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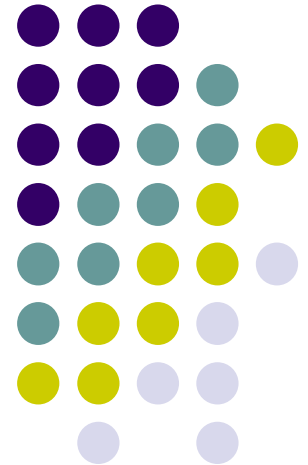# (実践的並列コンピューティング)

Part 3: MPI
No 2: Message Passing
May 22, 2023

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
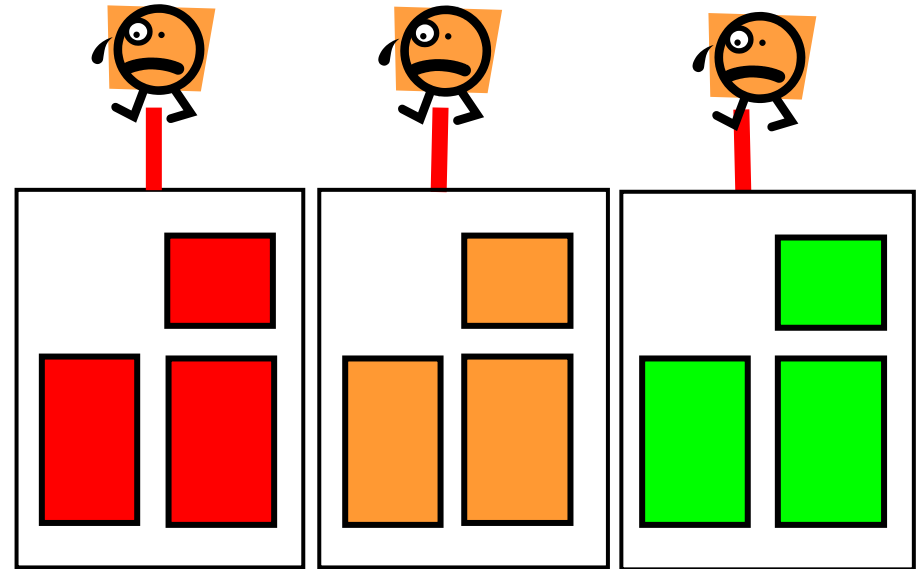 - 4 classes   ← We are here (2/4)

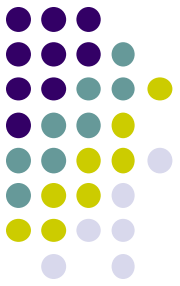# Shared Memory Model and Distributed Memory Model

Shared Memory
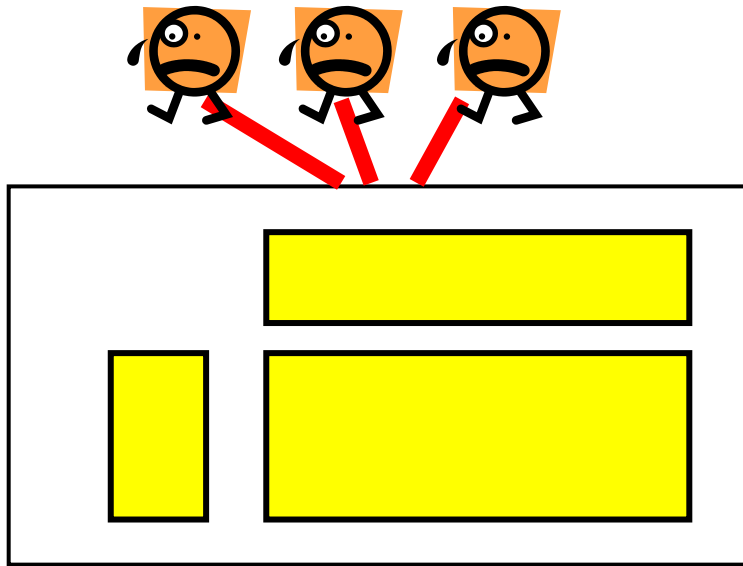
Distributed Memory

- In distributed memory model, a process CANNOT read/write other processes' memory directory

- How can a process access data on others?

→ Message passing (communication) is required

3

test-mpi sample

- /gs/hs1/tga-ppcomp/23/test-mpi

```
[make sure that you are at a interactive node (r7i7nX) ]
module load cuda openmpi    [Do once after login]
cd ~/t3workspace            [In web-only route]
cp -r /gs/hs1/tga-ppcomp/23/test-mpi  .
cd test-mpi
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: /gs/hs1/tga-ppcomp/23/test-mpi/

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

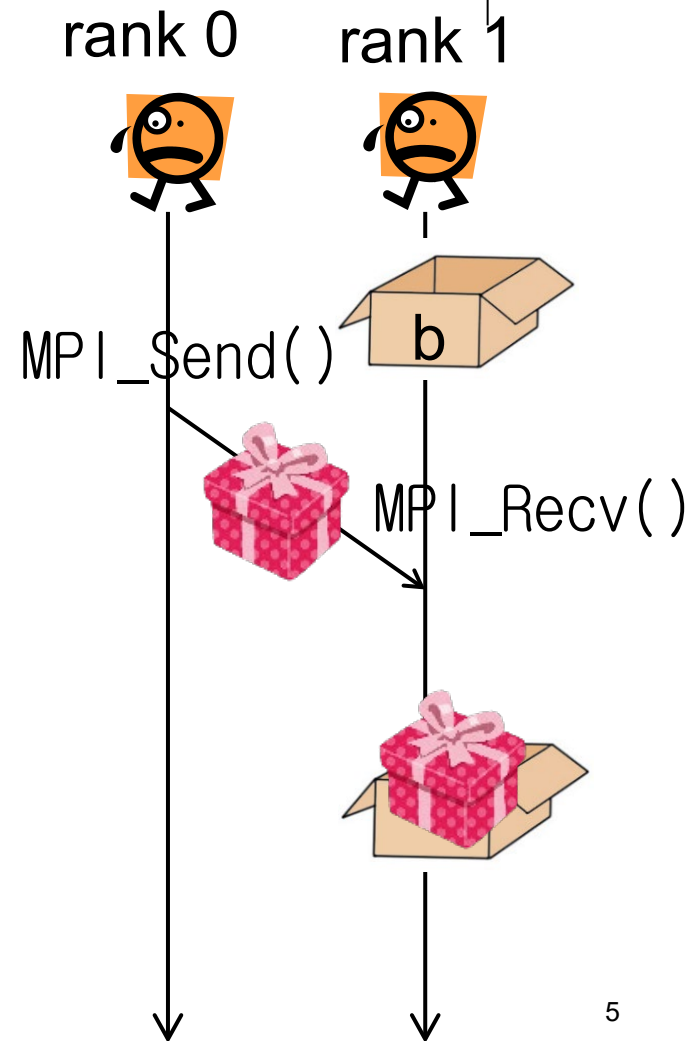Rank 1 wants to see contents of a!

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 !

rank 0   rank 1

MPI_Send()   b

MPI_Recv()

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)

rank 0

source:0
dest: 1
tag:100

6

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);
```

