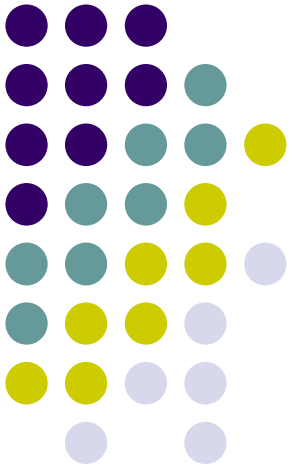
2017
Practical Parallel Computing (実践的並列コンピューティング) No. 8

Distributed Memory Parallel Programming with MPI (2)

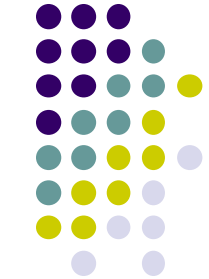


Toshio Endo

School of Computing & GSIC

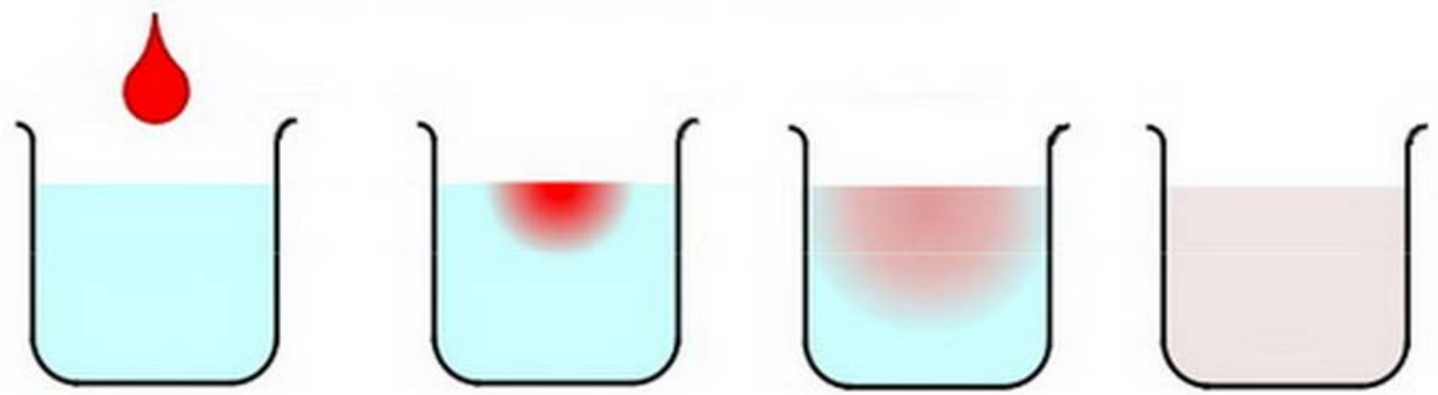
endo@is.titech.ac.jp

“diffusion” Sample Program (1) (Revisited)



An example of diffusion phenomena:

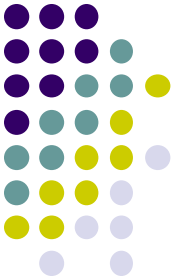
- Pour a drop of ink into a water glass



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

- Density of ink in each point vary according to time → Simulated by computers
- Stencil computation

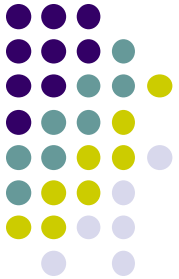
“diffusion” Sample Program (2) (Revisited)



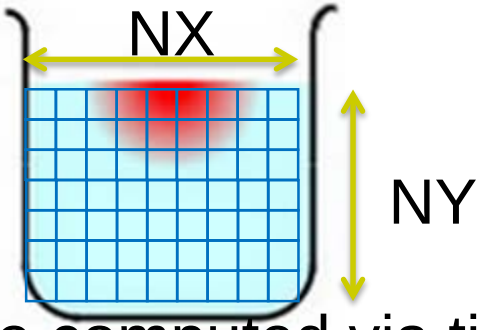
Available at ~endo-t-ac/ppcomp/17/diffusion/

