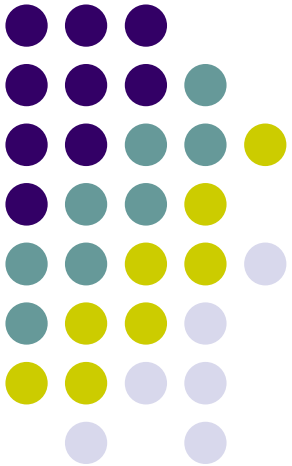
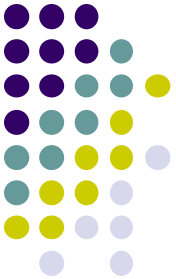
Practical Parallel Computing (実践的並列コンピューティング)

Part3: MPI (2)
June 15, 2020



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

Overview of This Course

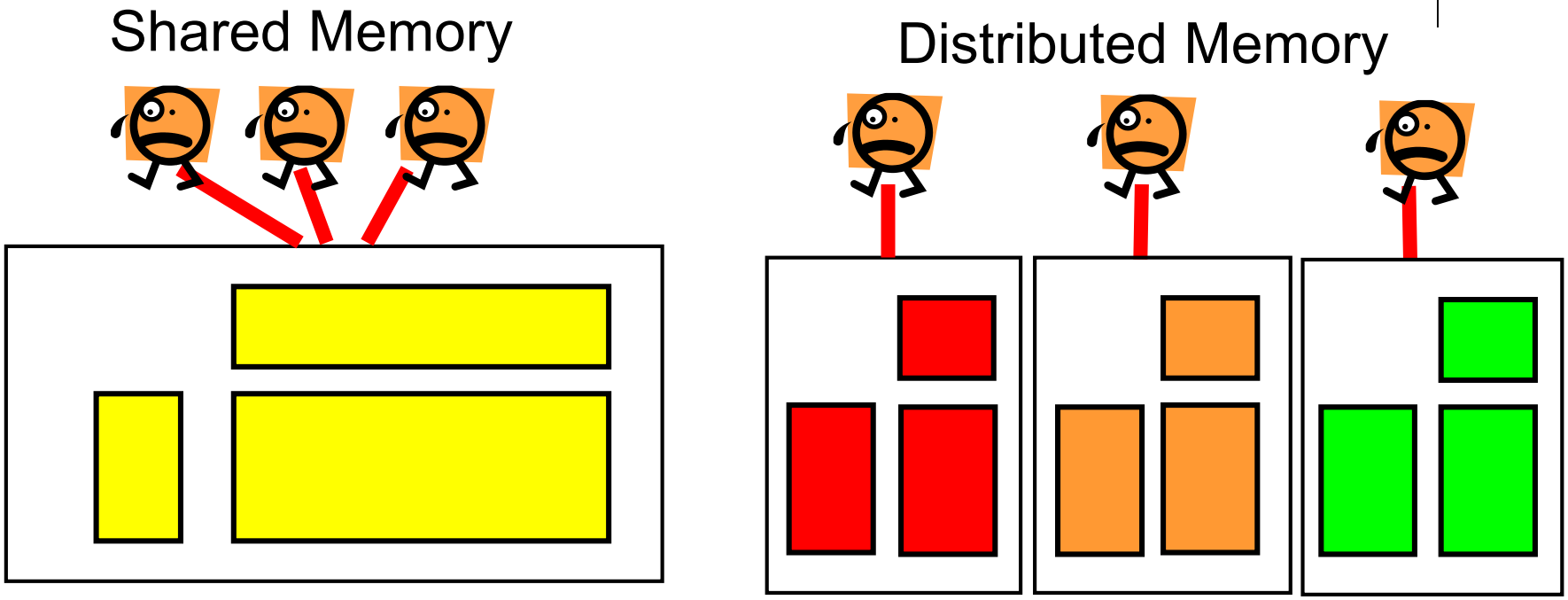


- Part 0: Introduction
  - 2 classes
- Part 1: OpenMP for shared memory programming
  - 4 classes
- Part 2: GPU programming
  - 4 classes ← We are here (1/4)
  - OpenACC (1.5 classes) and CUDA (2.5 classes)
- Part 3: MPI for distributed memory programming
  - 3 classes ← We are here (2/3)

Class Evaluation/授業アンケート

- The URL is announced in the Zoom chat

Shared Memory Model and Distributed Memory Model



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

→ Message passing (communication) is required

Basics of Message Passing: Peer-to-peer Communication

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

Rank 0 computes "int a[16]"

