

2018

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

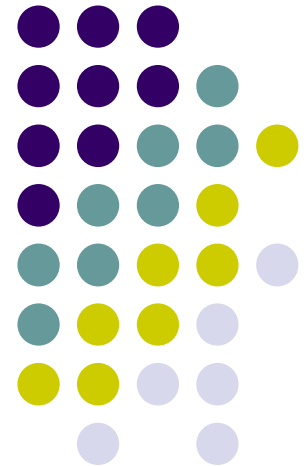
No. 8

Distributed Memory Parallel Programming with MPI (2)

Toshio Endo

School of Computing & GSIC

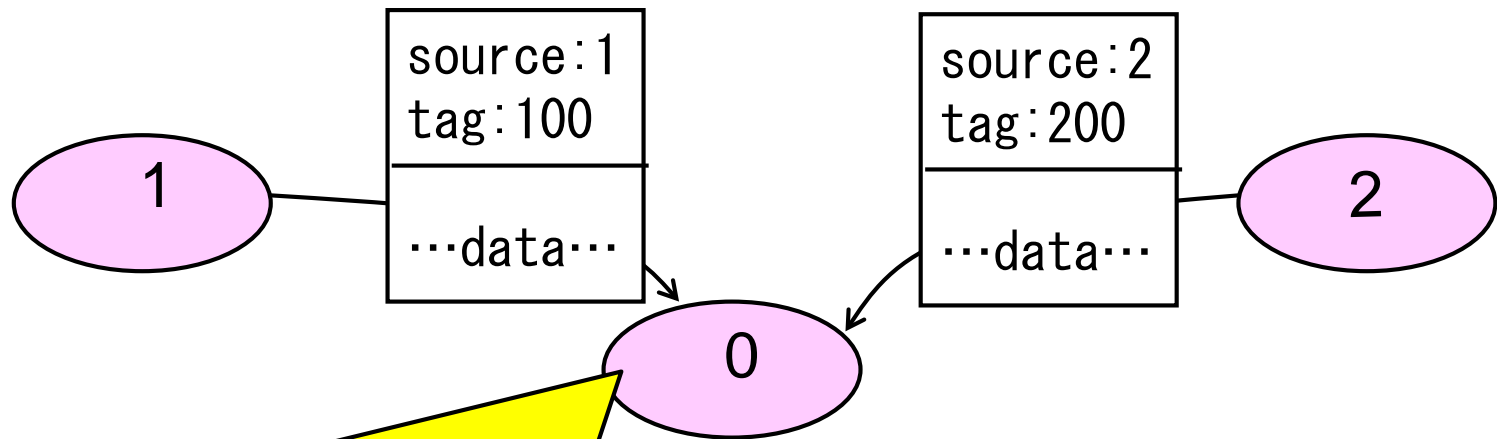
endo@is.titech.ac.jp



Notes on MPI_Recv: Message Matching (1)



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



I only want a message with tag 200 from 2 !

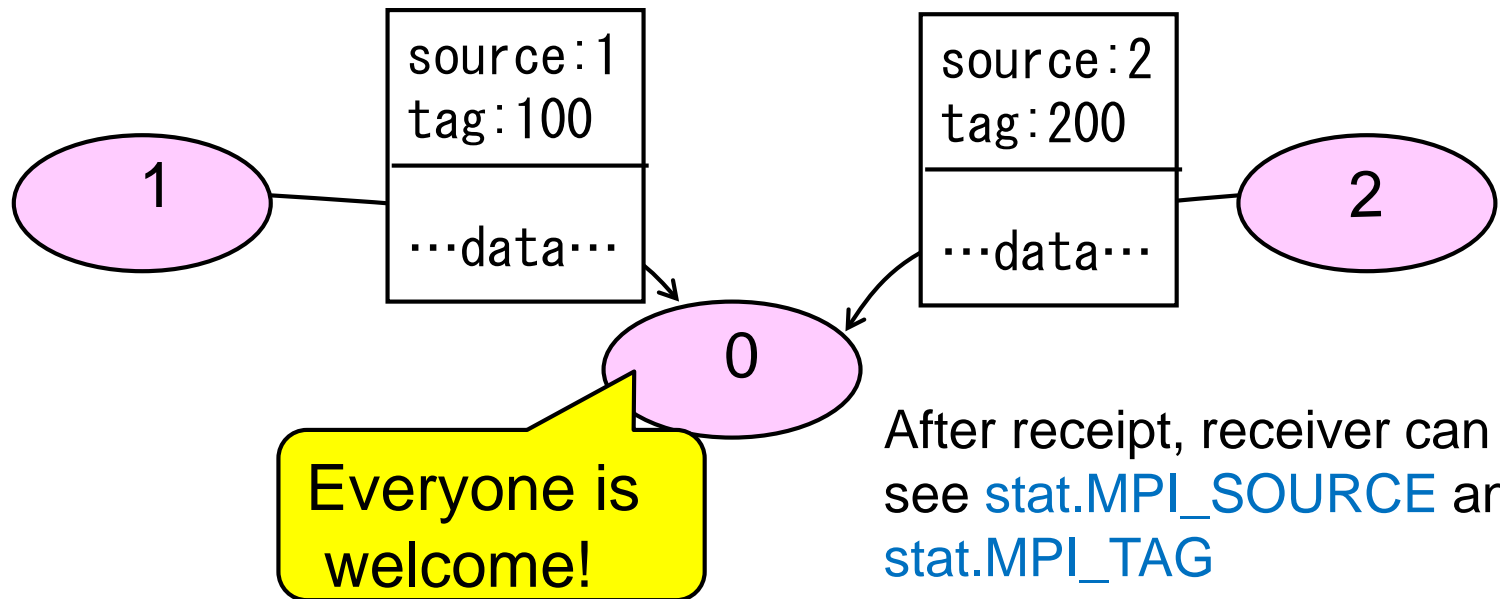
- Receiver specifies “source” and “tag” that it wants to receive
- Generally, several messages may arrive indefinite order
- The message that **matches the condition** is delivered
- Other messages should be received by other MPI_Recv calls