- Execution：./diffusion [nt]
- nt: Number of time steps
- nx, ny: Space grid size
  - nx=8192, ny=8192 (Fixed. See the code)
  - How can we make them variables? (See mm sample)
- Compute Complexity：O(nx × ny × nt)

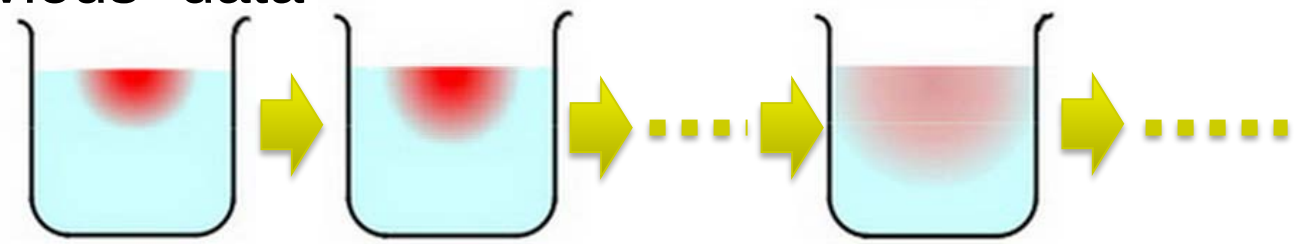
Data Structures in diffusion (Revisited)



- Space to be simulated are divided into grids, and expressed by arrays (2D in this sample)



- Array elements are computed via timestep, by using “previous” data

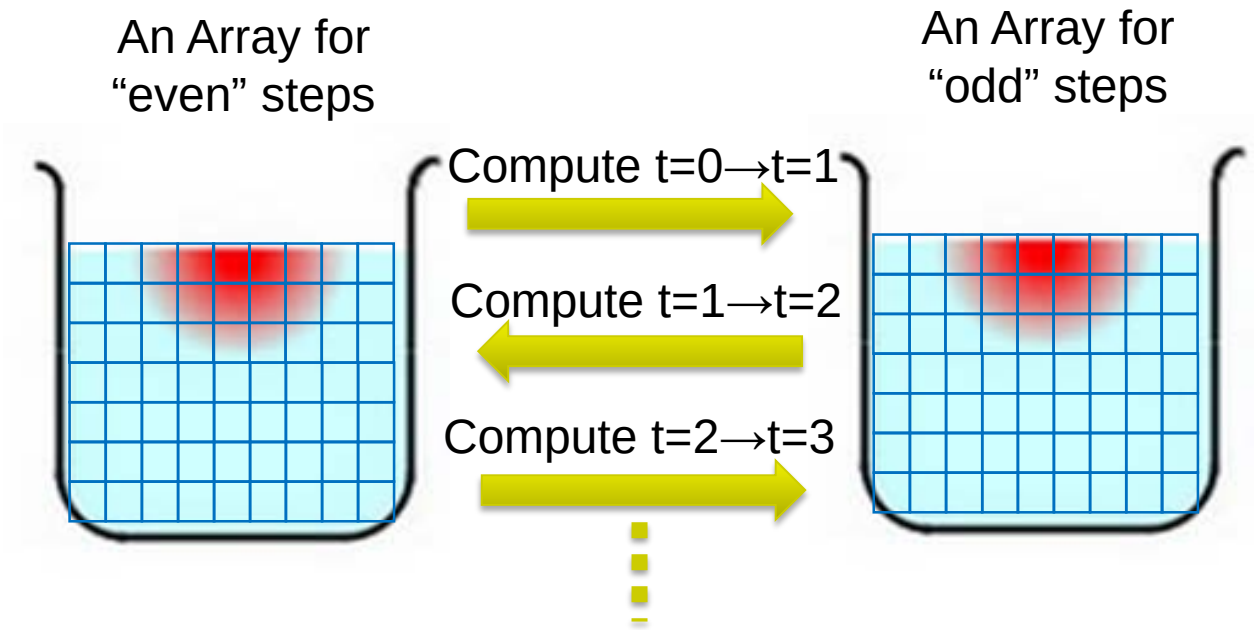


Time step t=0 t=1 t=20

Double Buffering Technique (Revisited)

- A simple way is to make arrays for all time steps, but it consumes too much memory!