rank 0

source:0
dest:1
tag:100

source:2
dest:1
tag:200
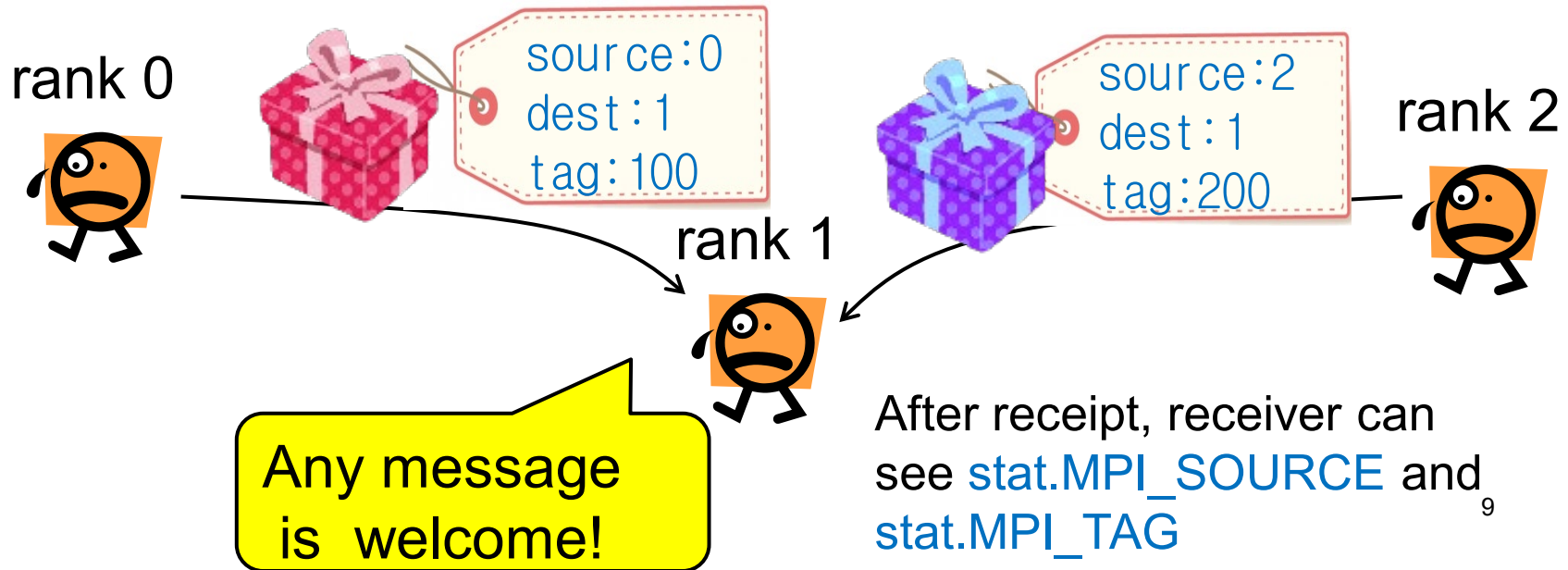
rank 2

rank 1

I only want a message with tag 200 from 2 !