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);
```

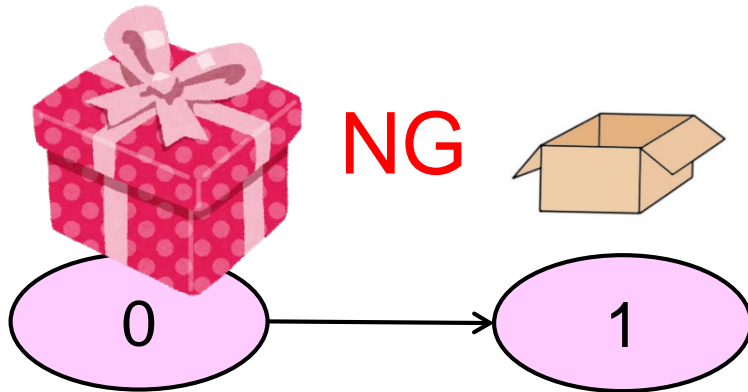


Notes on MPI_Recv:

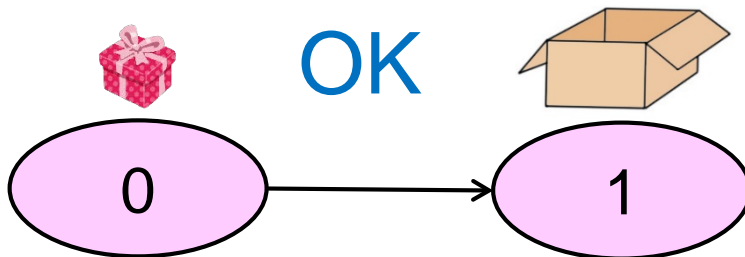
What If Message Size is Unmatched



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



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



If message is **smaller** than expected, it's **ok**

→ Receiver can know the actual size by
`MPI_Get_Count(&stat, MPI_INT, &s);`

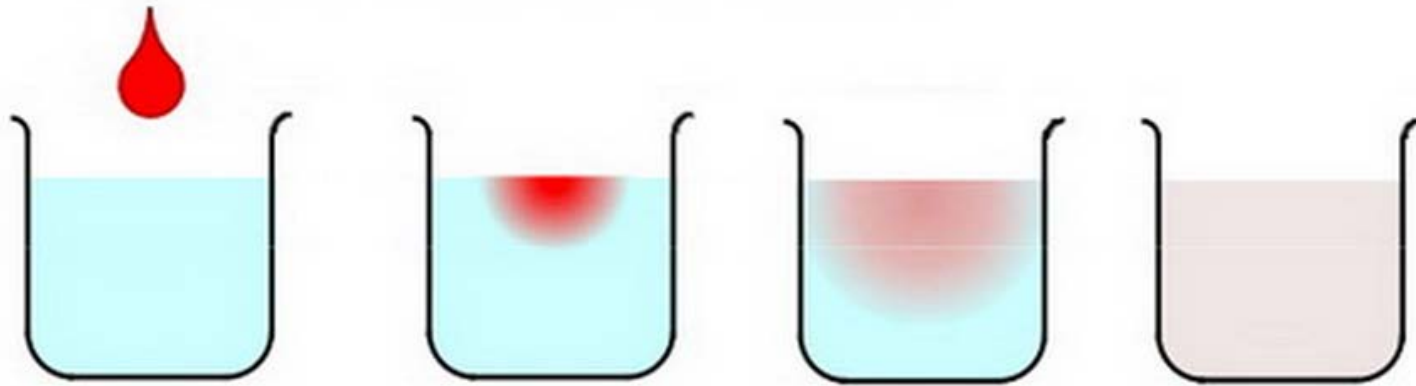
It is a good idea for receiver to prepare enough memory