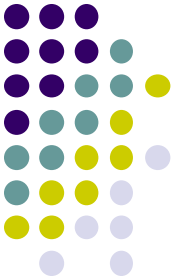
→ It is sufficient to have “current” array and “previous” array. “Double buffers” are used for many times



※ Sample program uses a global variables

float data[2][NY][NX];

How Do We Parallelize "diffusion" Sample?



Parallelization method with OpenMP:

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

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

[Data] Data structure is same as sequential version

With MPI:

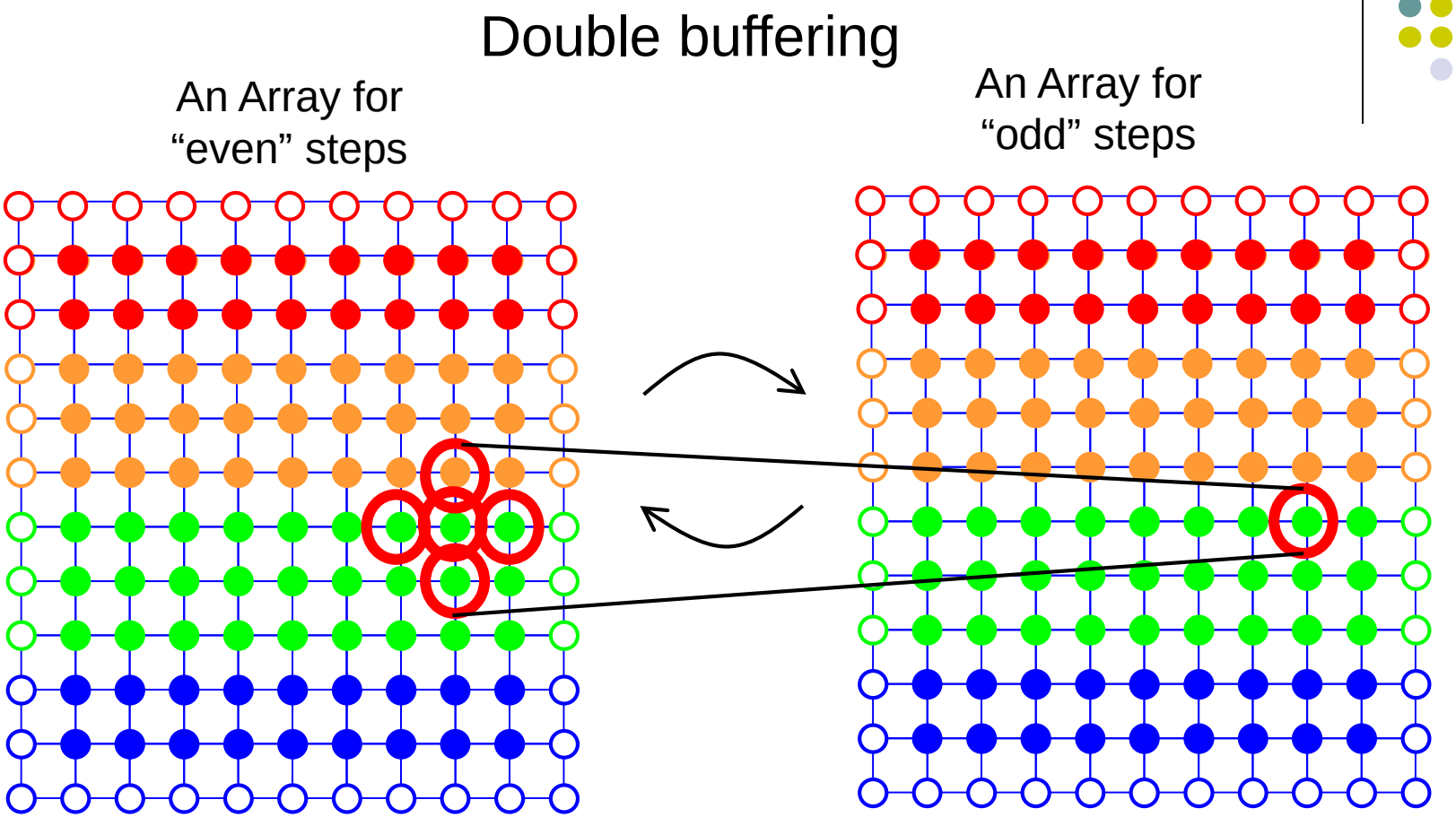
[Algorithm] Same policy as OpenMP version