- 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);
```


rank 0

source:0
dest:1
tag:100

source:2
dest:1
tag:200

rank 2

rank 1

Any message is welcome!

After receipt, receiver can see stat.MPI_SOURCE and stat.MPI_TAG

9

Notes on MPI_Recv:
What If Message Size is Unmatched

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


OK

$\leqq$

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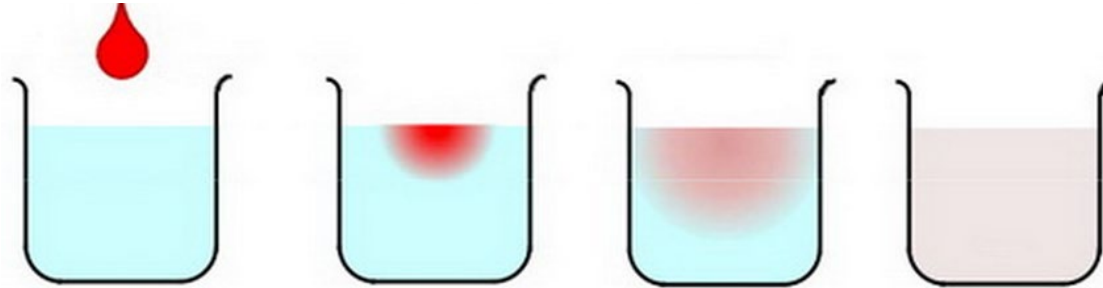→ Receiver can know the actual size by
MPI_Get_Count(&stat, MPI_INT, &s);

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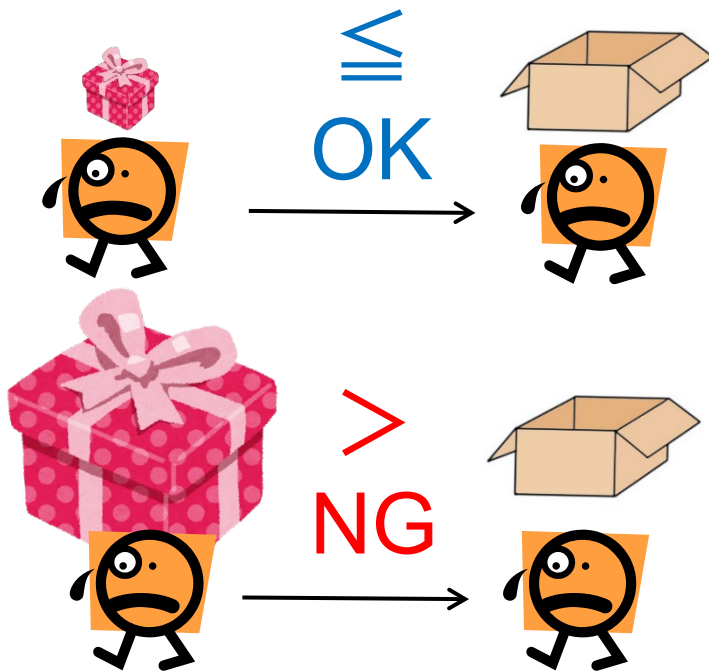
If message is larger than expected, it's an error
(the program aborts)

If the message size is UNKNOWN beforehand, the receiver should prepare enough memory
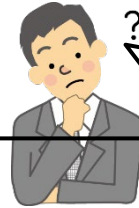
Case of "diffusion" Sample
related to [M1]

An example of diffusion phenomena:

The ink spreads gradually, and finally the density becomes uniform   (Figure by Prof. T. Aoki)
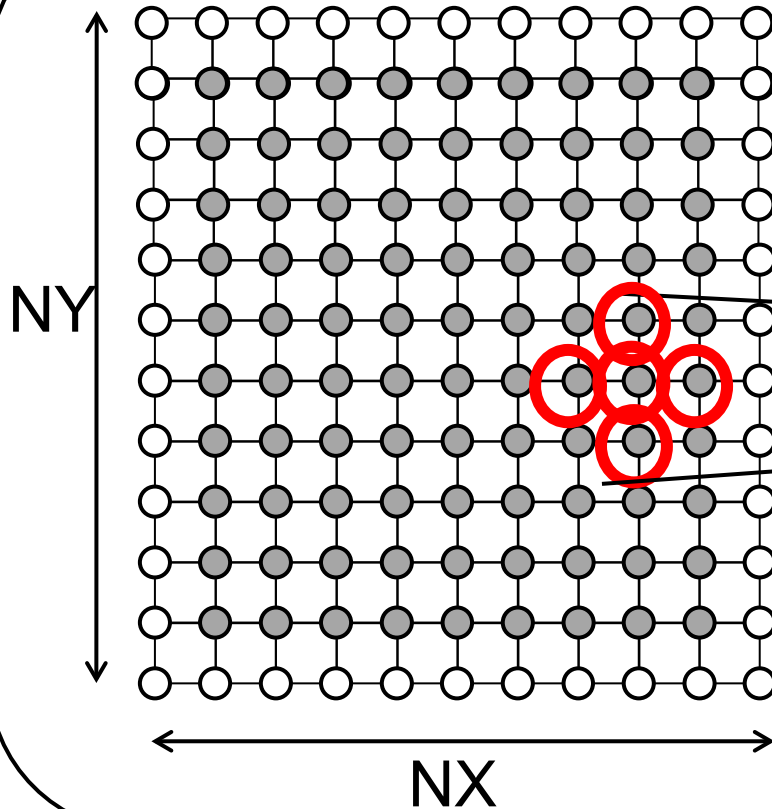
Available at /gs/hs1/tga-ppcomp/23/diffusion/

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

You can use /gs/hs1/tga-ppcomp/23/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
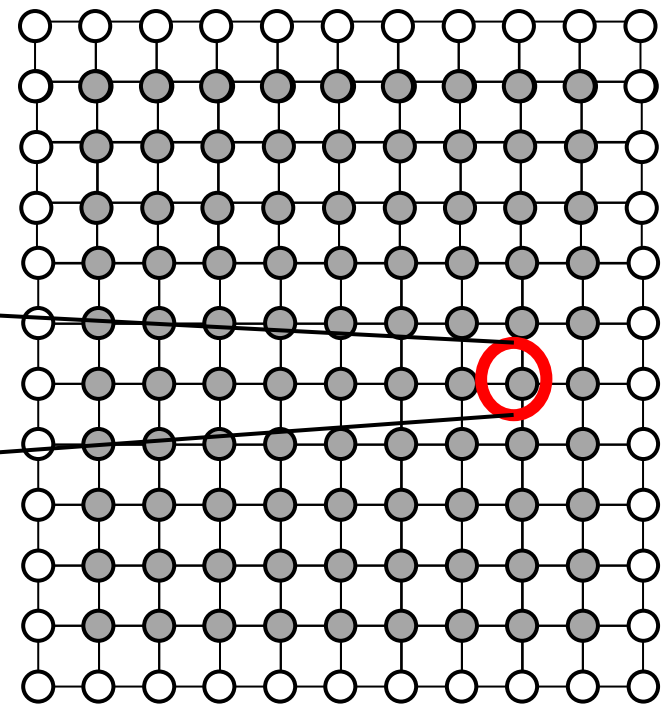
How can we distribute data?

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:

- 2 x 2D arrays ➔ float data[2][NY][NX];