“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

“diffusion” Sample Program (2) (Revisited)



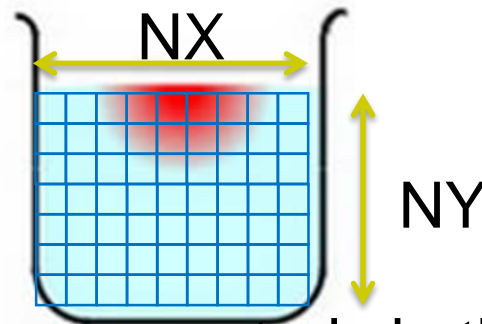
Available at [~endo-t-ac/ppcomp/18/diffusion/](https://endo-t-ac/ppcomp/18/diffusion/)

- 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: $O(nx \times ny \times nt)$

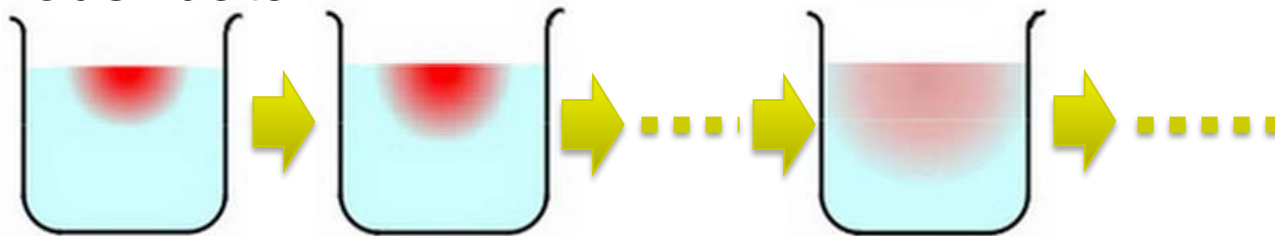
Data Structures in diffusion (Revisited)



- 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 $t=0$

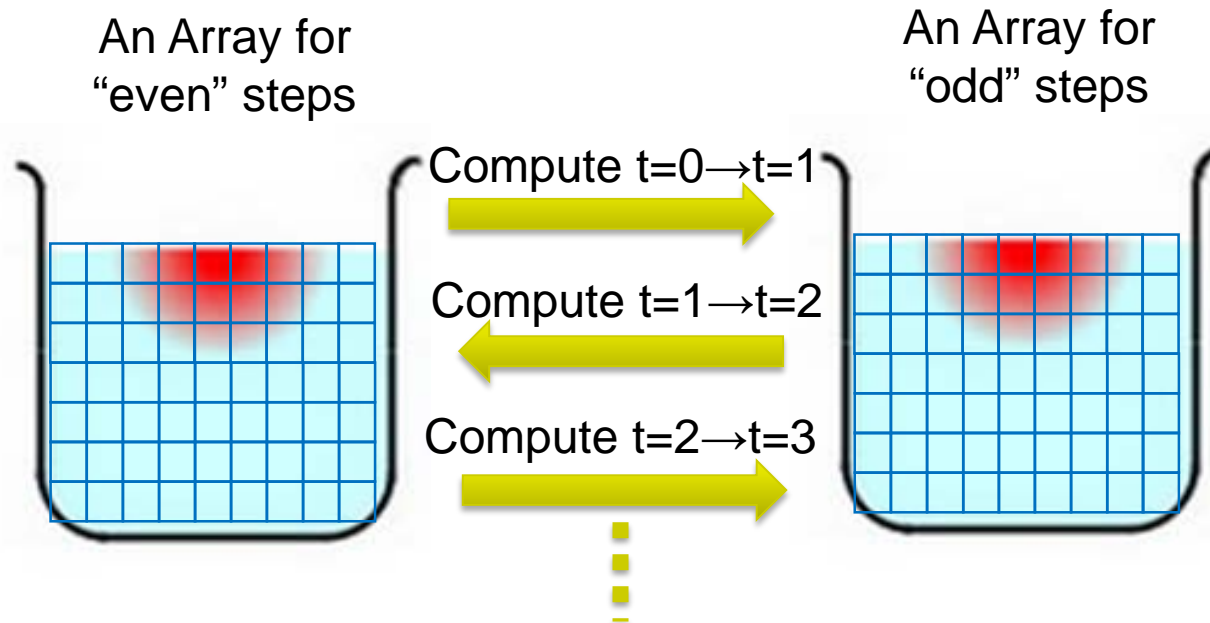
