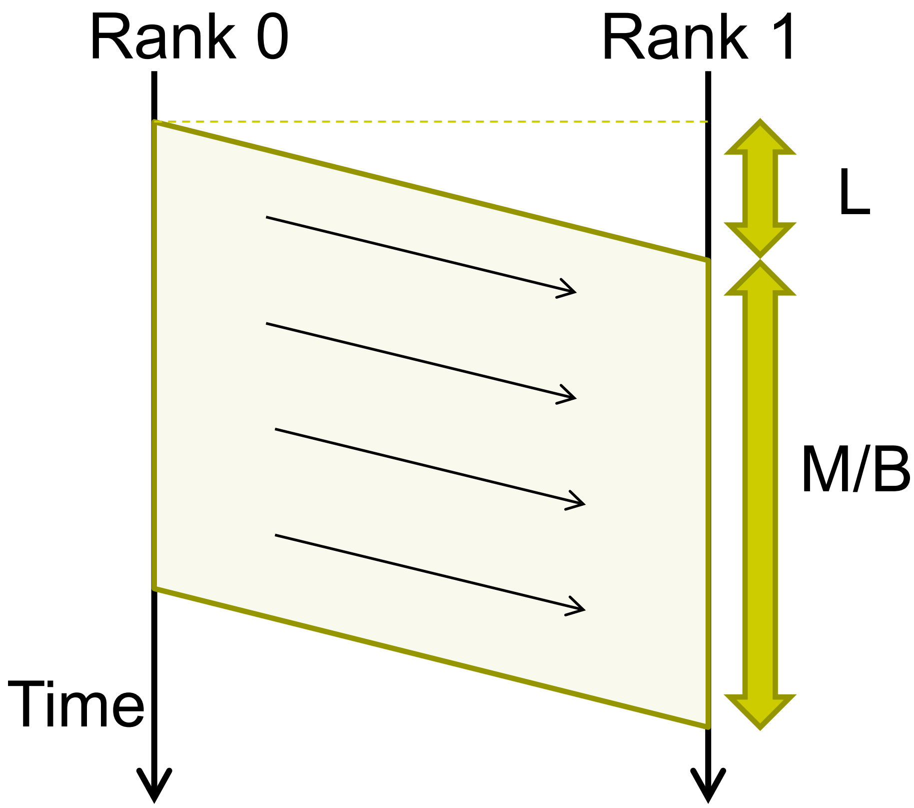


Scatter&Allgather algorithm



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

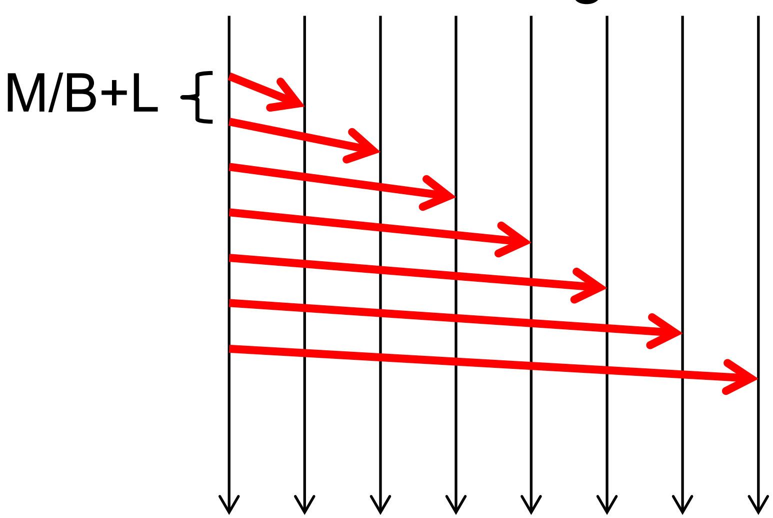
※ Be aware of difference between "Byte" and "bit": 1Byte=8bit

