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

2025 Class No.4
[OpenMP Part] (2)
Sample Programs with Data Parallelism

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

# Overview of This Course

- Introduction Part
  - 2 classes
- OpenMP (OMP) Part
  - 4 classes ← We are here (2/4)
  - Report (required)
- OpenACC (ACC) Part
  - 2 classes
  - Report (required)
- CUDA Part
  - 3 classes
  - Report (elective)
- MPI Part
  - 3 classes
  - Report (elective)

Slack channel
in Science Tokyo Workspace
#dp-ppcomp-mcs-t418-2025

# Summary of Previous Class

OpenMP is for shared-memory parallel programming

- #pragma omp parallel defines a parallel region, where multiple threads work simultaneously
- With #pragma omp for, loop-based programs can be parallelized easily
- Shared variables and private variables
- We have reviewed OpenMP version of mm sample

# Today's Contents

Choose one of [O1]—[O4], and submit a report

Due date: May 1 (Thu)

[O1] Parallelize "diffusion" sample program by OpenMP.
[O2] Parallelize "bsort" sample program by OpenMP.
[O3] Parallelize "qsort" sample program by OpenMP.
[O4] (Freestyle) Parallelize *any* program by OpenMP.

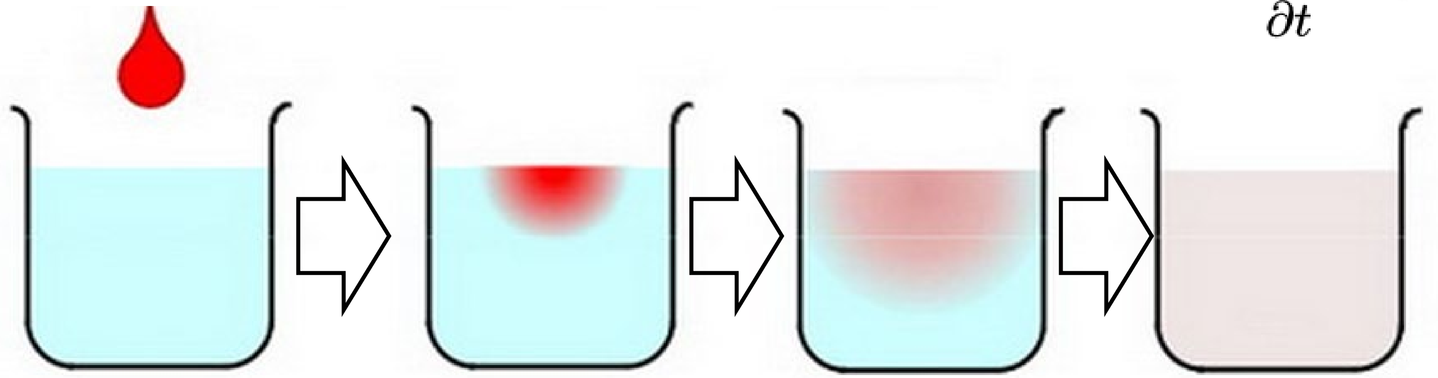
- Sample programs written with "for" loops are explained:
  - diffusion and bsort
- Sequential versions are at base/diffusion, base/bsort
  - There are already omp/diffusion, omp/bsort directories
  - They are only template and <u>NOT</u> parallelized
  - You can edit .c file there to solve [O1] or [O2]

# “diffusion” Sample Program (Target of [O1])

An example of diffusion phenomena:

- Pour a drop of ink into a water glass

$$\frac{\partial \phi}{\partial t} = D\nabla^2 \phi(\vec{r}, t)$$



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

- Density of ink in each point vary according to time → Simulated by computers
  - cf) Weather forecast compute wind speed, temperature, air pressure…