$t=1$

$t=20$

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

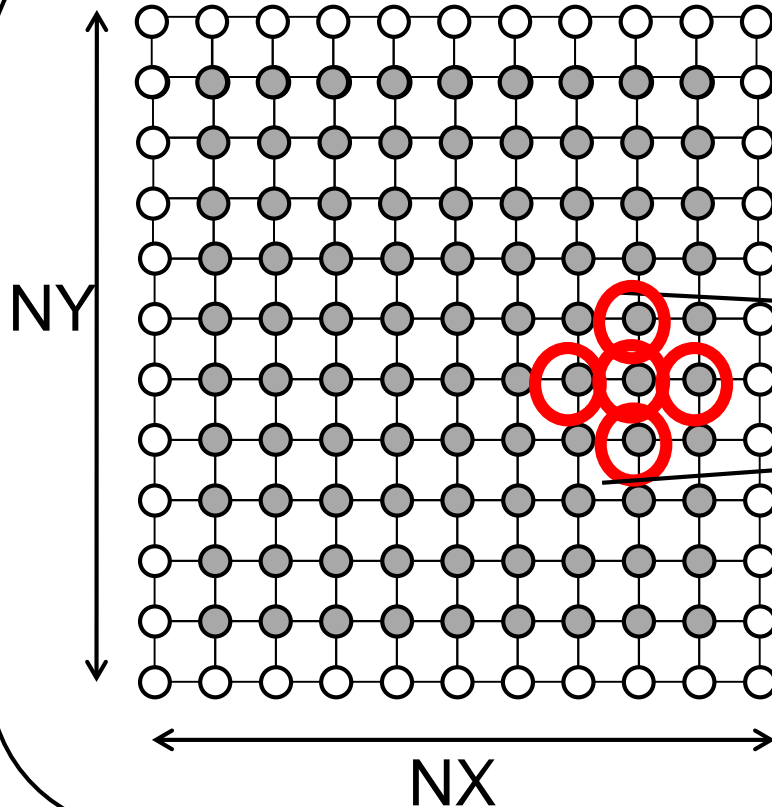


※ Sample program uses a global variables
`float data[2][NY][NX];`

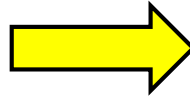
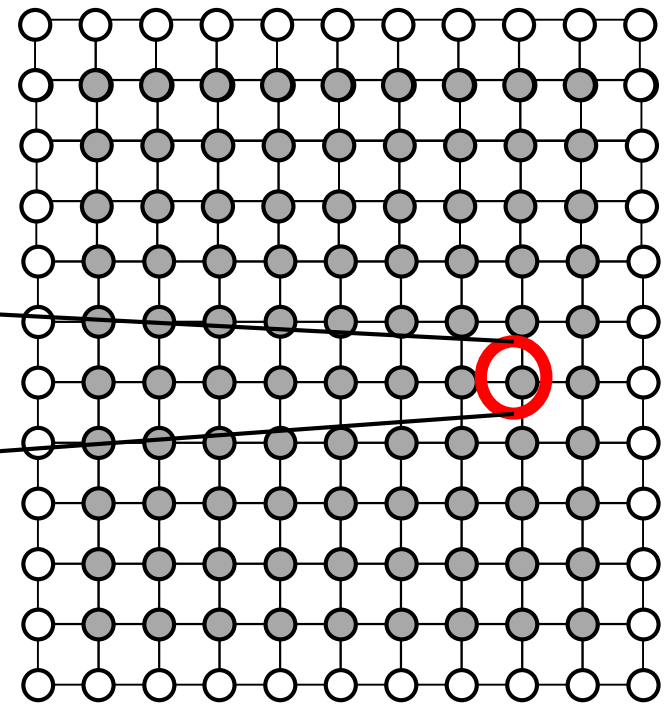
Data Structure in Original “diffusion”



An Array for “even” steps



An Array for “odd” steps



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) 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

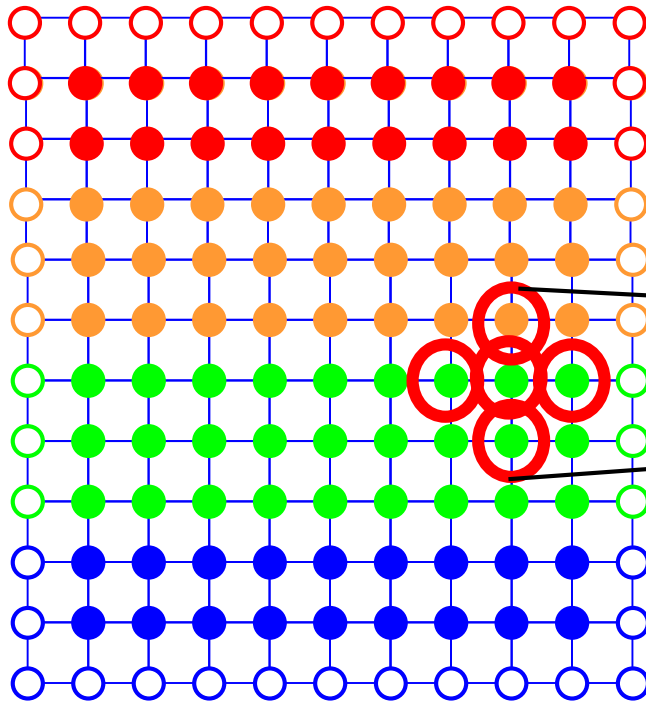