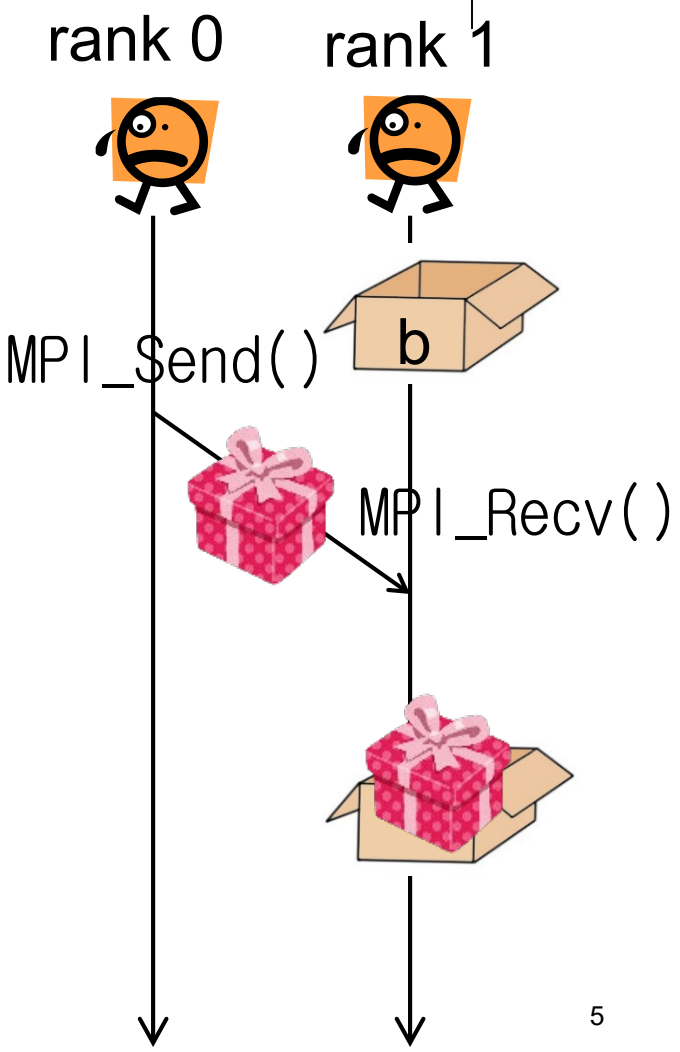
Rank 1 wants to see contents of a!

Rank0:

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

Rank1:

- Prepares a memory region (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)

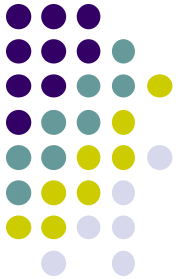
```
MPI_Recv(b, 16, MPI_INT, 2, 200, MPI_COMM_WORLD, &stat);
```

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

→ The message that matches the condition is delivered

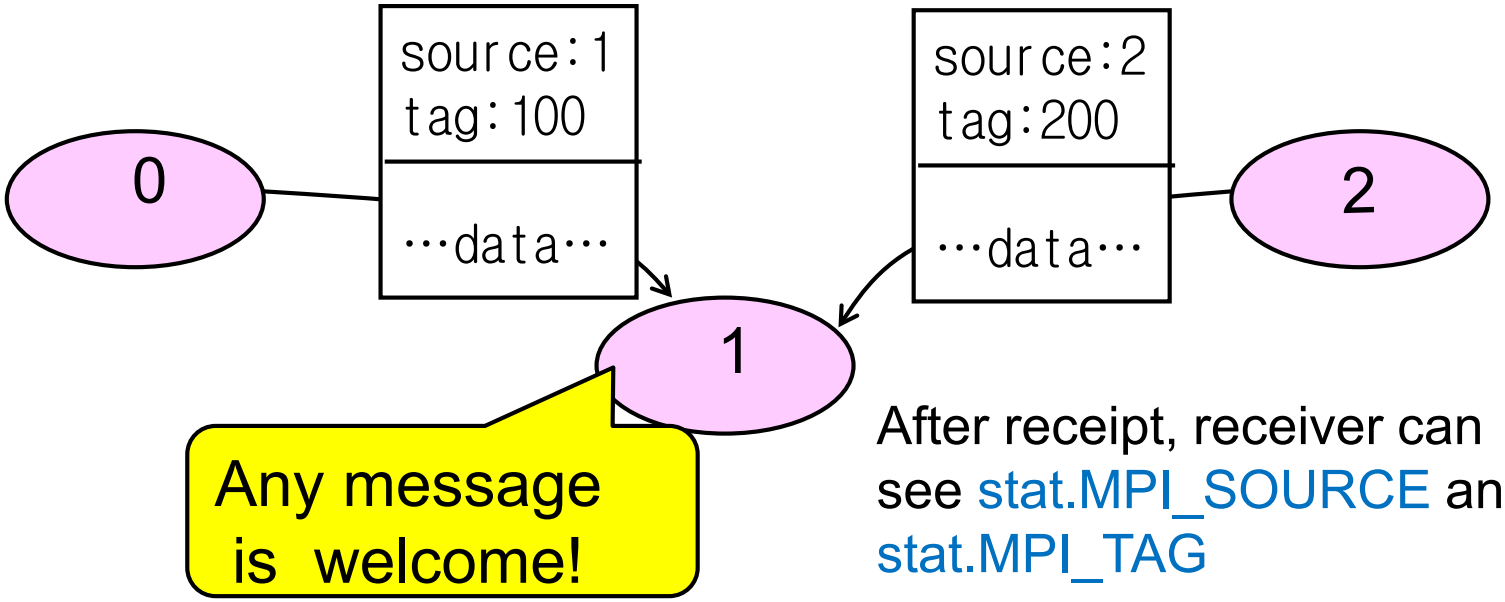
- Other messages should be received by other MPI_Recv calls