On MPI:

[Algorithm] Same as above

- Each process computes its part in the space

[Data] 2 x 2D arrays are divided among processes

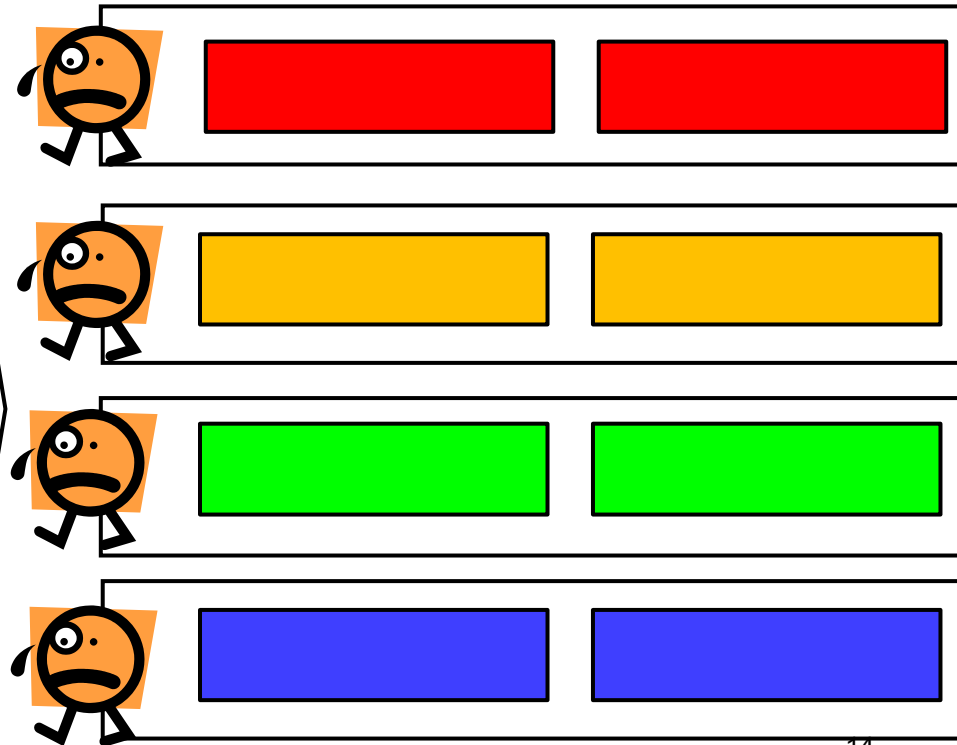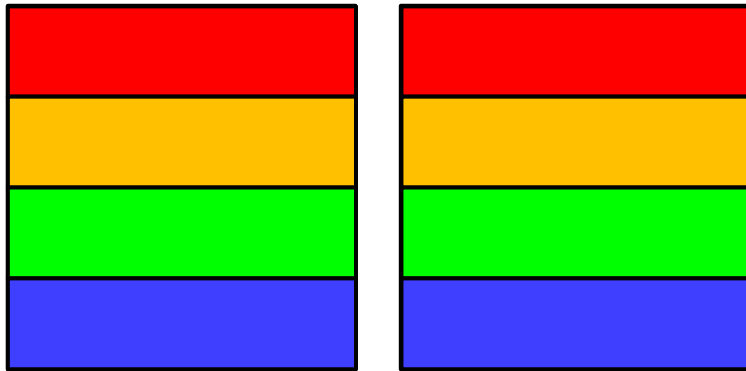- Each process has its own part of arrays

Considering Data Distribution (1)

2 x 2D arrays are divided among
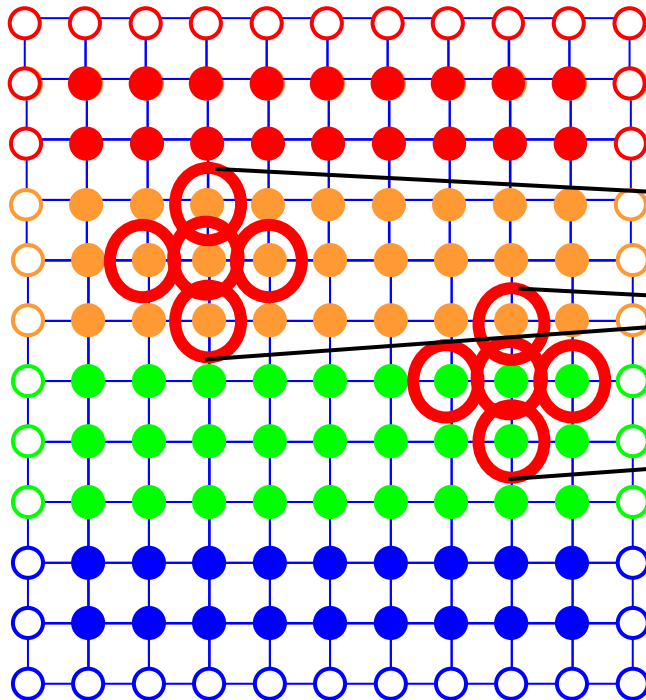P processes (in this case, horizontally)

※ A color = a process

This looks ok, but will be improved next

Each array size is (roughly)
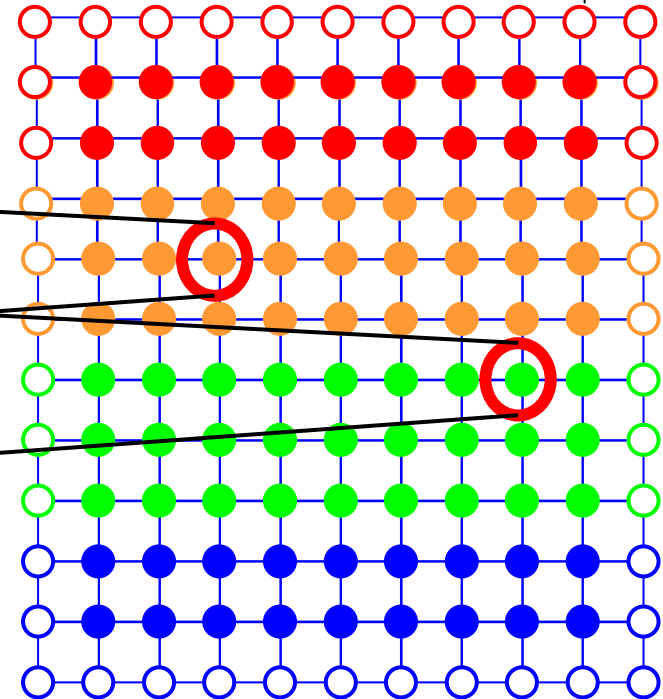NX x (NY/P)

14

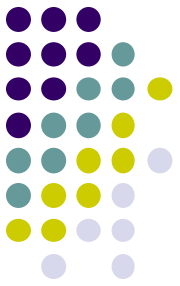# Improving Data Distribution (1)

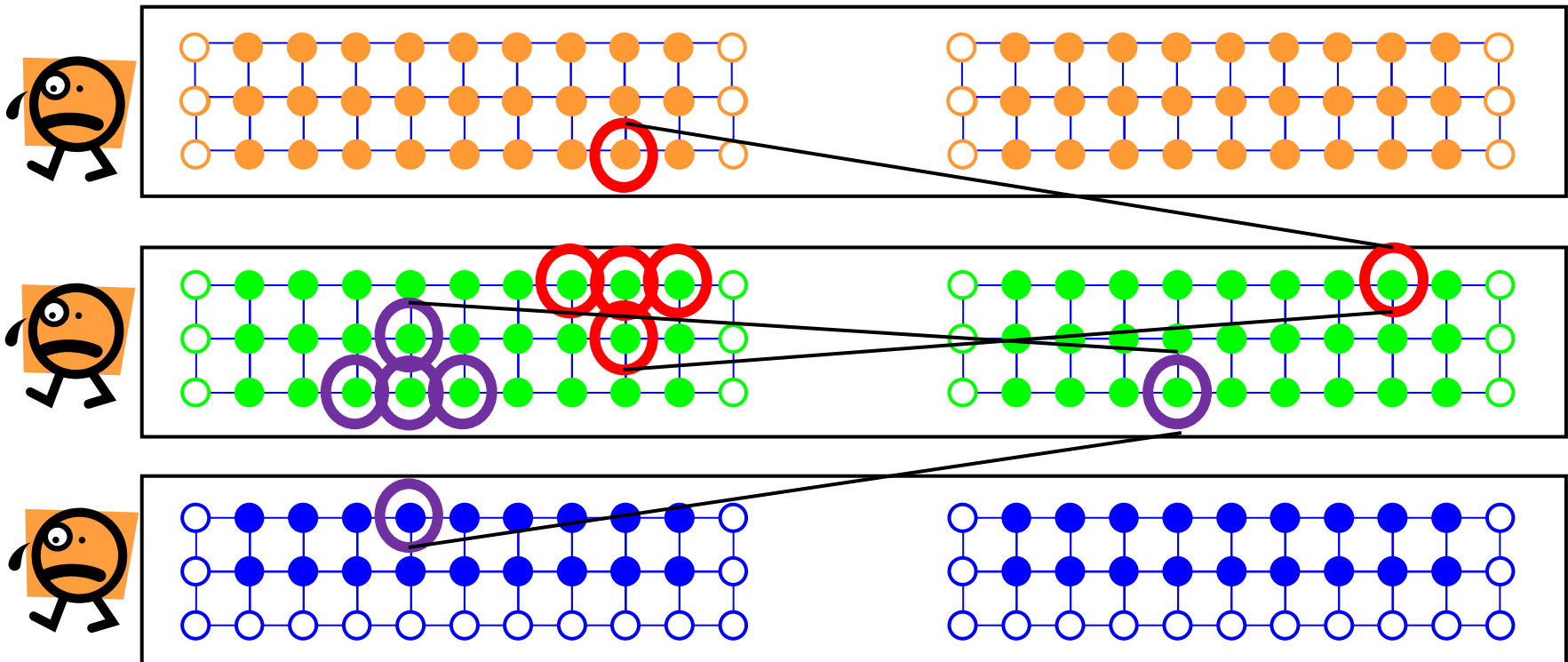An Array for "even" steps   An Array for "odd" steps

- Let's remember computation of each point
→ 5 points are read and 1 point is written

Improving Data Distribution (2)

- What's wrong with the simple distribution?

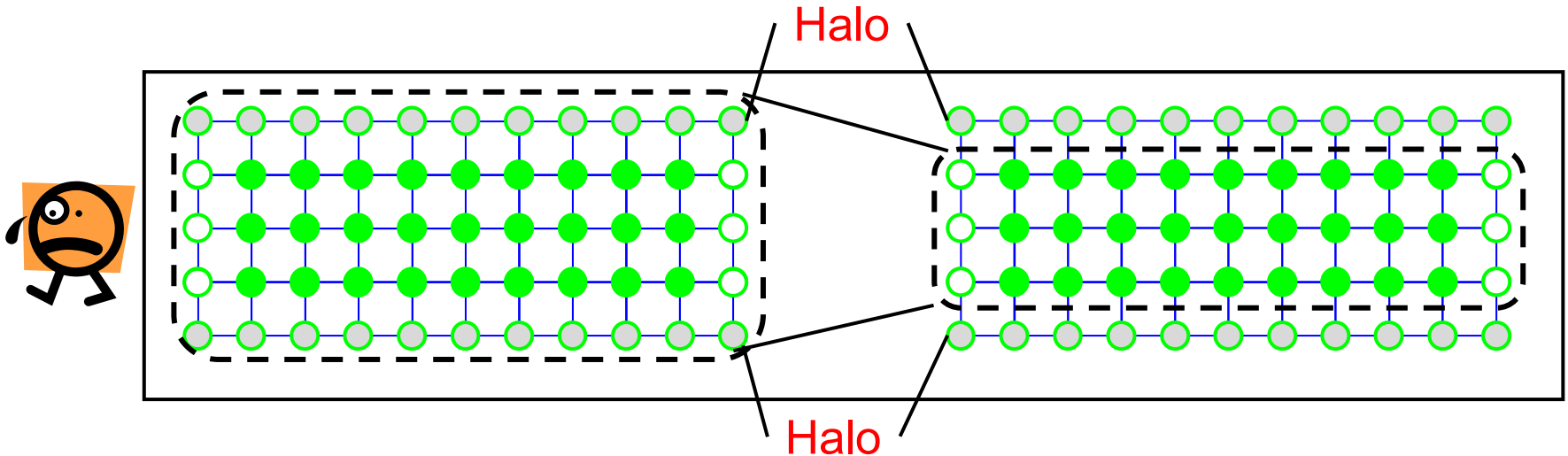

Computation requires data in other processes

→ Message passing is required

We need memory region for received data!

A Technique in Stencil: Introducing "Halo" Region

● In stencil computation, it is a good idea to make additional rows to arrays

→ called "Halo" region

Halo

Halo

Each array size is (roughly) NX x (NY/P + 2)

Halo regions are used to receive outside border data from neighbor processes

Using "Halo" Region

Each time step consists of:
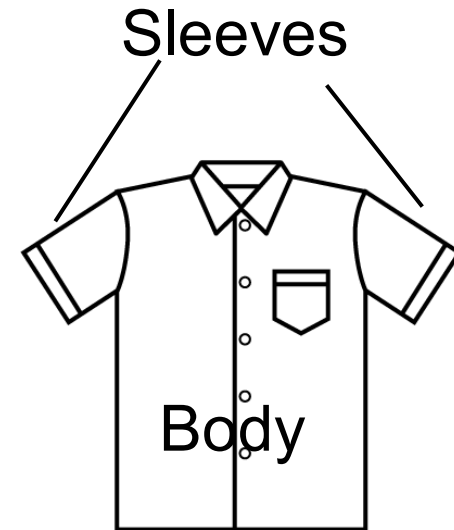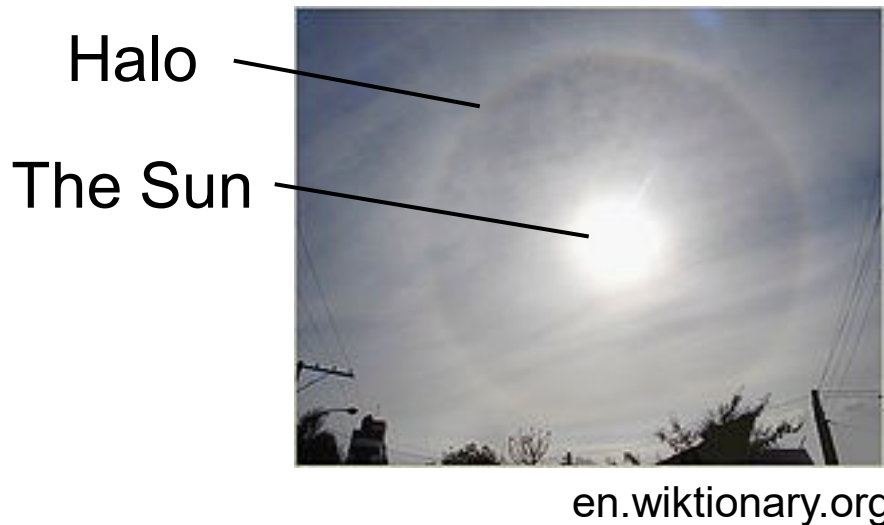
(1) Communication: Recv data and store into "halo" region
   - Also neighbor processes need "my" data

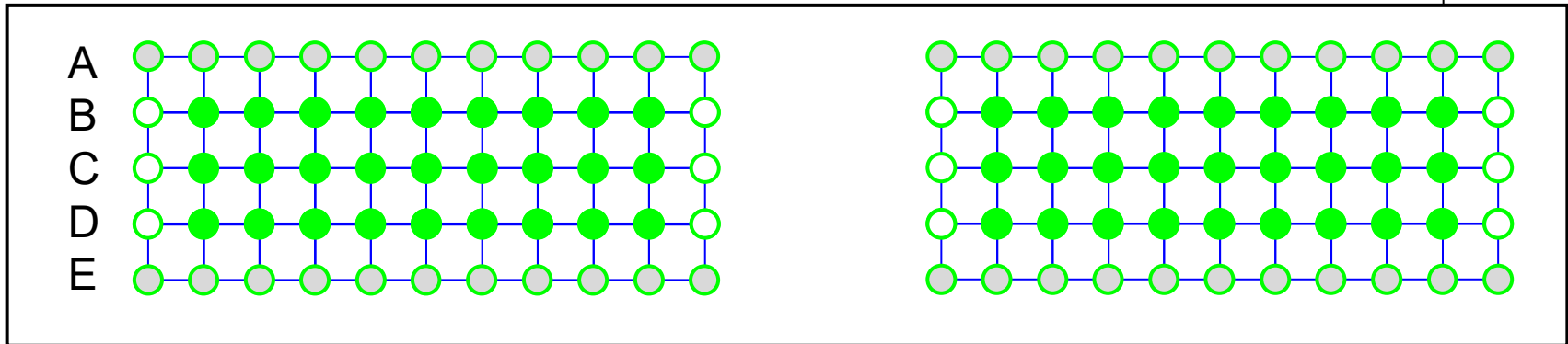(2) Computation: Old data at time t (including "halo")
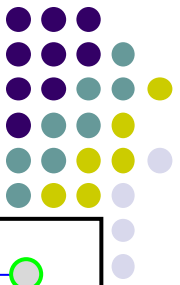   → New data at time t+1

(1)Comm

Halo

(2) Comp

Halo

(1)Comm

The name of "Halo" Region

Halo

The Sun

en.wiktionary.org

Sleeves

Body

© dak

"Halo regions" are sometimes called "sleeve regions" or "overlap regions"

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
→ Explained next

A Sample with Neighbor Communication

When considering neighbor communication, we have to avoid deadlock (a serious bug)!

A sample is available at /gs/hs1/tga-ppcomp/23/neicomm-mpi

Execution: mpiexec -n [P] ./neicomm

(1) Each process prepares its local data

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

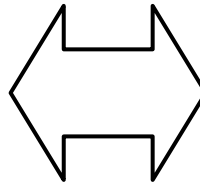| Rank 0	Rank 1	Rank 2	Rank 3
0 1	0 1 4	1 4 9	4 9

leftdata   mydata   rightdata
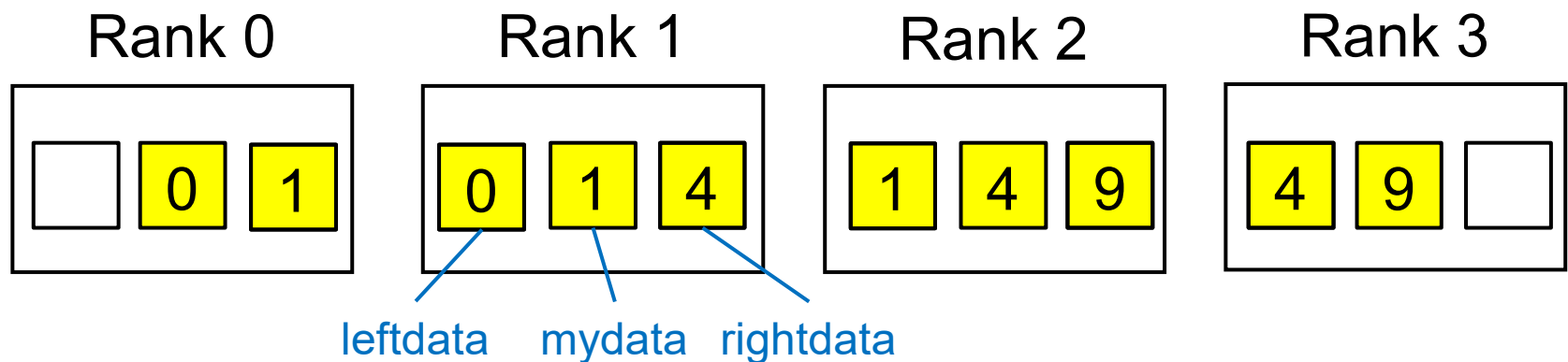
Behavior of neicomm-mpi Sample

Unsafe version ☹

When neicomm_unsafe() is used

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

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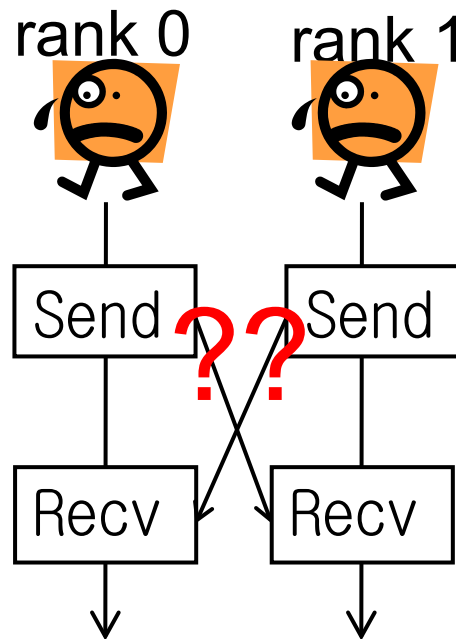
Safe version ☺

When neicomm_safe() is used