[Data] Arrays are divided among processes

- Each process has its own part of arrays

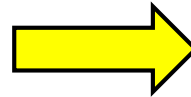
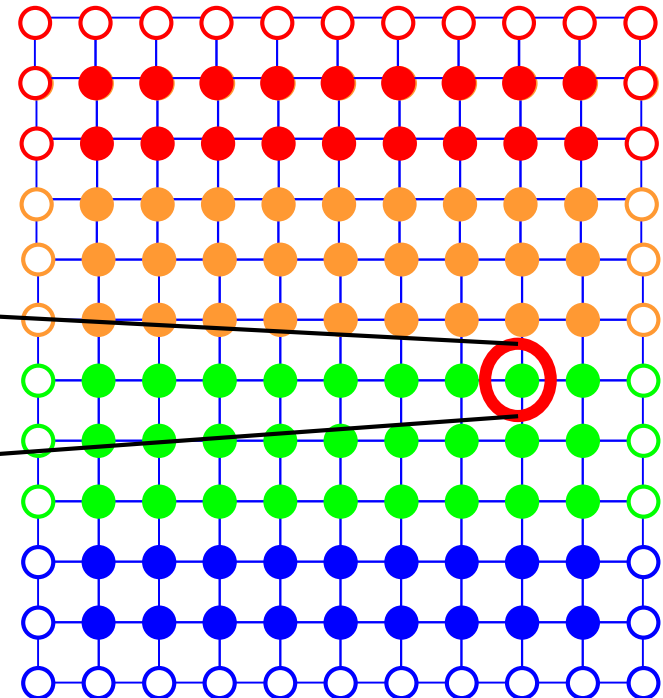
Considering Data Distribution (1)



An Array for “even” steps



An Array for “odd” steps

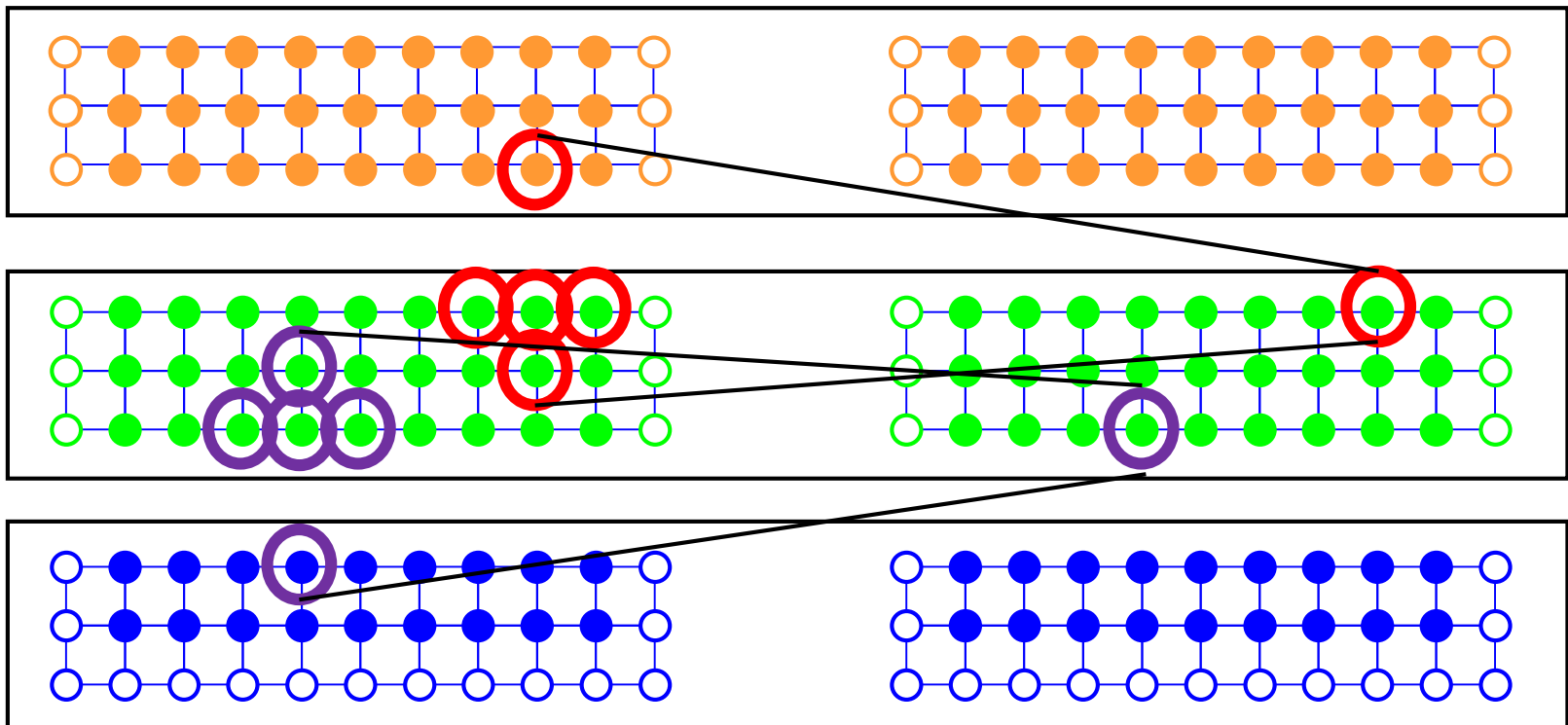


- A color = a process



Considering Data Distribution (2)

- A simple distribution is like:



Computation requires data in other processes

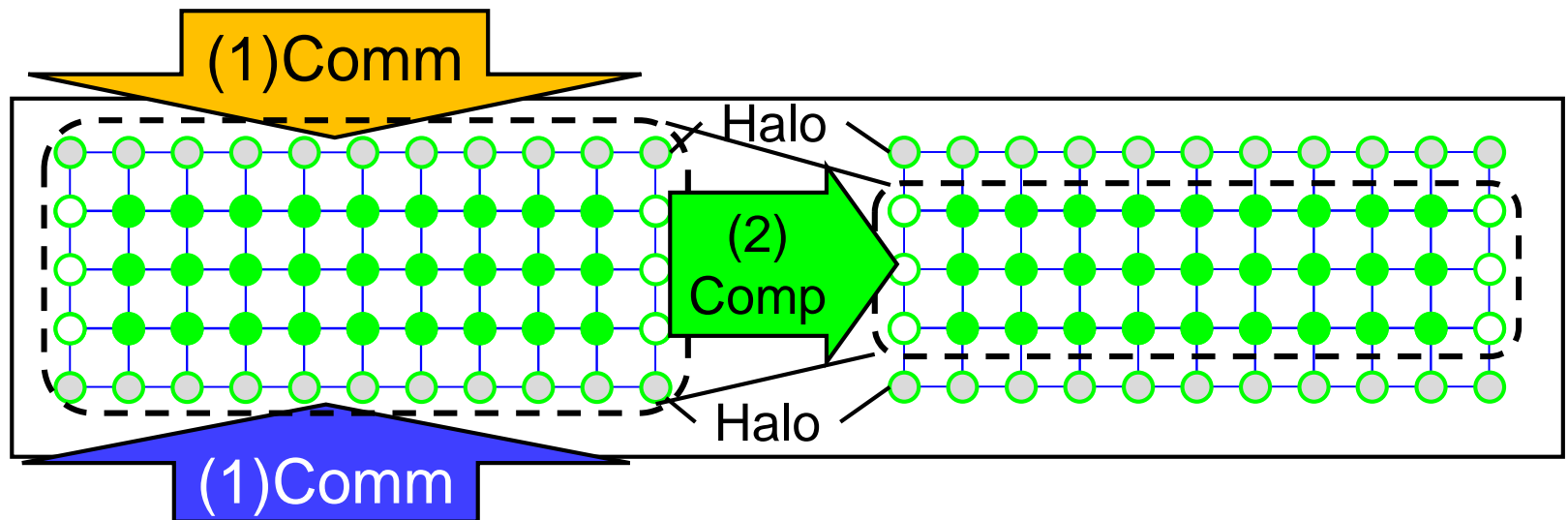
→ **Communication is required**

So, where should received data be put?