Notes on MPI_Recv: Message Matching (2)

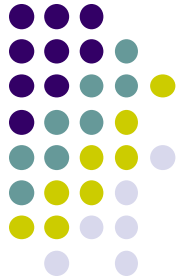


- In some algorithms, the sender may not be known beforehand
  - cf) client-server model
- For such cases, MPI_ANY_SOURCE / MPI_ANY_TAG can be used

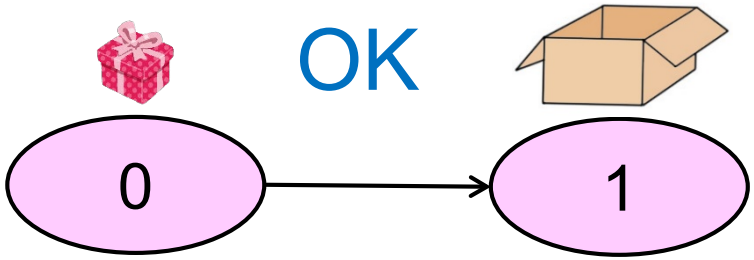
```
MPI_Status stat;
MPI_Recv(b, 16, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
   MPI_COMM_WORLD, &stat);
```



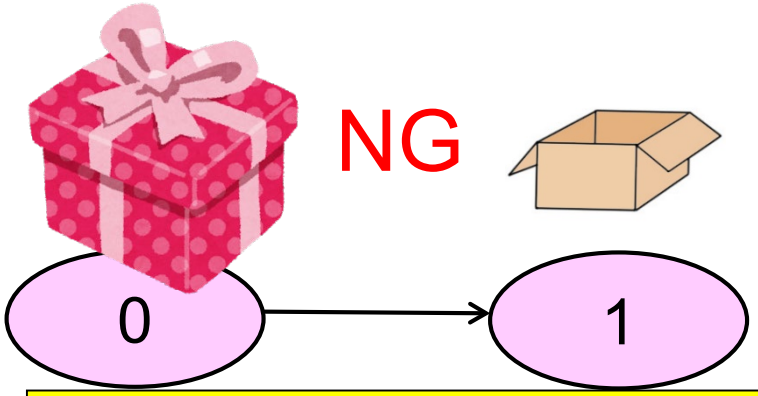
Notes on MPI_Recv: What If Message Size is Unmatched



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



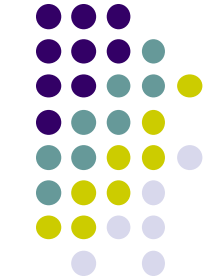
If message is smaller than expected, it's ok
→ Receiver can know the actual size by
MPI_Get_Count(&stat, MPI_INT, &s);



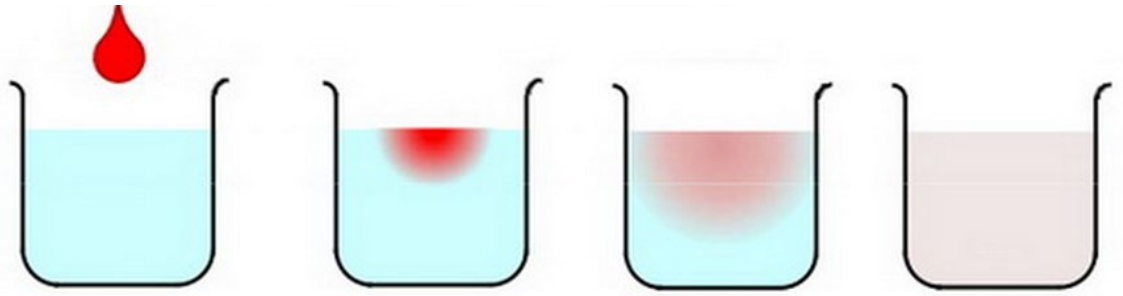
If message is larger than expected, it's an error (the program aborts)

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

Case of “diffusion” Sample
related to [M1]



An example of diffusion phenomena:



The ink spreads gradually, and finally the density becomes uniform (Figure by Prof. T. Aoki)

Available at /gs/hs1/tga-ppcomp/20/diffusion/

- Execution：./diffusion [nt]
  - nt: Number of time steps

You can use /gs/hs1/tga-ppcomp/20/diffusion-mpi/ as a base. Makefile uses mpicc

Data Structure in Original "diffusion"

An Array for "even" steps

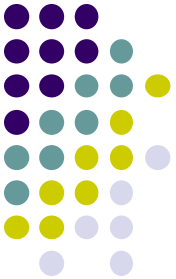
An Array for "odd" steps

NY

NX

How can we distribute data?