- Each process computes its part in the space

[Data] Arrays are divided among processes

- Each process has its own part of arrays

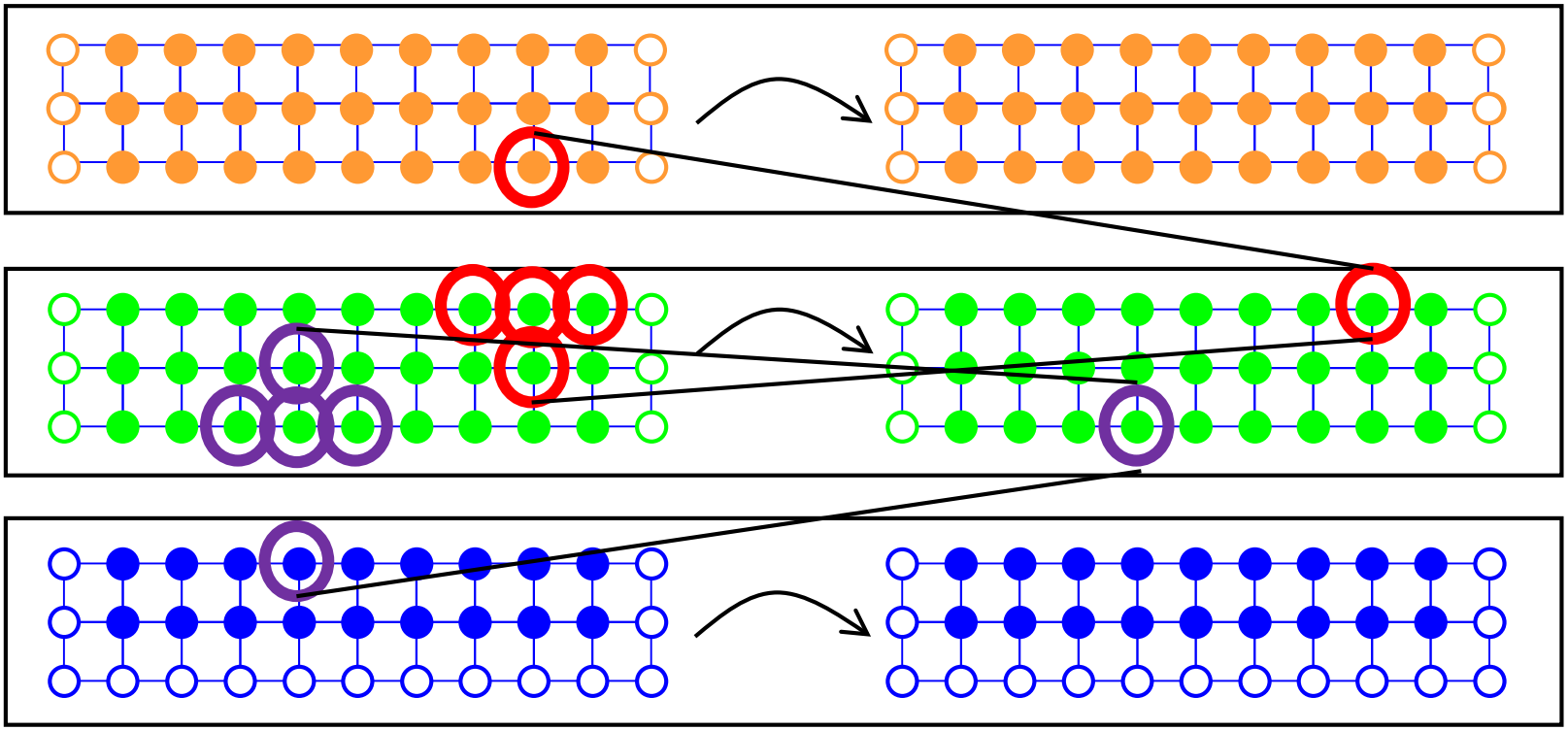
Parallelize “diffusion” Sample



- In diffusion, computation of a new point requires 5 old points (5-point stencil)