※ 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

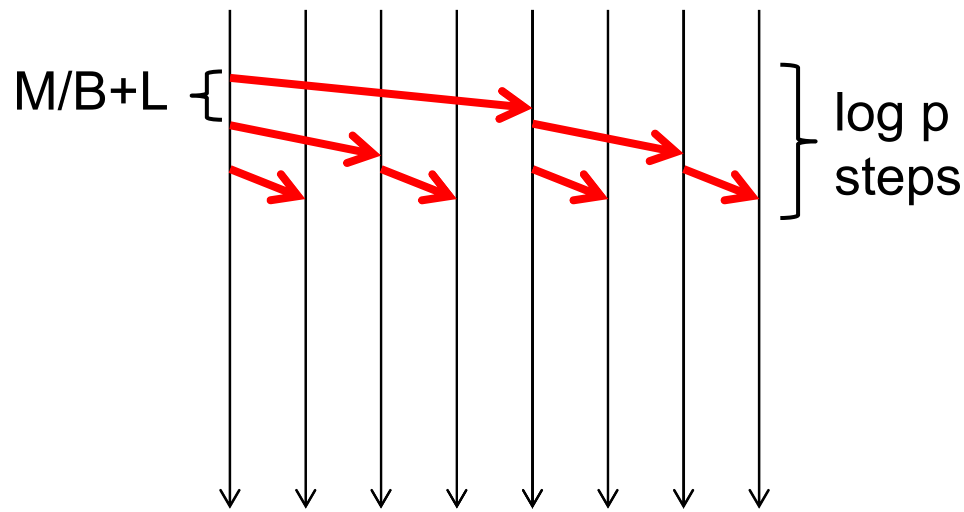
Flat tree algorithm



p(M/B+L)

→ Slow

Binomial tree algorithm



(log p)(M/B+L)

※difference between p-1 and p is ignored

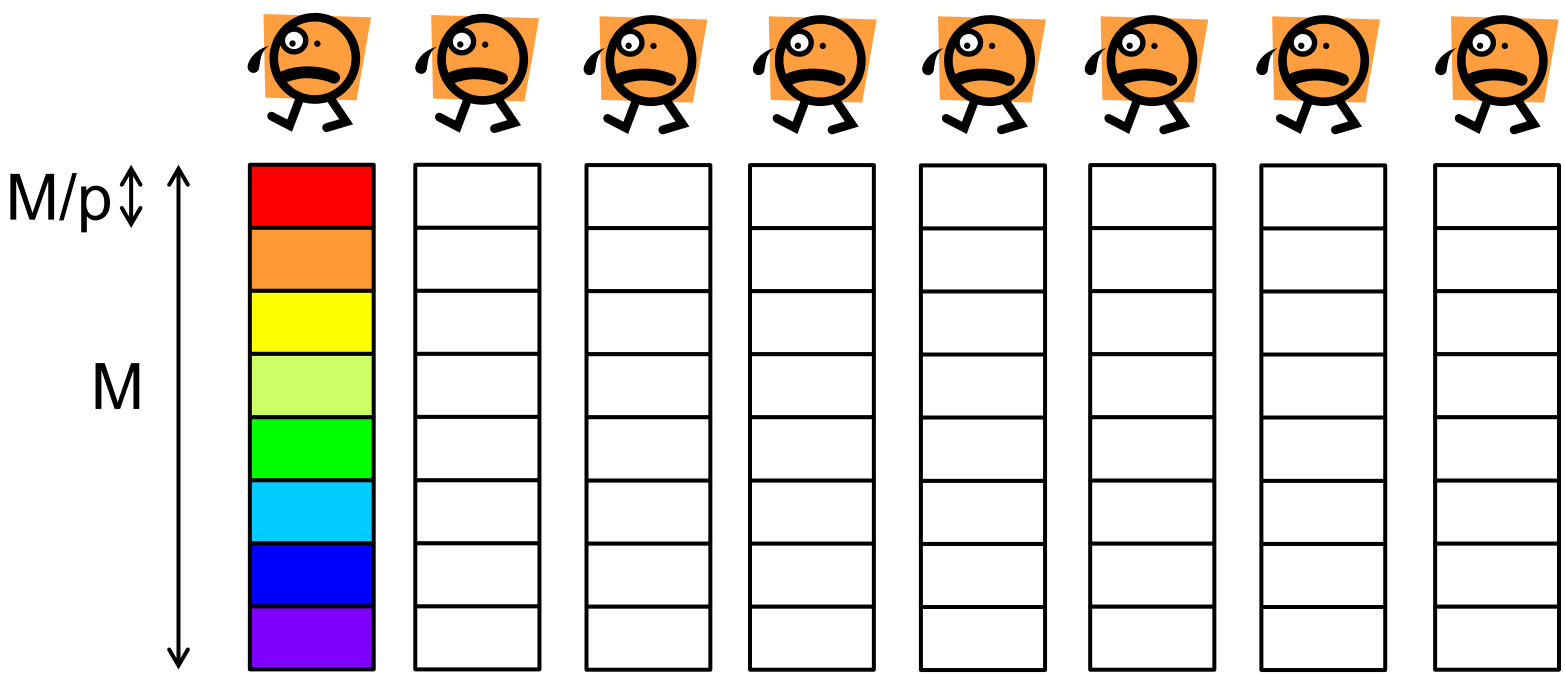
Broadcast by Scatter&Allgather Algorithm (1)

(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.