How Do We Parallelize "diffusion" Sample?



On OpenMP:

[Algorithm] Parallelize spatial (Y or X) for-loop

- Each thread computes its part in the space
- Time (T) loop cannot be parallelized, due to dependency

[Data] Data structure is same as original

On MPI:

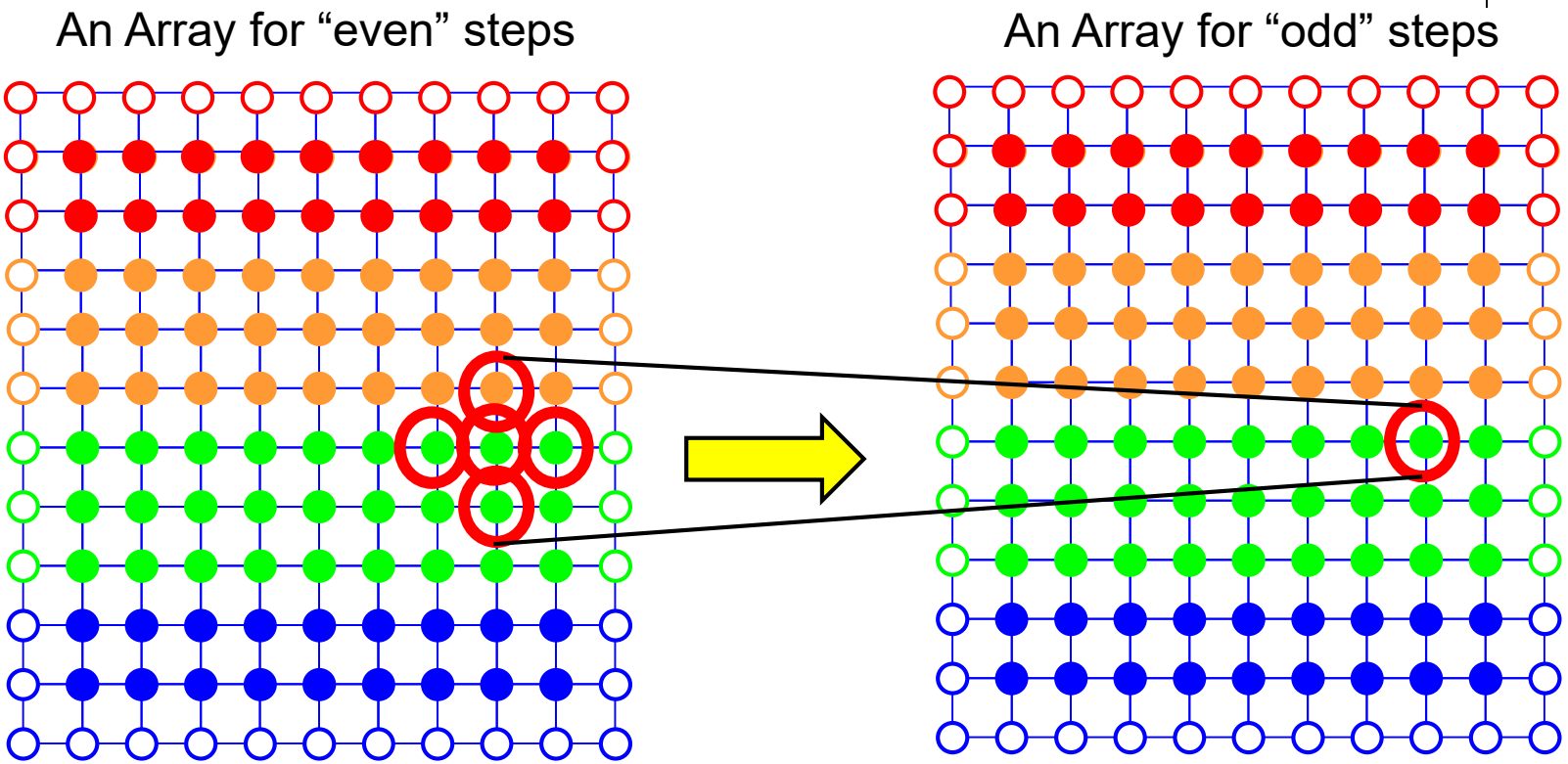
[Algorithm] Same as above