Considering Data Distribution

- A simple distribution is like:



Computation requires data in other processes

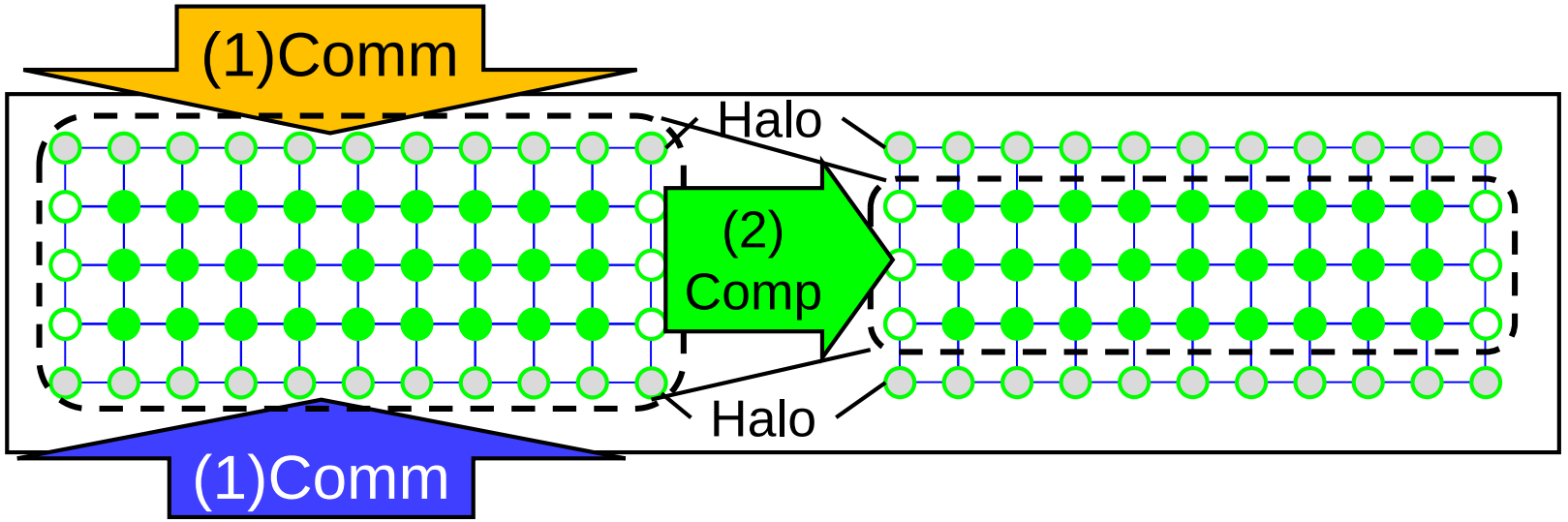
→ Communication is required

So, where should received data be put?

Introducing “Halo” Region

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

→ called “Halo” region

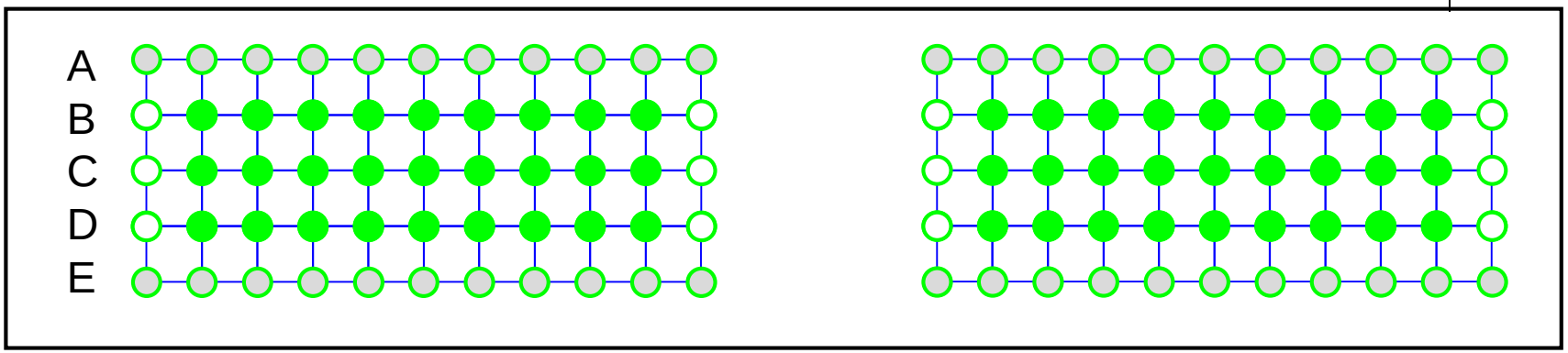


Each time step consists of:

(1) Communication: Recv data and store into “halo” region
- Also neighbor processes need “my” data

(2) Computation: Old data (including “halo”) → New data

Overview of MPI “diffusion” (Still Unsafe)



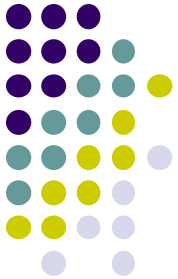

```
for (t = 0; t < nt; t++) {
  Send B to rank-1, Send D to rank+1
  Recv A from rank-1, Recv E from rank-1

  Computes points in rows B-D
  Switch old and new arrays
}
```

(1) Communication

(2) Computation

This version is still unsafe, because this may cause deadlock

Sample for Neighbor Communication

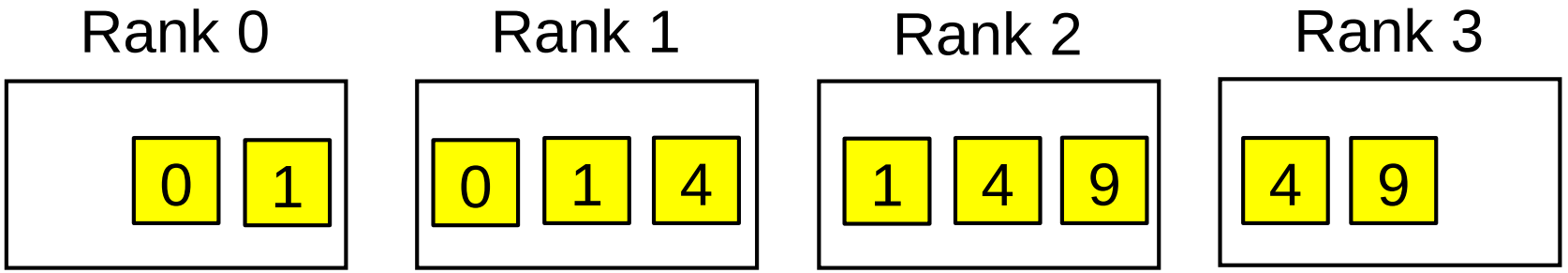


Available at ~endo-t-ac/ppcomp/17/neicomm/

Execution: mpirun –np [np] ./neicomm

(1) Each process produces a single value (rank^2 here)

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

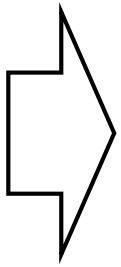


Neighbor Communication

Unsafe version ☹

neicomm_unsafe()
in neicomm sample

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



Safe version ☺

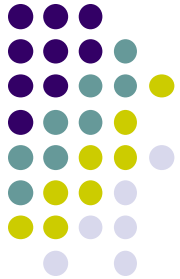
neicomm_safe()
in neicomm sample

```
Start to send to rank-1
Start to send to rank+1
Recv from rank-1
Recv from rank-1
Finish to send to rank-1
Finish to send to rank+1
```

※ It requires a long story to see the reason of deadlock, so omitted here

Hint: Not only MPI_Recv, but MPI_Send is “blocking” communication if message size is very large