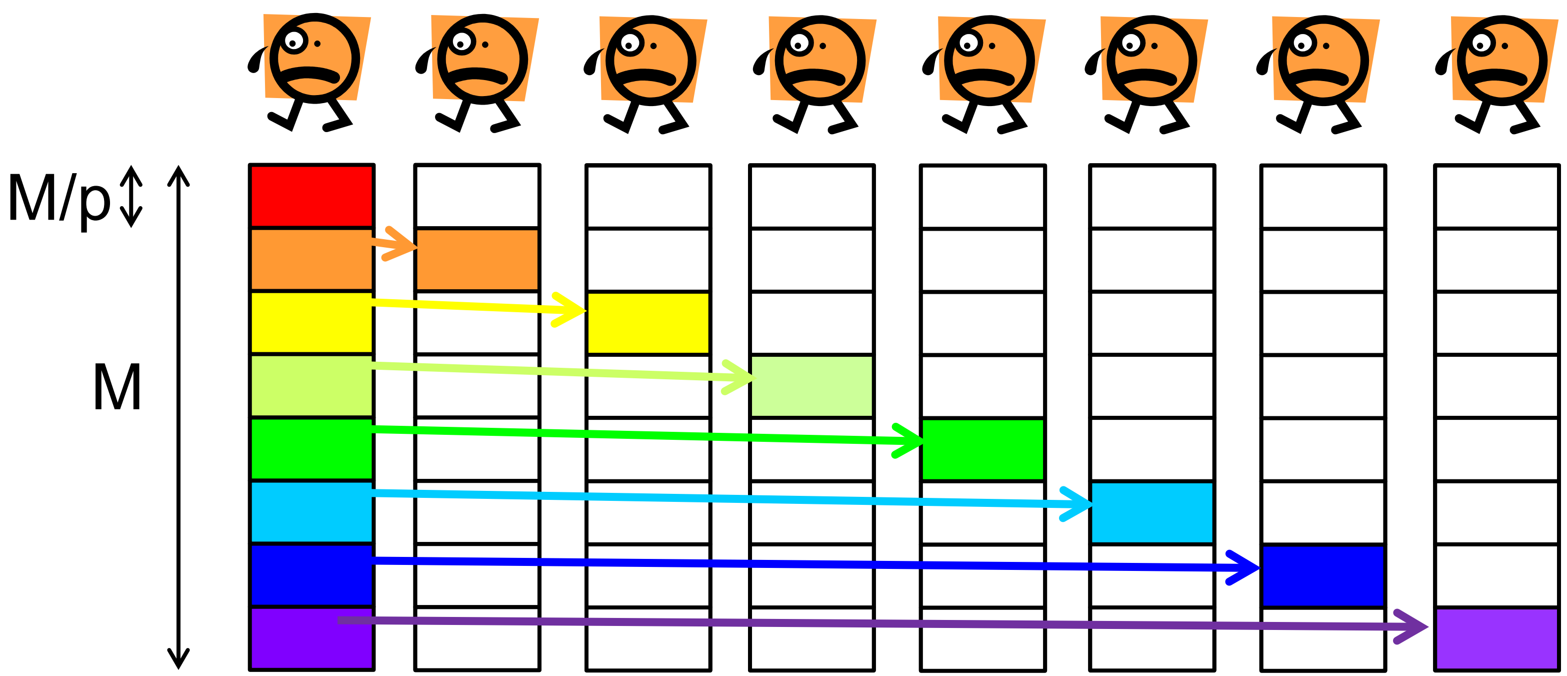


Scatter&Allgather Algorithm (2)