- Each process computes its part in the space

[Data] Both arrays are divided among processes

- Each process has its own part of arrays

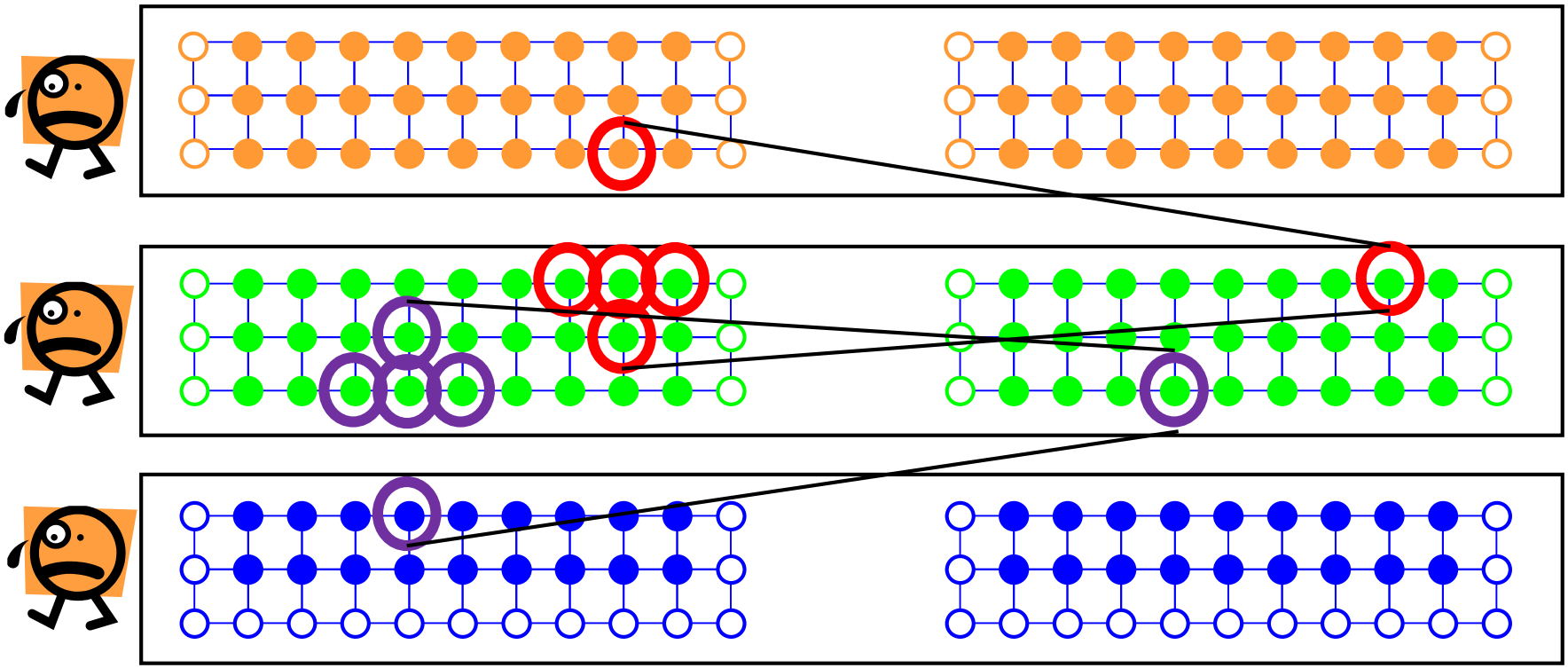
Considering Data Distribution (1)



- A color = a process

Considering Data Distribution (2)

- How about below distribution?