Non-Blocking Communication



- Non-blocking communication: starts a communication (send or receive), but does not wait for its completion
  - cf) MPI_Recv is blocking communication, since it waits for message arrival
- Process should wait for the 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
MPI_Wait(&req, &stat);     ←wait for completion
```

MPI_Irecv: starts receiving, but it returns Immediately

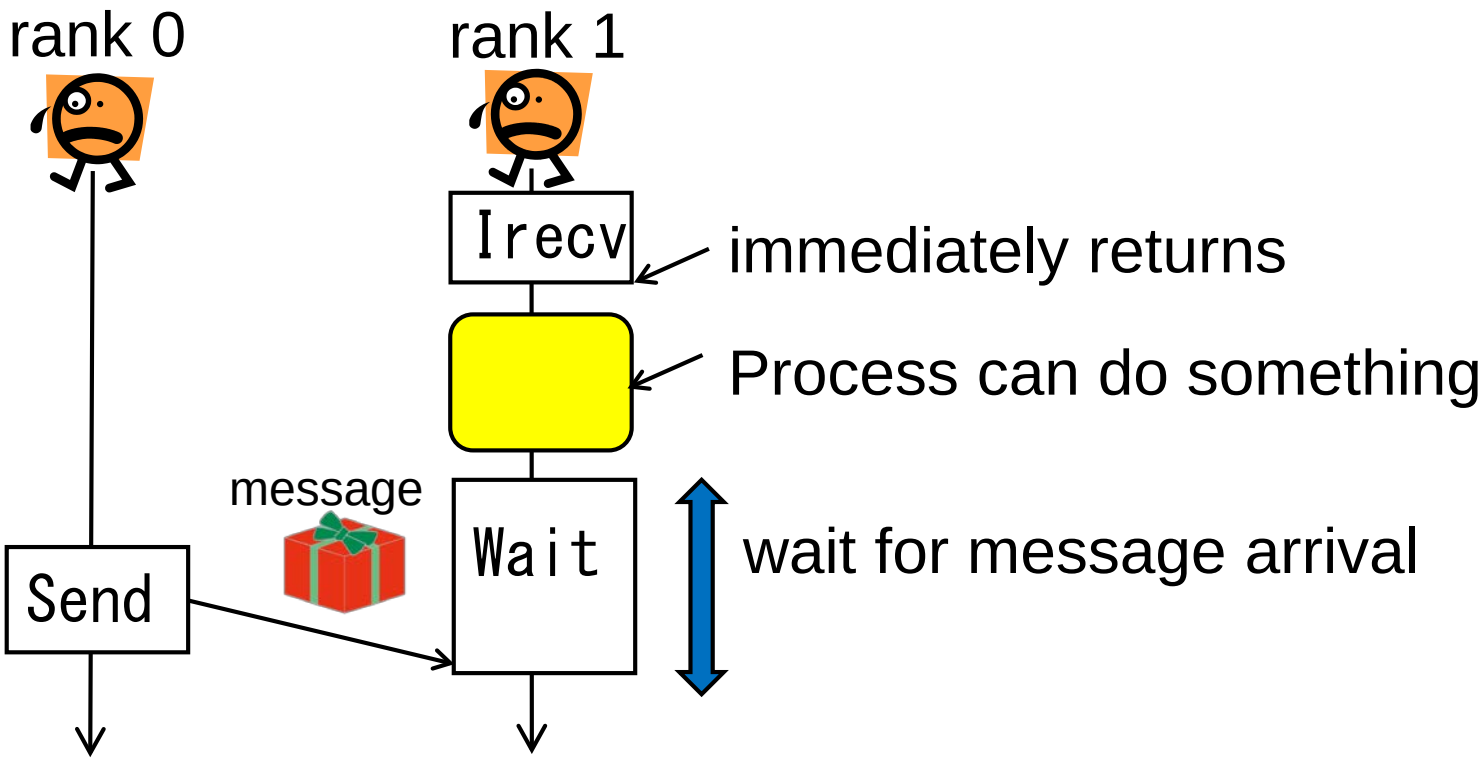
MPI_Wait: wait for message arrival

MPI_Request looks like a “ticket” for the communication

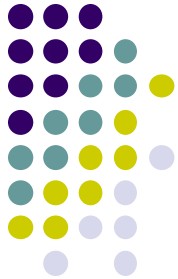
Behavior of MPI_Irecv

- MPI_Irecv itself immediately returns
- Program can use received data after MPI_Wait

※ MPI_Recv = MPI_Irecv + MPI_Wait



Non-Blocking Send



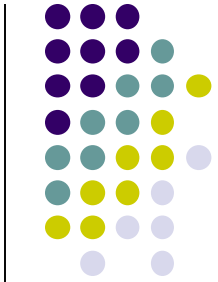
```
MPI_Send(buf, n, type, dest, tag, comm);
```