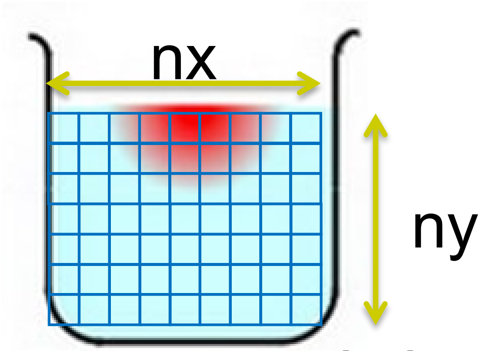
# “diffusion” Sample on TSUBAME

Available at ppcomp-ex/base/diffusion/
Go to the directory and do “make” command

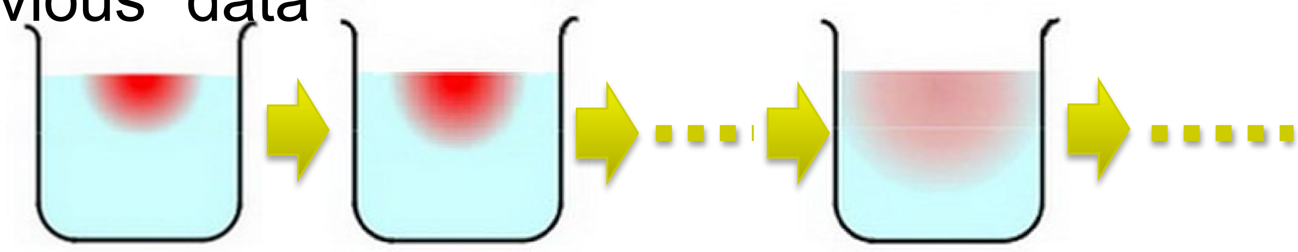
- Execution：./diffusion [nt]
- nt: Number of time steps
- nx, ny: Space grid size
  - nx=20000, ny=20000 (Fixed. See the code)
  - How can we make them variables? (mm sample will be useful as a reference)
- Compute Complexity：$O(nx \times ny \times nt)$

# Expression of Space to be Simulated

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



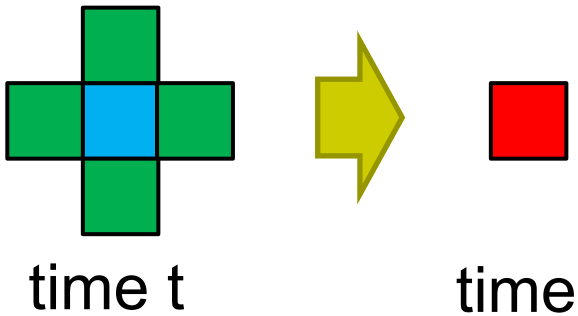

- Array elements are computed via timestep, by using "previous" data



Time step t=0 t=1 t=20

# Stencil Computations

- A data point *(x,y)* at time *t+1* is computed using following data
  - point *(x,y)* at time t
  - "Neighbor" points of *(x,y)* at time t




  - In diffusion sample, the computation is simply "average of 5 points"
- Computations of similar type are called "stencil computations"
  - Frequently used in fluid simulations

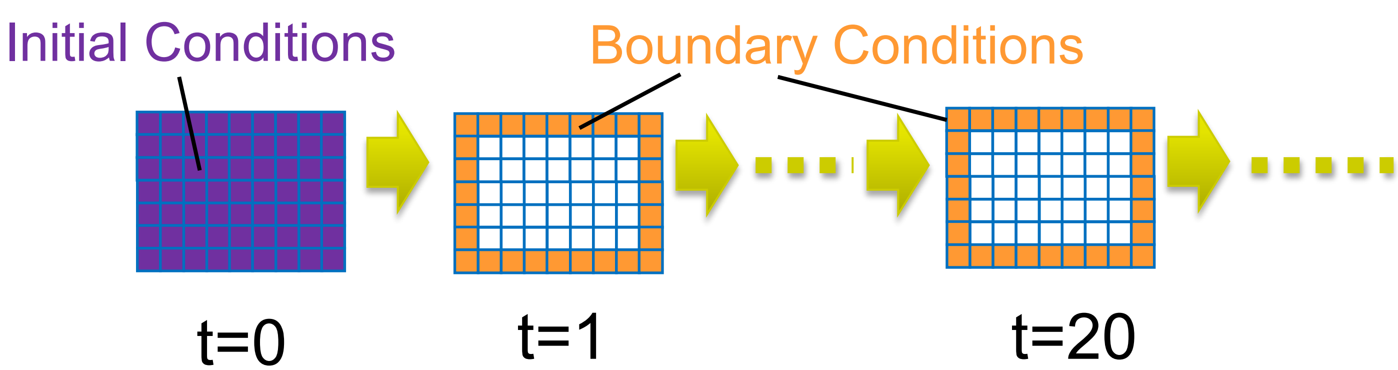

Original meanings of "stencil"

# Initial Conditions & Boundary Conditions

In stencil computations, following data points cannot be computed

Instead, we have to give them (for example, as input data)