Introducing “Halo” Region

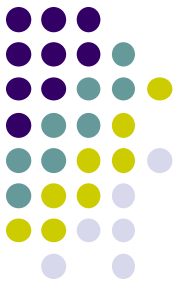
- It is a good idea to make additional rows to arrays
→ called “Halo” region or “sleeve” region



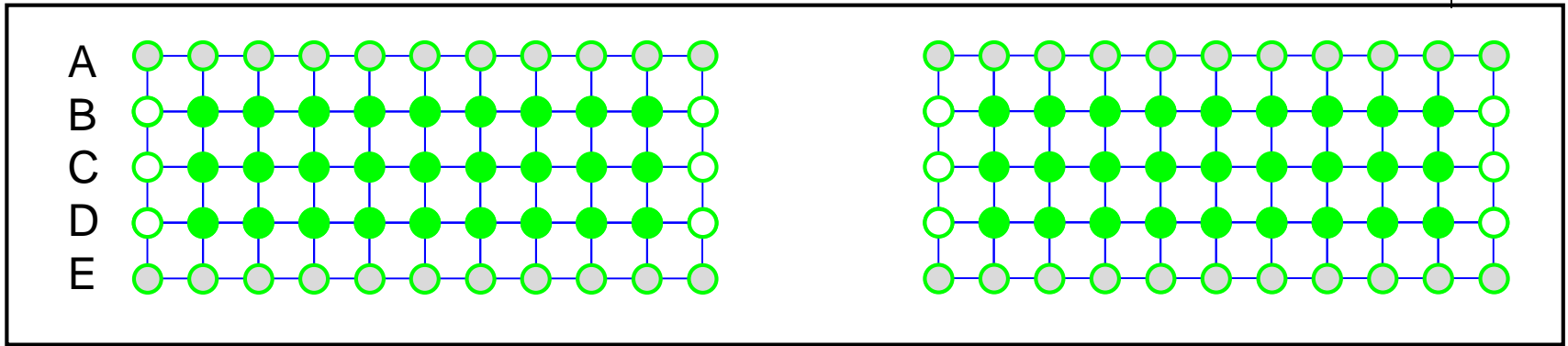
Each time step consists of:

- Communication:** Recv data and store into “halo” region
 - Also neighbor processes need “my” data
- Computation:** Old data at time t (including “halo”)
→ New data at time $t+1$

The name of “Halo” Region



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
```