```
MPI_Status stat;
MPI_Request req;
MPI_Isend(buf, n, type, dest, tag, comm, &req); ←start send
MPI_Wait(&req, &stat);      ←wait for completion
```

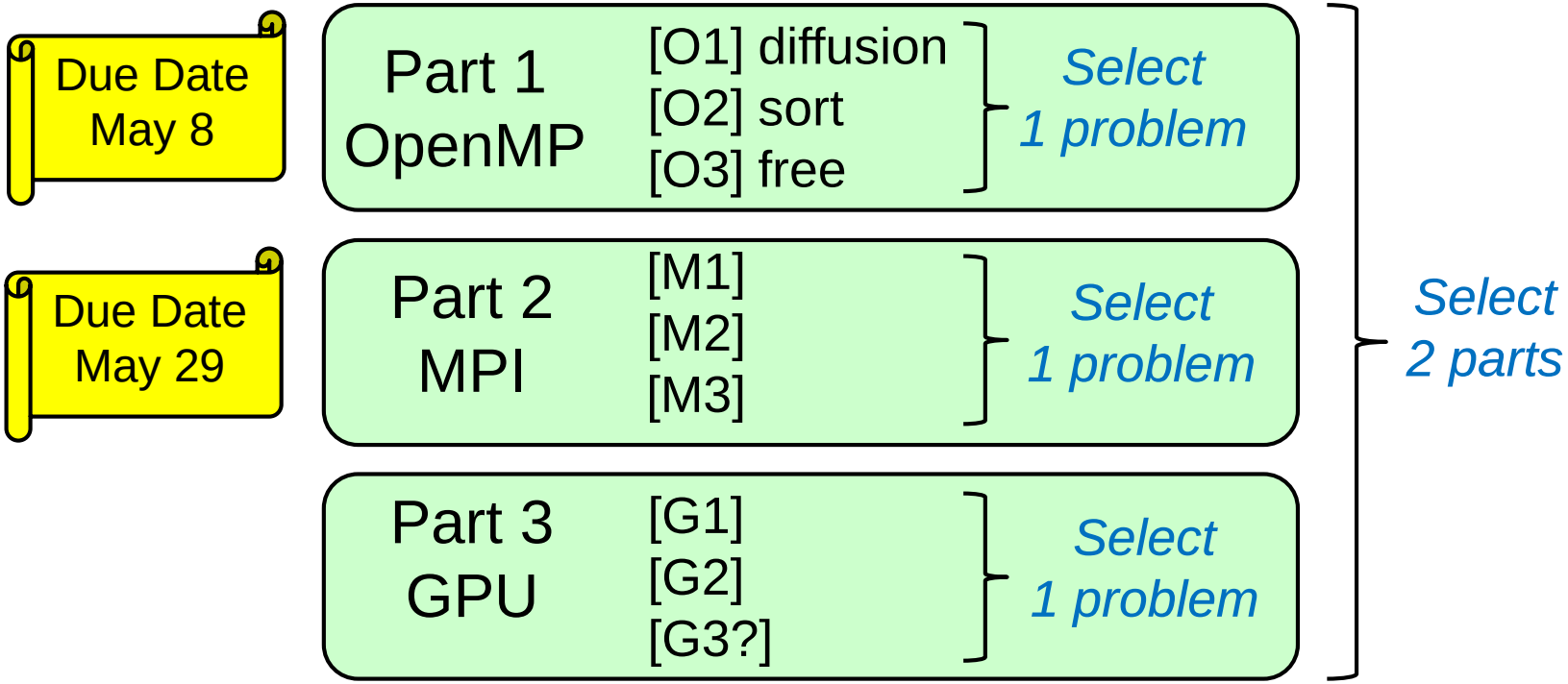
MPI_Isend: starts sending, but it returns Immediately

MPI_Wait can be also used for waiting completion

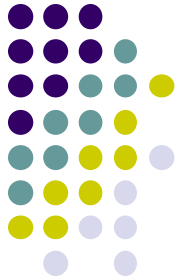
Assignments in this Course



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



Assignments in MPI Part (Abstract)



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

Due date: May 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 Apr 27 slides or OCW-i.

Next Class

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