(1) The root process divide the message into p parts

(2) Scatter: *i*-th part goes to process *i*

(3) Allgather



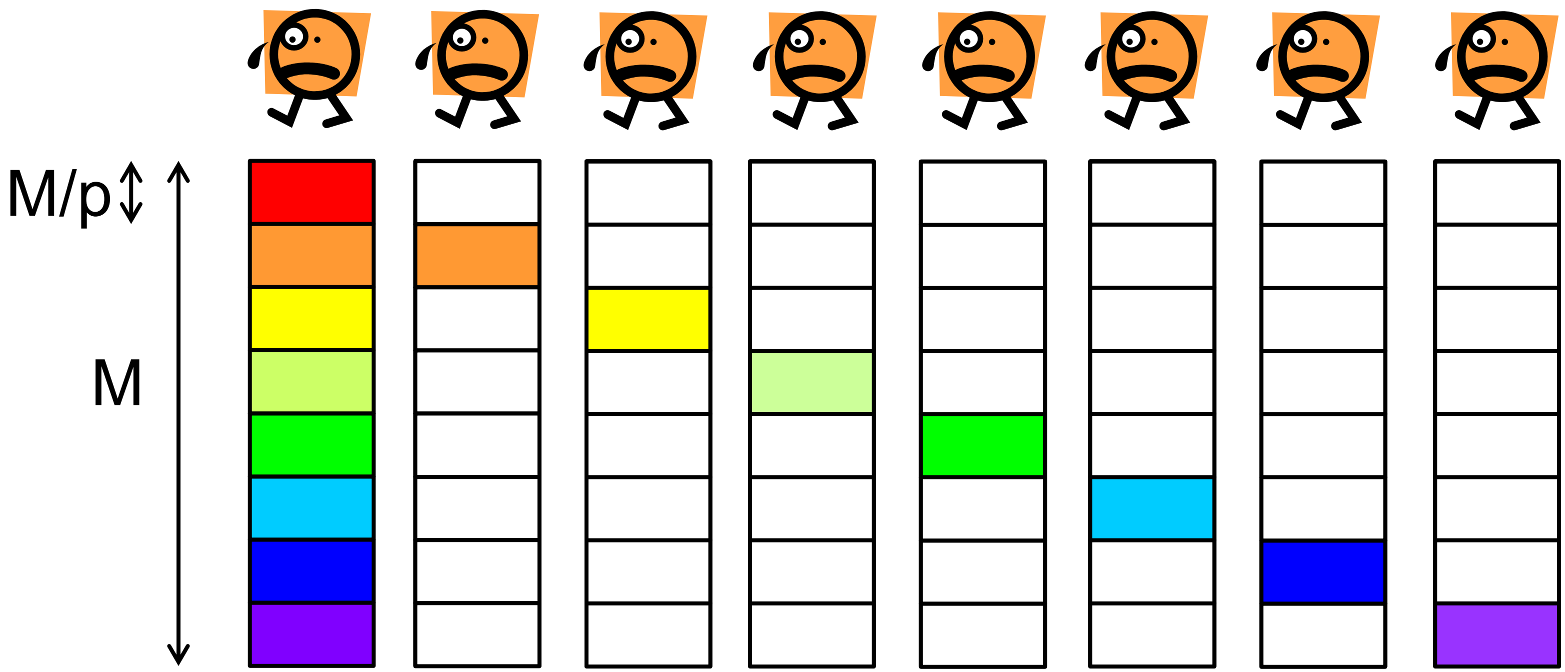
Scatter&Allgather Algorithm (3)

(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



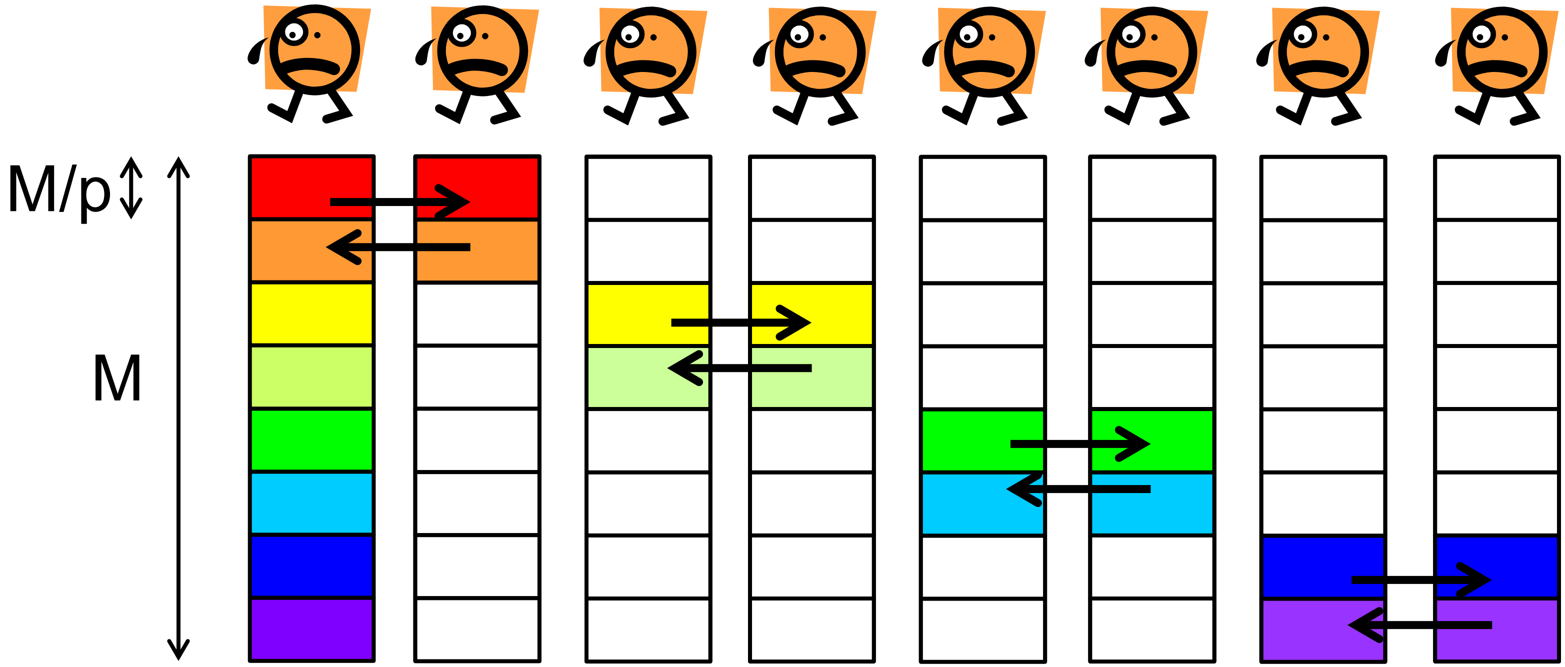
Scatter&Allgather Algorithm (3)