Computation requires data in other processes

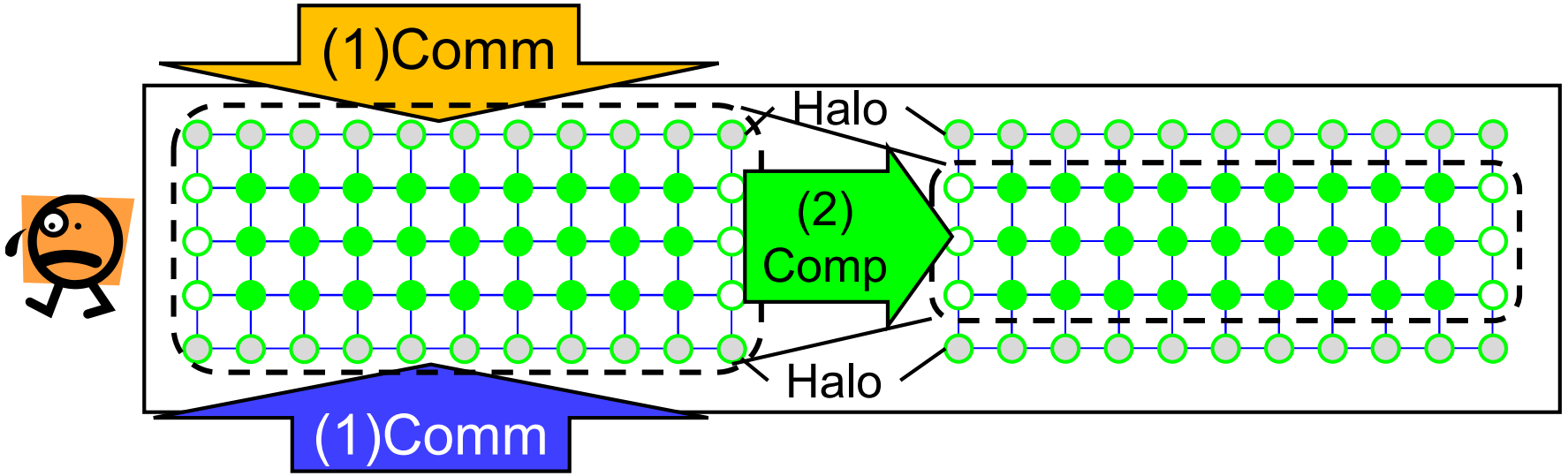
→ Message passing is required

So, where should received data be put?

Introducing “Halo” Region

- It is a good idea to make additional rows to arrays

→ called “Halo” region or “sleeve” region



Each time step consists of:

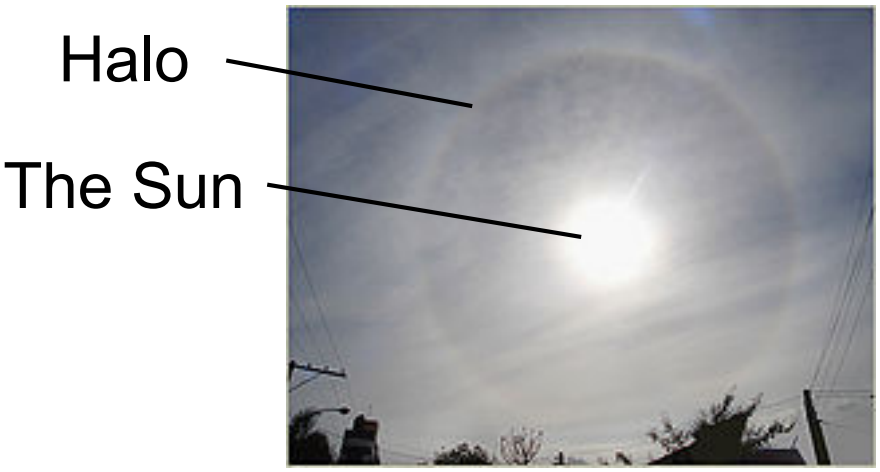
(1) Communication: Recv data and store into “halo” region

- Also neighbor processes need “my” data

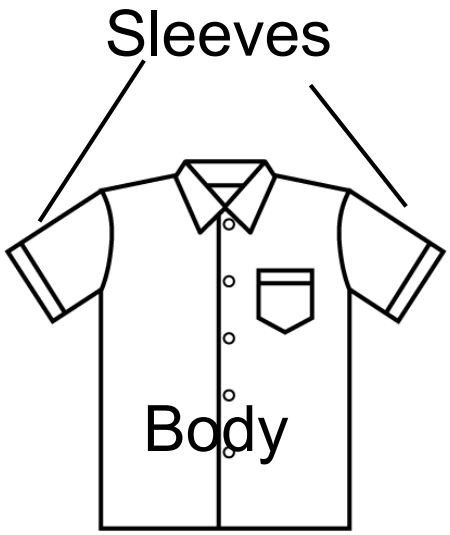
(2) Computation: Old data at time t (including “halo”)

→ New data at time t+1

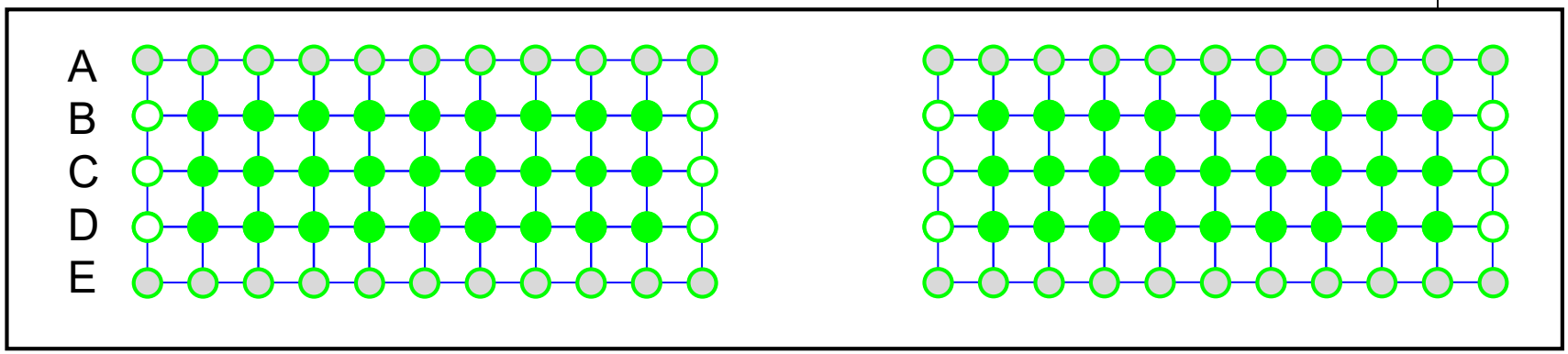
The name of “Halo” Region



en.wiktionary.org



Overview of MPI “diffusion”