```
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
```

Deadlock in MPI

- This case "deadlocks" with 2 processes. Why?

One of reasons is usual MPI_Send and MPI_Recv uses "blocking communication"
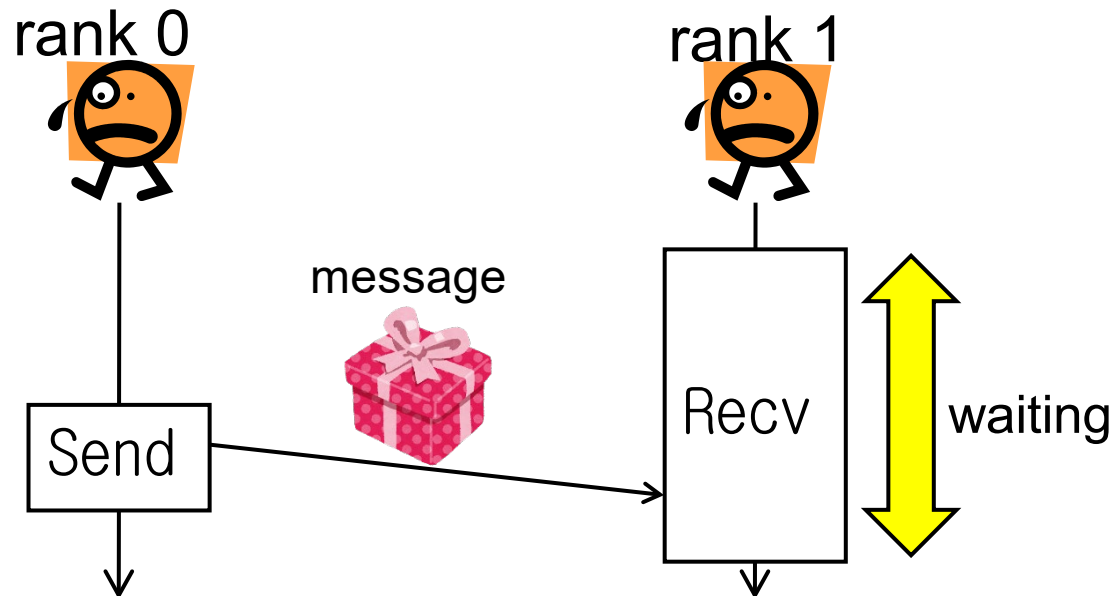- a process waits until "some event"

23

Behavior of MPI_Recv()

- 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)

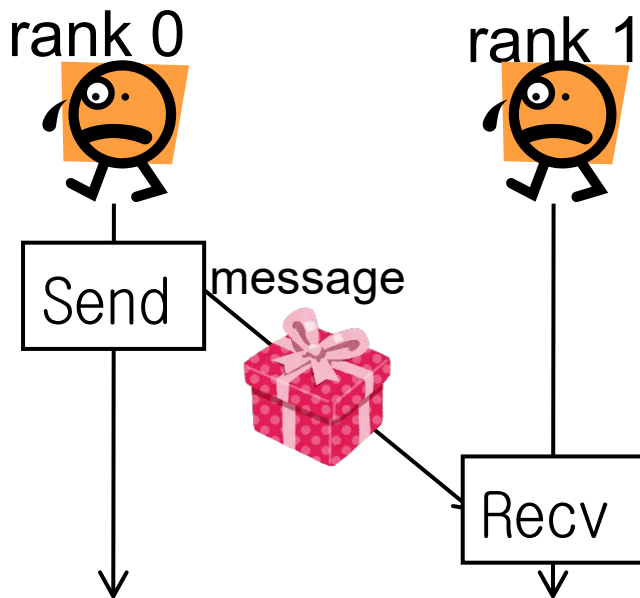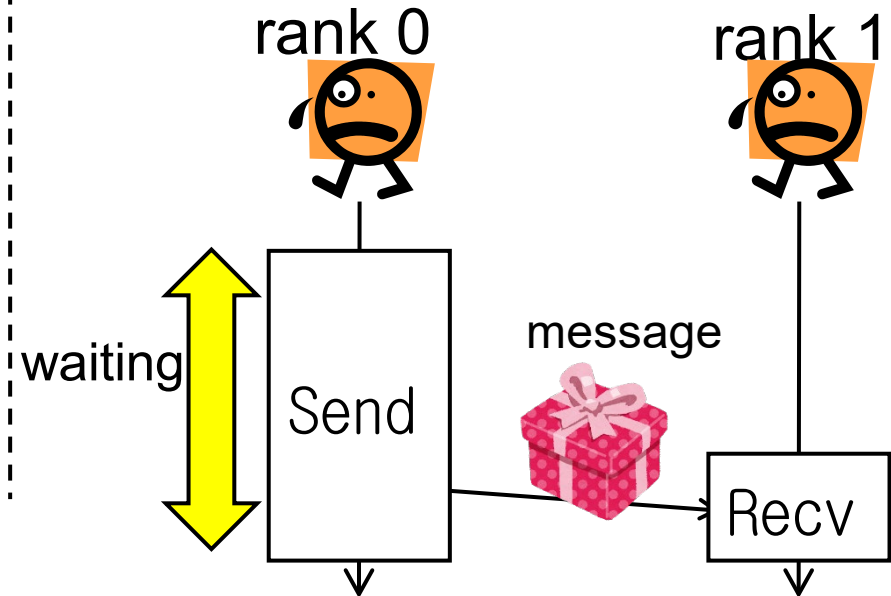rank 0   rank 1

message

Send → Recv   waiting

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)
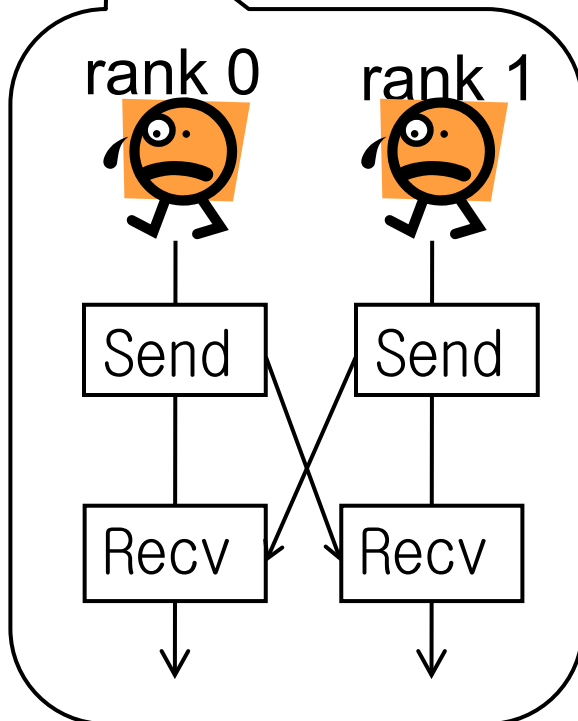

Which occurs?
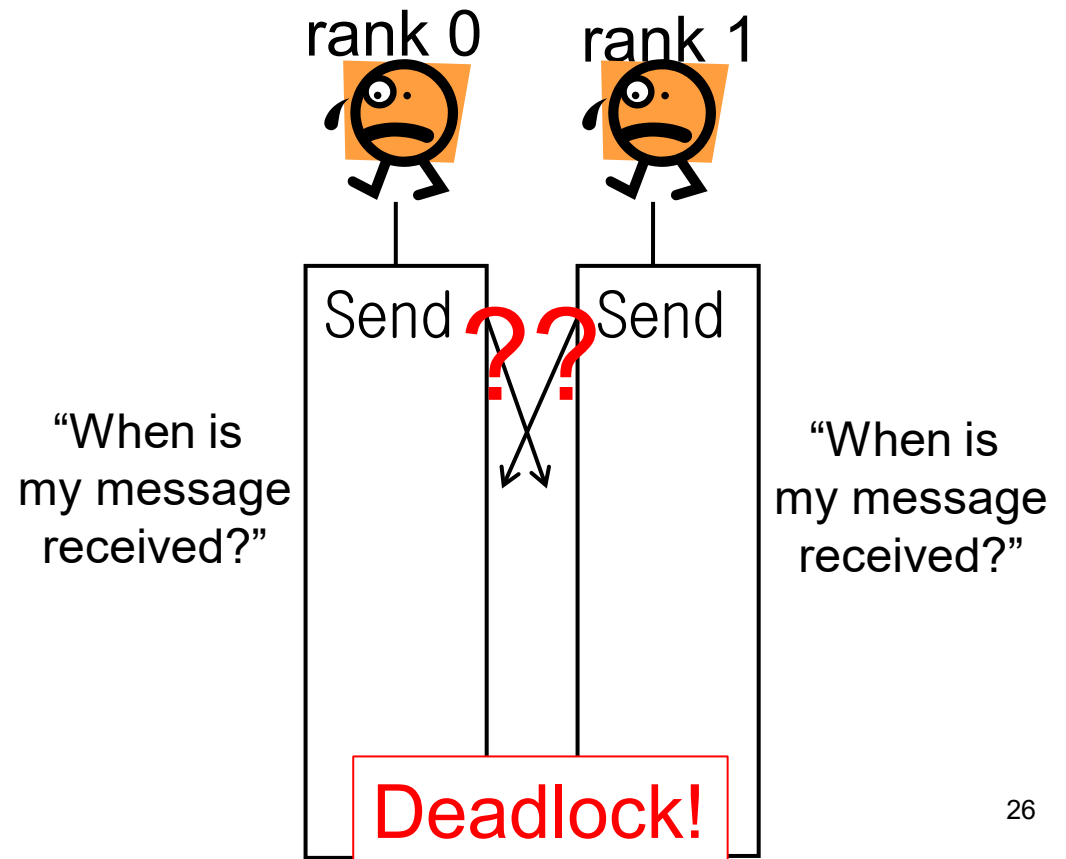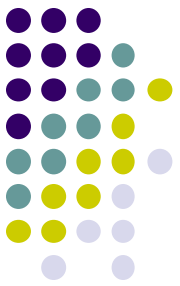It depends on MPI library, message size, etc. ➜ Unknown

Deadlock Happens

An Approach to Avoid Deadlock

- To change order of MPI_Send and MPI_Recv
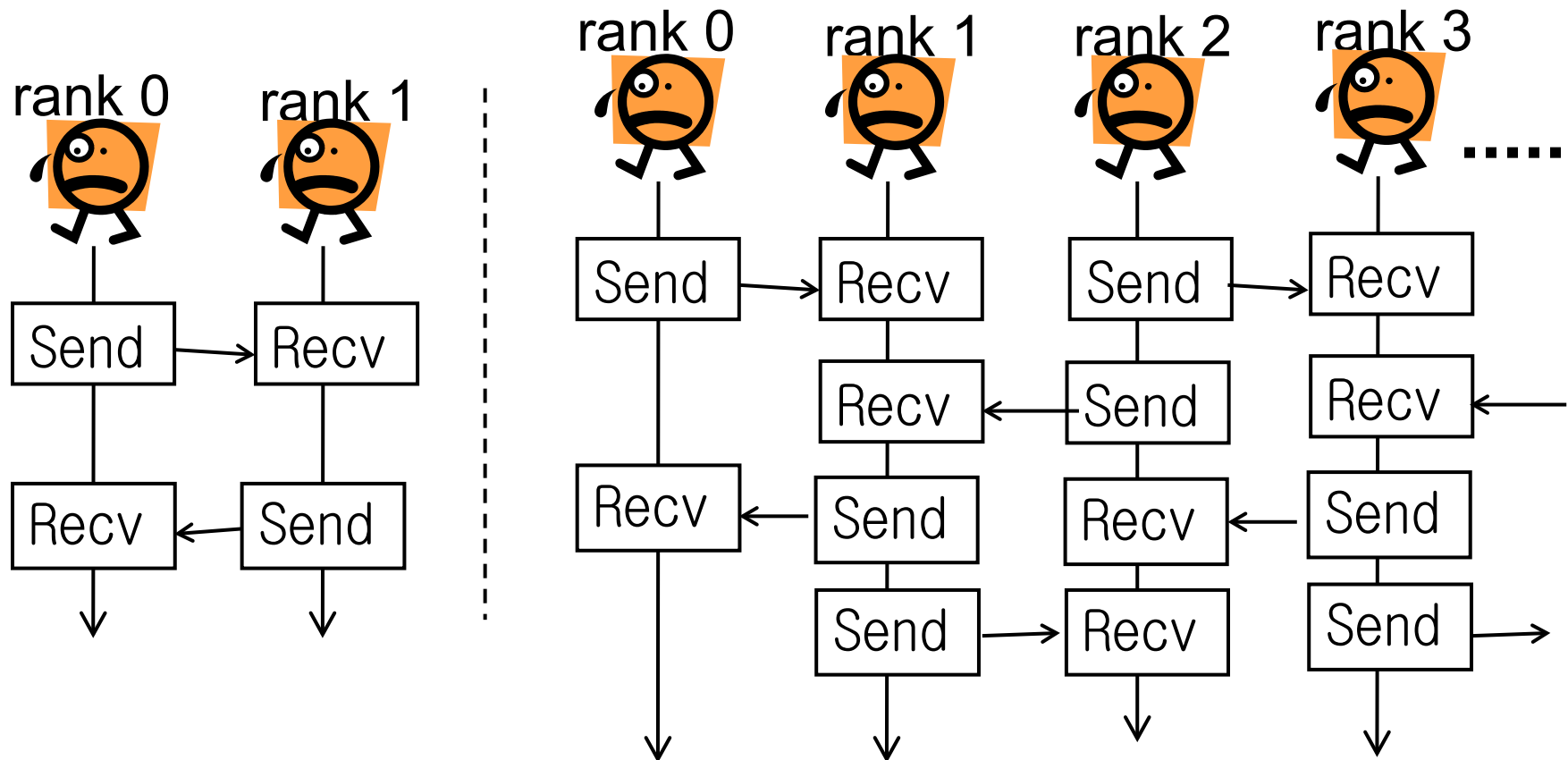
Even processes and odd processes work differently

Another Approach: Using 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()

rank 0   rank 1

Irecv ← immediately returns

[yellow box] ← Process can do something

message

Send   Wait   wait for message arrival
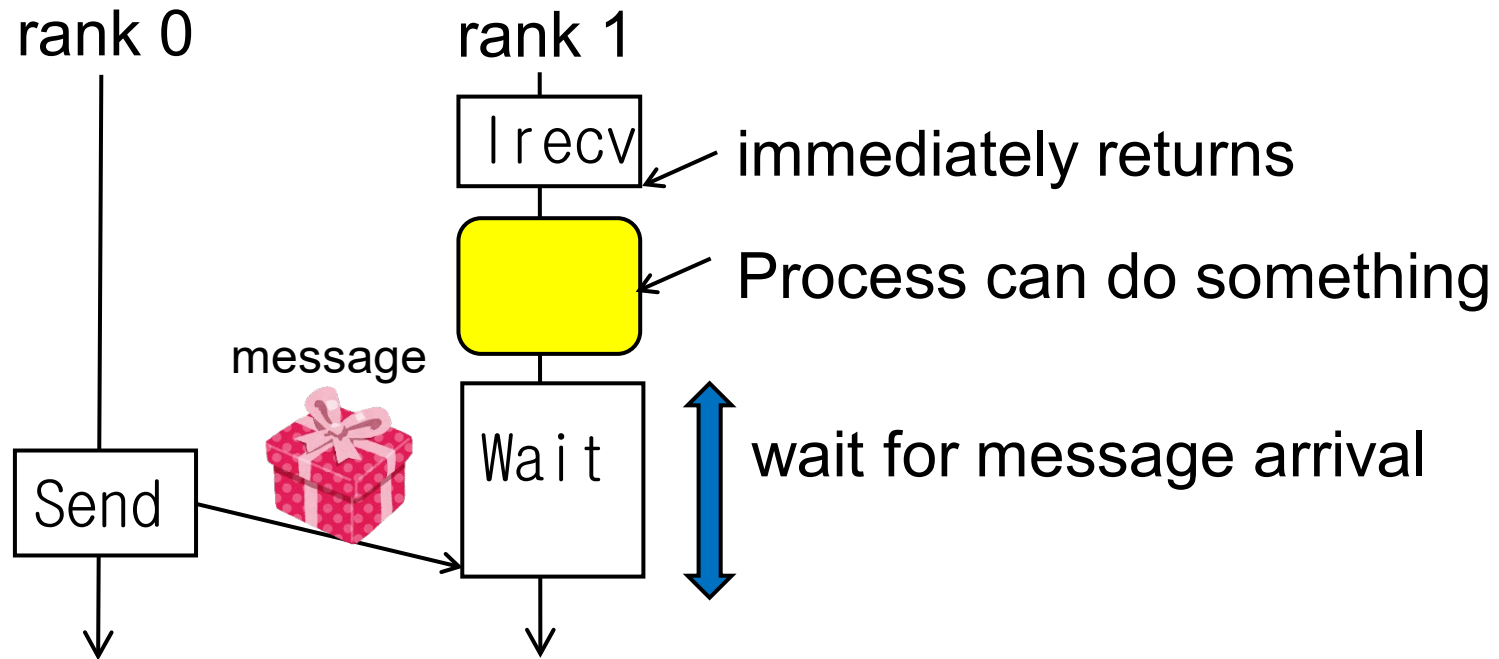
Non-Blocking Receive