(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



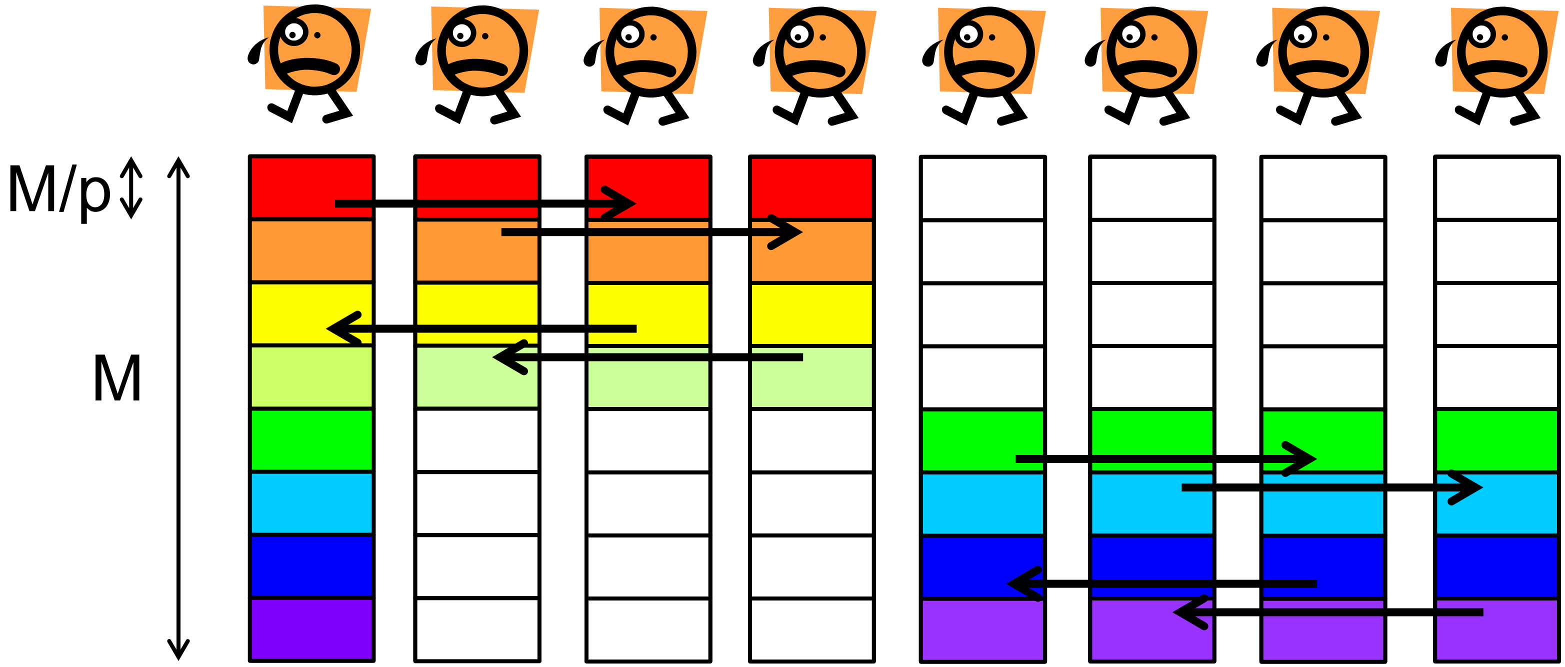
Scatter&Allgather Algorithm (3)