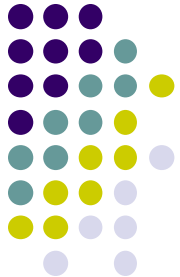

```
for (t = 0; t < nt; t++) {
  if (rank > 0) Send B to rank-1
  if (rank < size-1) Send D to rank+1
  if (rank > 0) Recv A from rank-1
  if (rank < size-1) Recv E from rank+1
  Computes points in rows B-D
  Switch old and new arrays
}
```

(1) Communication in “old” array

(2) Computation “old” array ⇒ “new” array

This version is still unsafe, for possibility of deadlock

A Sample with Neighbor Communication

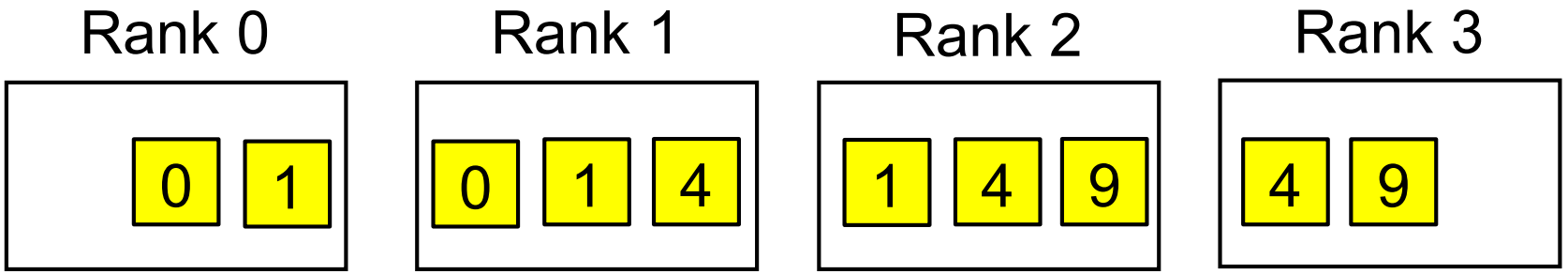


A sample is available at /gs/hs1/tga-ppcomp/20/neicomm-mpi

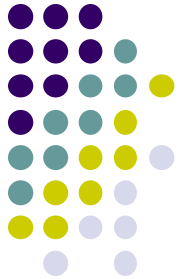
Execution: mpiexec –np [np] ./neicomm

(1) Each process prepares its local data

(2) Each process receives data from its neighbors (rank-1 and rank+1)



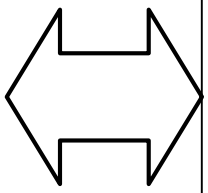
Behavior of neicomm-mpi Sample



Unsafe version ☹

When neicomm_unsafe()
is called in main()

```
Send to rank-1
Send to rank+1
Recv from rank-1
Recv from rank+1
```



Safe version ☺

When neicomm_safe()
is called in main()

```
Start to recv from rank-1
Start to recv from rank+1
Sent to rank-1
Sent to rank+1
Finish to recv from rank-1
Finish to recv from rank+1
```

※The sample does not finish!
To abort it, press Ctrl+C

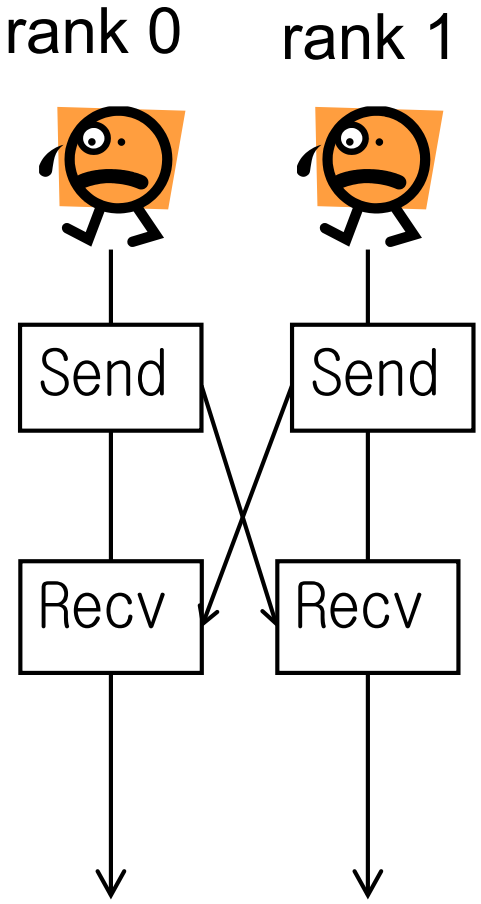
Deadlock

- Why?
  - The sample “deadlocks” with 2 processes

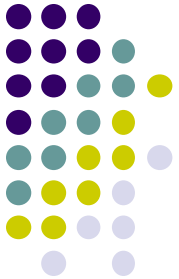
This is caused by behavior of MPI_Recv() and MPI_Send()