```
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
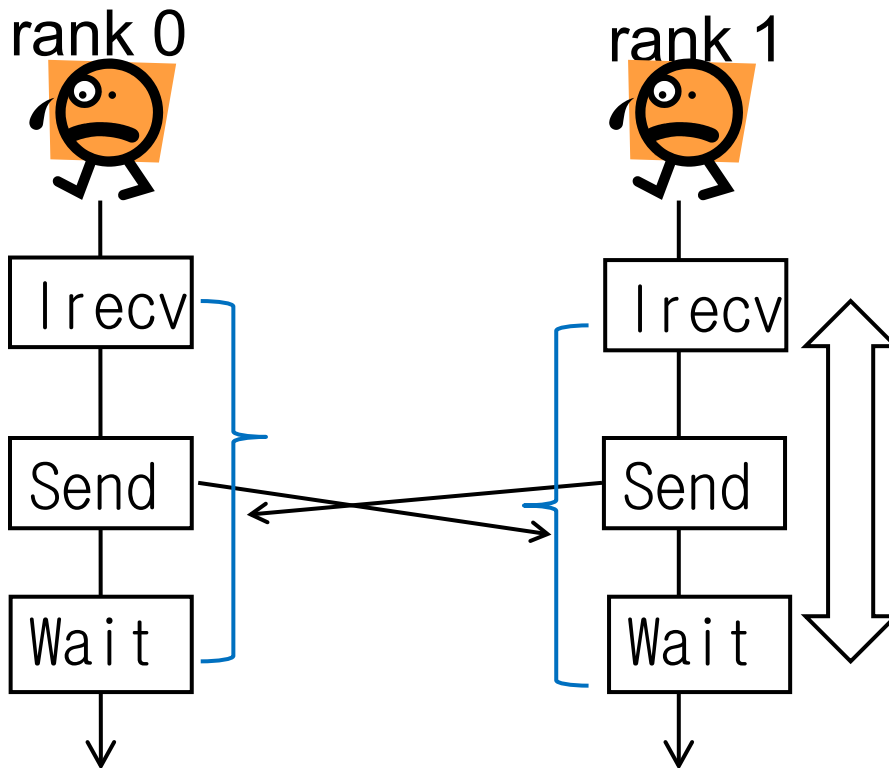
MPI_Request is like a "ticket" for the communication