(1) The root process divide the message into p parts

(2) Scatter

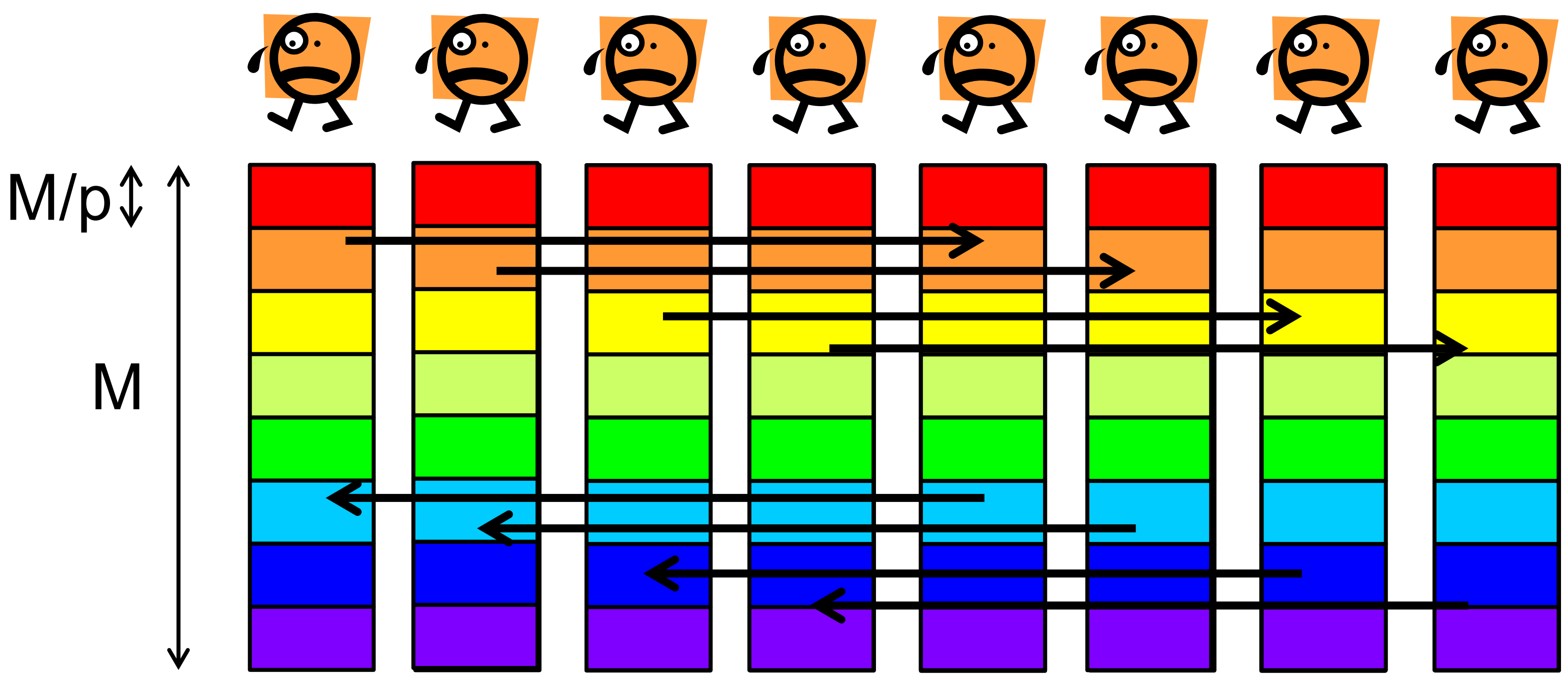
(3) Allgather in *log p* steps

- If p=8, we use 3 steps



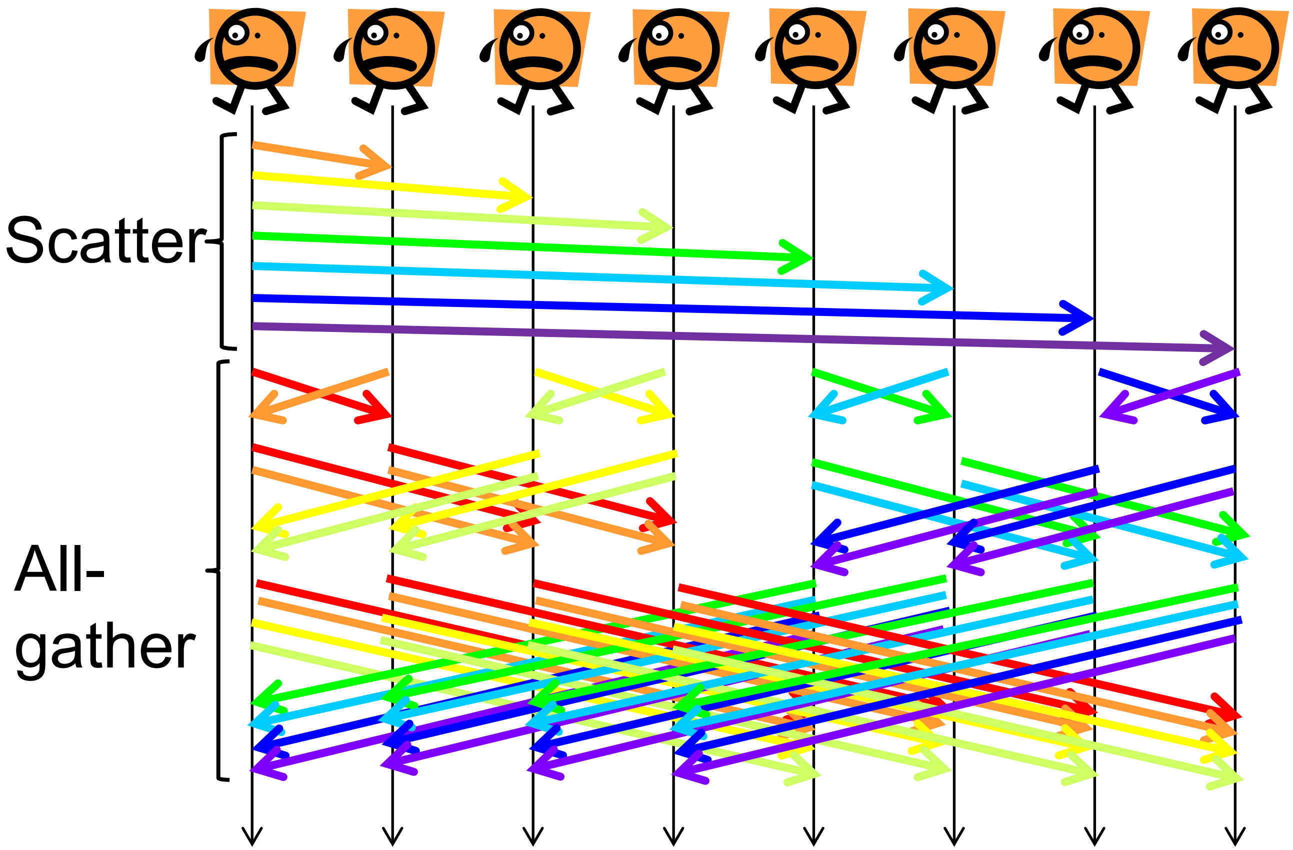
Scatter&Allgather Algorithm (3)

(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



$$\overbrace{M/B + pL}^{\text{Scatter}} + \underbrace{M/B + (\log p)L}_{\text{All-gather}}$$

Comparison of Broadcast Algorithms

- Consider two extreme cases
  - If M is sufficiently large: *M/B+L → M/B*
  - If M is close to zero: *M/B+L → 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 → Fastest