- MPI_Recv() blocks (does not finish) until the message arrives
- MPI_Send() may block until the message is received by receiver

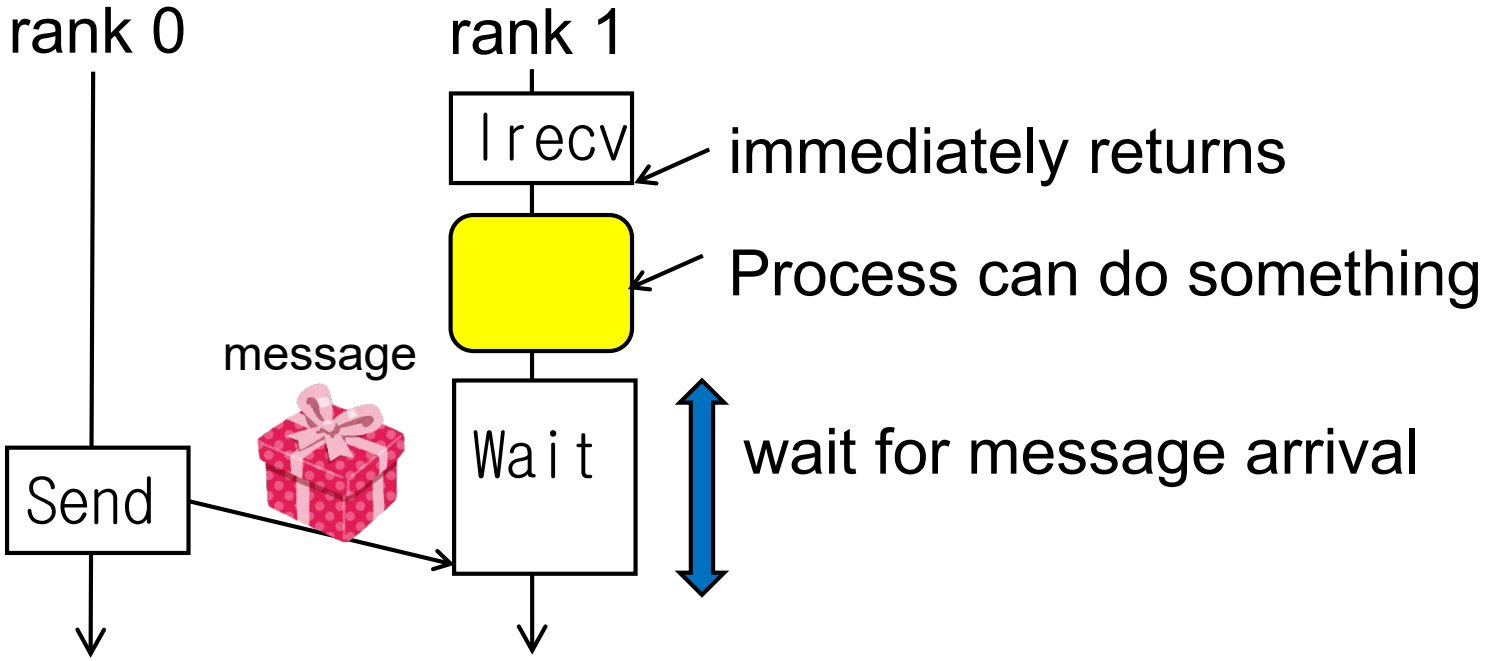
When message size is large



Non-Blocking Communication to Avoid Deadlock

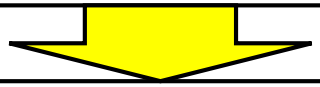


- Non-blocking communication: starts a communication (send or receive), but does not wait for its completion
  - MPI_Recv is blocking communication, since it waits for message arrival
- Program must wait for its completion later



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

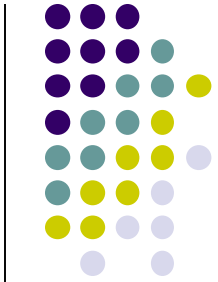
MPI_Wait: wait for message arrival

MPI_Request looks like a “ticket” for the communication

Functions Related to Non-blocking Communication

- MPI_Isend(buf, n, type, dest, tag, comm, &req); ←start send
- 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…

Assignments in MPI Part (Abstract)



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

Due date: 11AM, June 29 (Monday)

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

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

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

For more detail, please see June 11 slides

Next Class

- MPI (3)
  - Improvement of “matrix multiply” sample
  - Group Communication