} (1) Communication

```
    Computes points in rows B-D
```

```
    Switch old and new arrays
```

} (2) Computation

```
}
```

This version is still unsafe, because
this may cause **deadlock**

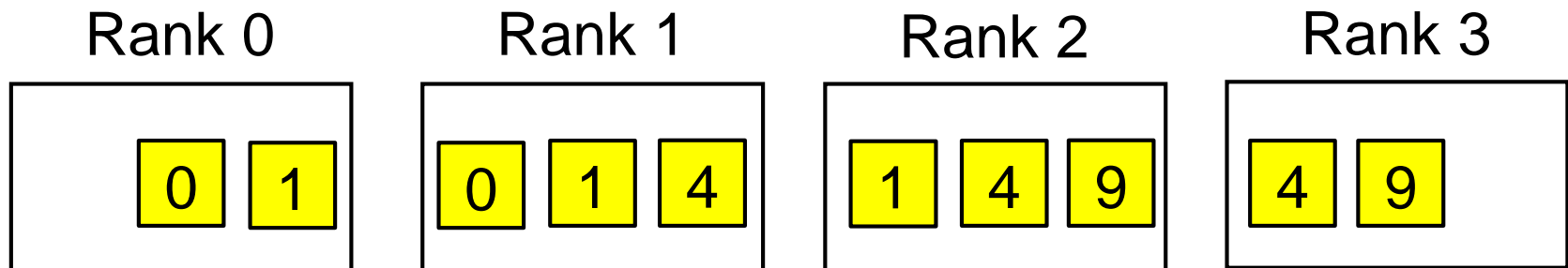
A Sample for Safe Neighbor Communication



Available at [~endo-t-ac/ppcomp/18/neicomm/](https://github.com/endo-t-ac/ppcomp/18/neicomm/)

Execution: `mpirun -np [np] ./neicomm`

- (1) Each process produces a single value (rank² here)
- (2) Each process receives values from its neighbors (rank-1 and rank+1)