Cost with very small M (M is ignored)	p L	(log p) L → Fastest	(p + log p) L

Many MPI libraries implement multiple algorithms
They switch them automatically according to message size M ☺

Assignments in MPI Part

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

Due date: June 9 (Monday)

[M1] Parallelize “diffusion” sample program by MPI

[M2] Parallelize “bsort” sample program by MPI

[M3] Evaluate speed of “mpi/mm” sample in detail

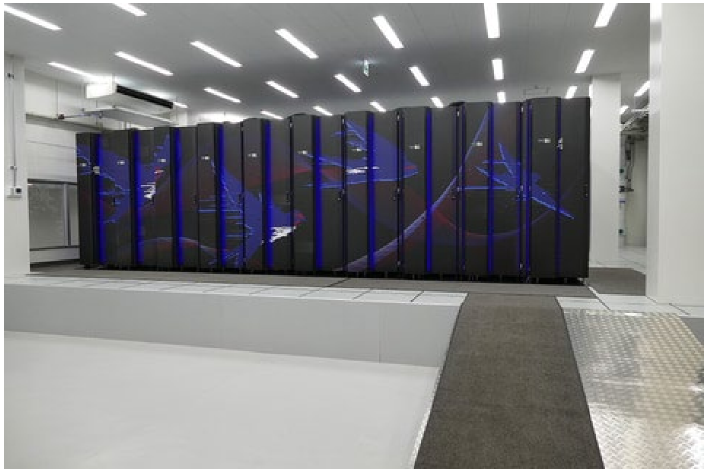
[M4] (Freestyle) Parallelize *any* program by MPI