Avoiding Deadlock
with Non-Blocking Communication

On each process, Recv is divided into Irecv & Wait

rank 0   rank 1

```
Irecv        Irecv

Send         Send

Wait         Wait
```
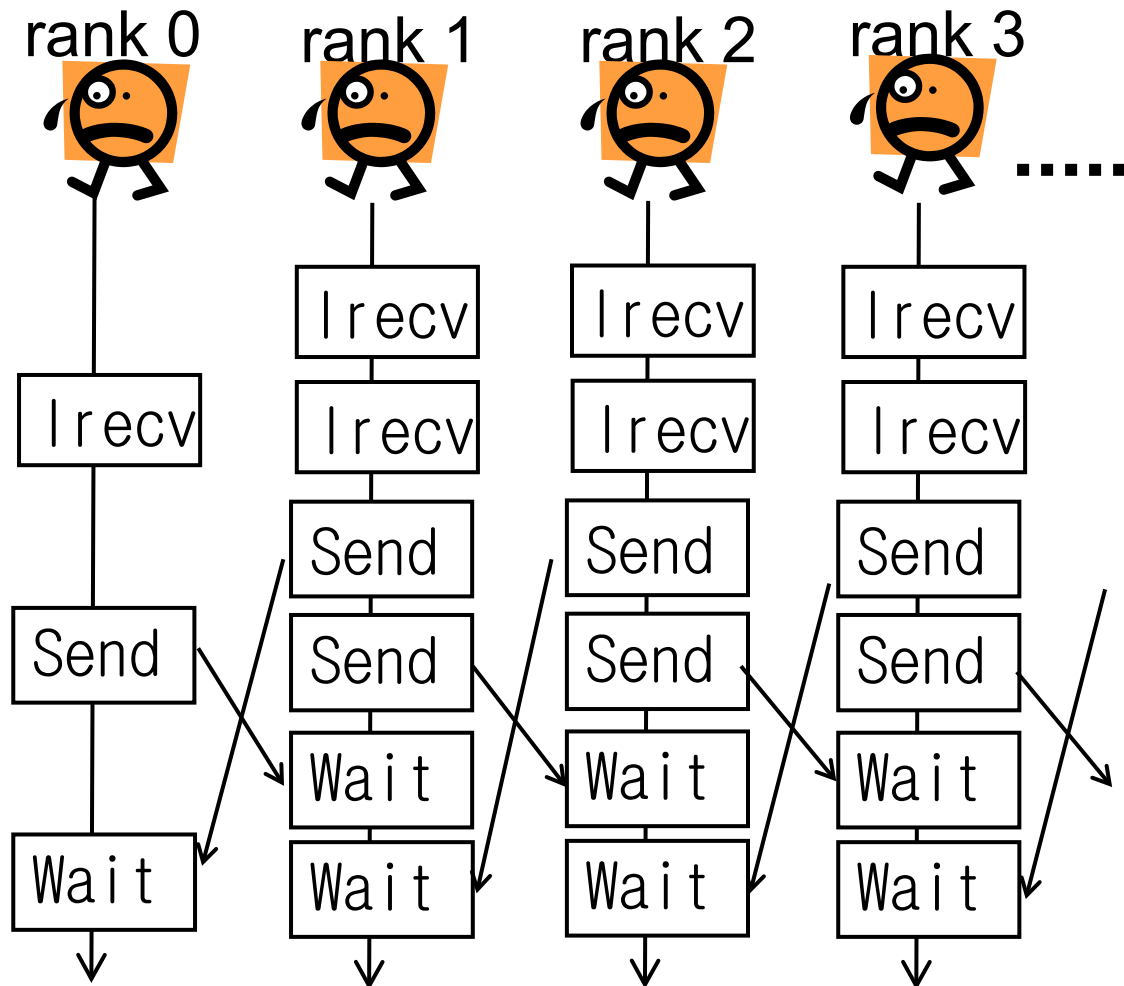
What's difference than before?

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

rank 0   rank 1   rank 2   rank 3

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
    - 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?

Assignments in MPI Part (Abstract)

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

Due date: June 12 (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 3-1 slides

Next Class

- MPI (3)
  - Improvement of "matrix multiply" sample
    - Related to [M2]
  - Group Communication

- Planned schedule
  - May 25: cancelled (休講) & Due for Part2 assignment
  - May 29: Part 3 (3)
  - June 1: Part 3 (4)