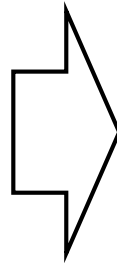


Neighbor Communication

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
 - 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 something)  
MPI_Wait(&req, &stat); ←wait for completion
```

MPI_Irecv: starts receiving, but it returns **I**mmEDIATELY

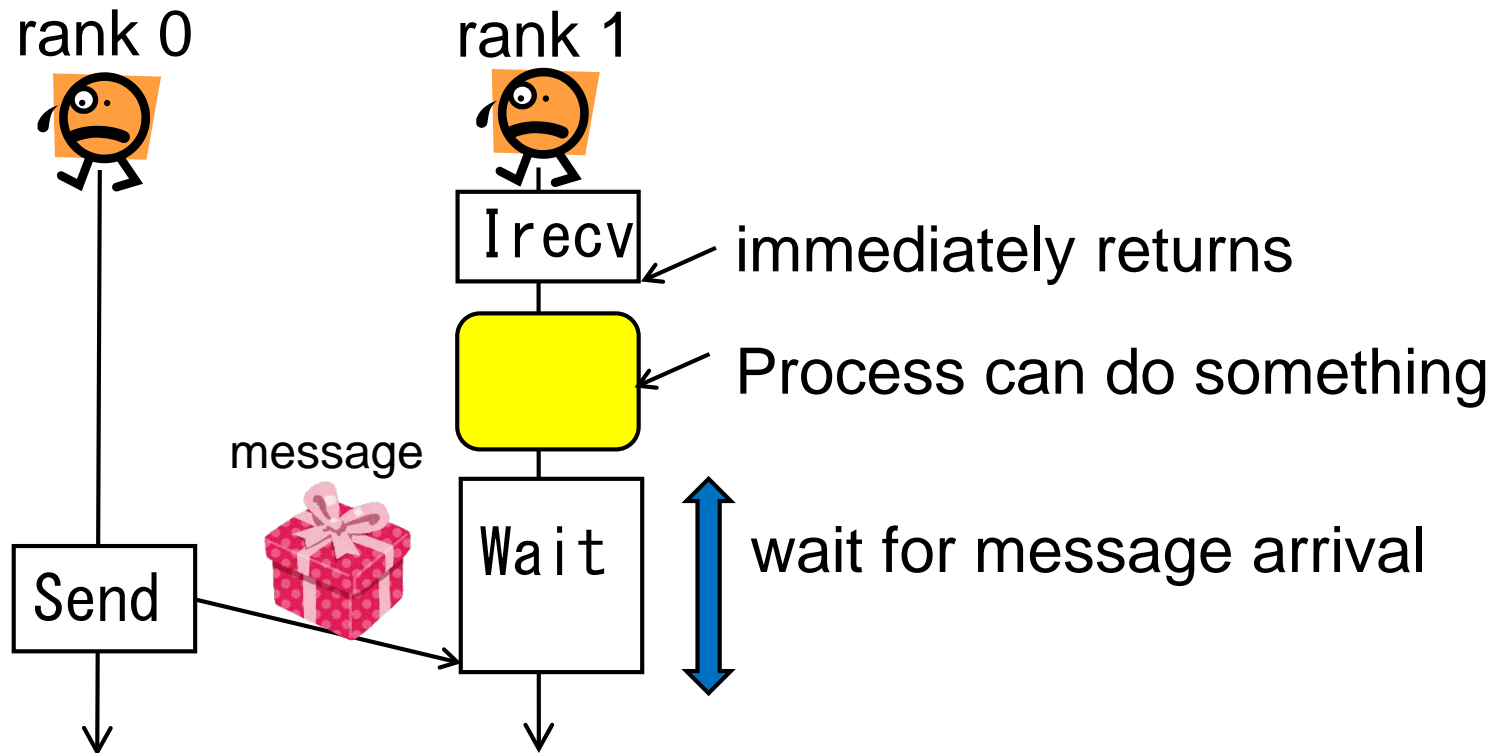
MPI_Wait: wait for message arrival

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
- ✧ $\text{MPI_Recv} = \text{MPI_Irecv} + \text{MPI_Wait}$





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  
: (Do something)
```

```
MPI_Wait(&req, &stat); ←wait for completion
```

MPI_Isend: starts sending, but it returns **I**mmediately

MPI_Wait must be used later

⌘ MPI_Send = MPI_Isend + MPI_Wait



MPI_Wait Family

- `MPI_Wait(&req, &stat);` ←wait for completion of one communication
- `MPI_Waitall(n, reqs, stats);` ←wait for completion of all n communications
- `MPI_Waitany(n, reqs, &idx, &stat);` ←wait for completion of one of n communications
- `MPI_Test(&req, &flag, &stat);` ←check completion of one communication
- `MPI_Testall, MPI_Testany...`

Assignments in MPI Part (Abstract)



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

Due date: May 28 (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 No. 7 slides or OCW-i.



Next Class

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