For more details, please see ppcomp25-12 slides

Next Optional Class

- May 29: (Short) class + TSUBAME4 tour
  - Shorter explanation in Classroom + zoom, as usual
  - 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 31 in Campus map
https://www.titech.ac.jp/english/0/maps/suzukakedai

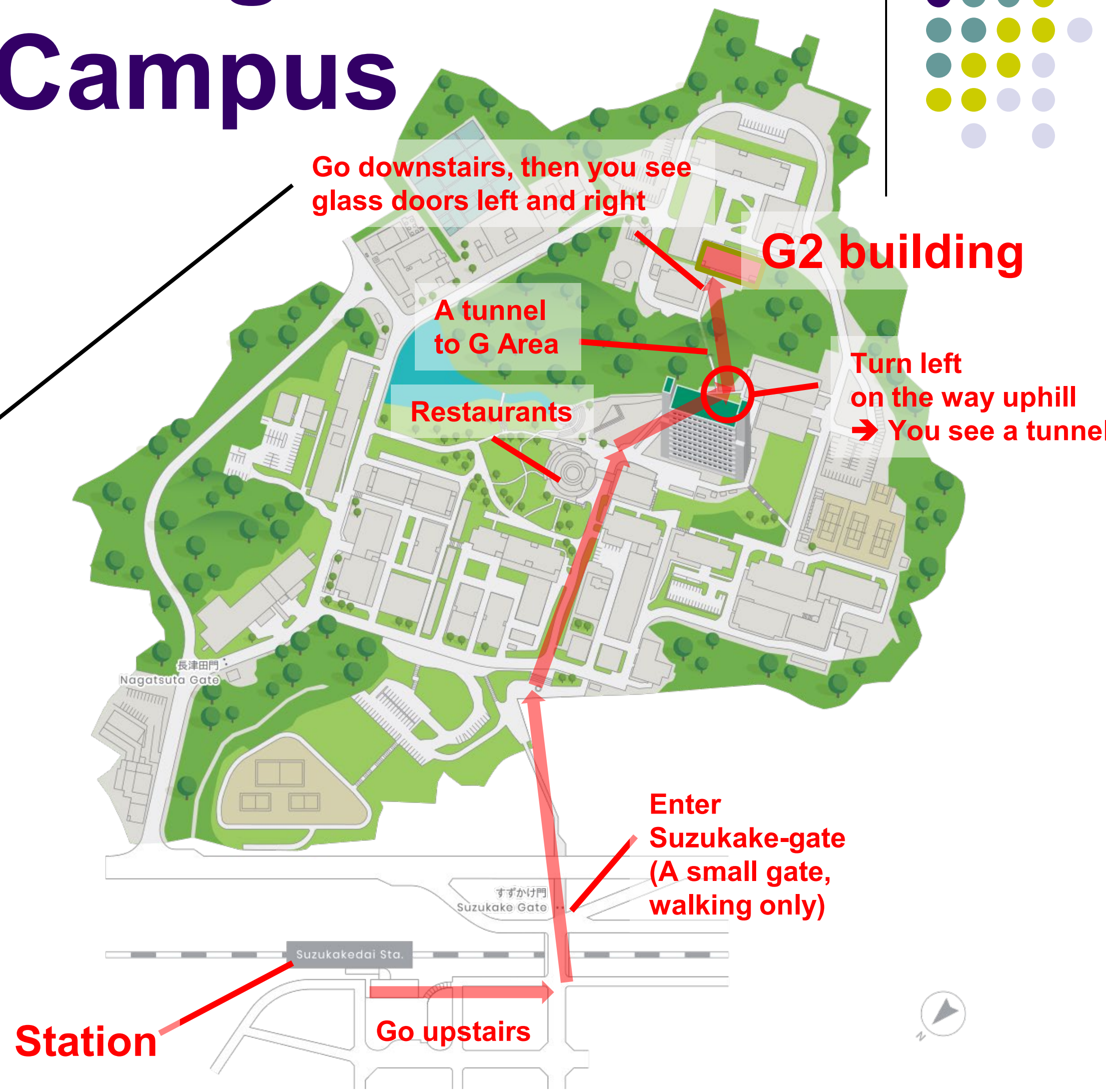


Way to G2 Building at Suzukakedai Campus

Suzukakedai Campus:

10 minutes walk from Suzukakedai Station, Denentoshi Line

Enter the right door and go up to the 2^{nd} 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