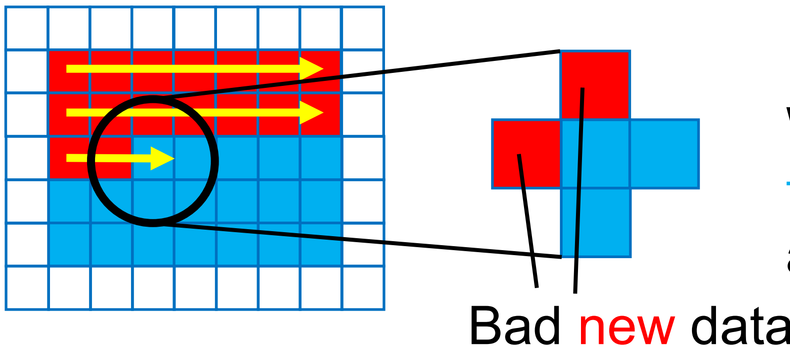
- All points at t=0 (Initial conditions)
  - In diffusion sample, given in init()
- "Boundary" points for all t (Boundary conditions)
  - In diffusion sample, they are constant during simulation
  - → See ranges of for-loops in calc(); boundaries are skipped
  - This is not good for simulation of a water glass ☹, but it's simple…



# A Single Array Does not Work

Let us compute t ➔ t+1

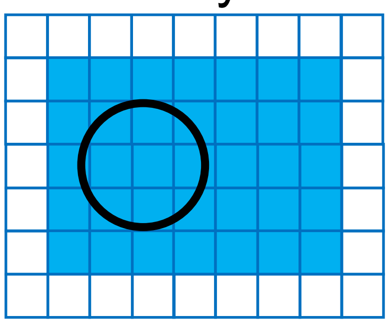
- With a single 2D array (Bug! ☹)



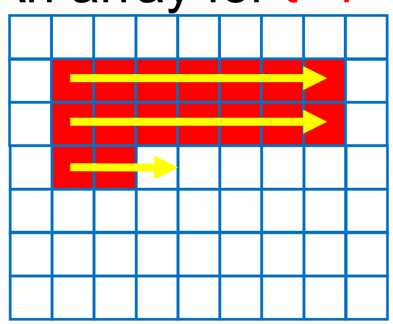

We need neighbor points at time t, but some have been already updated to t+1 ☹

- With separate 2D arrays (Good ☺)

An array for t



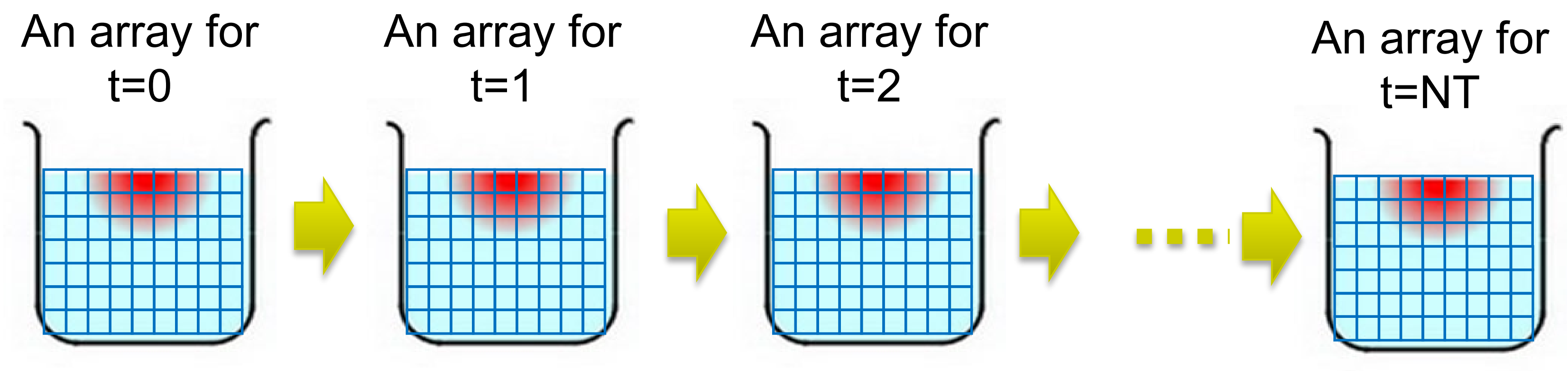
An array for t+1



We can access "old" neighbor points correctly ☺

# Multiple Arrays?

- We repeat update of the array for NT times



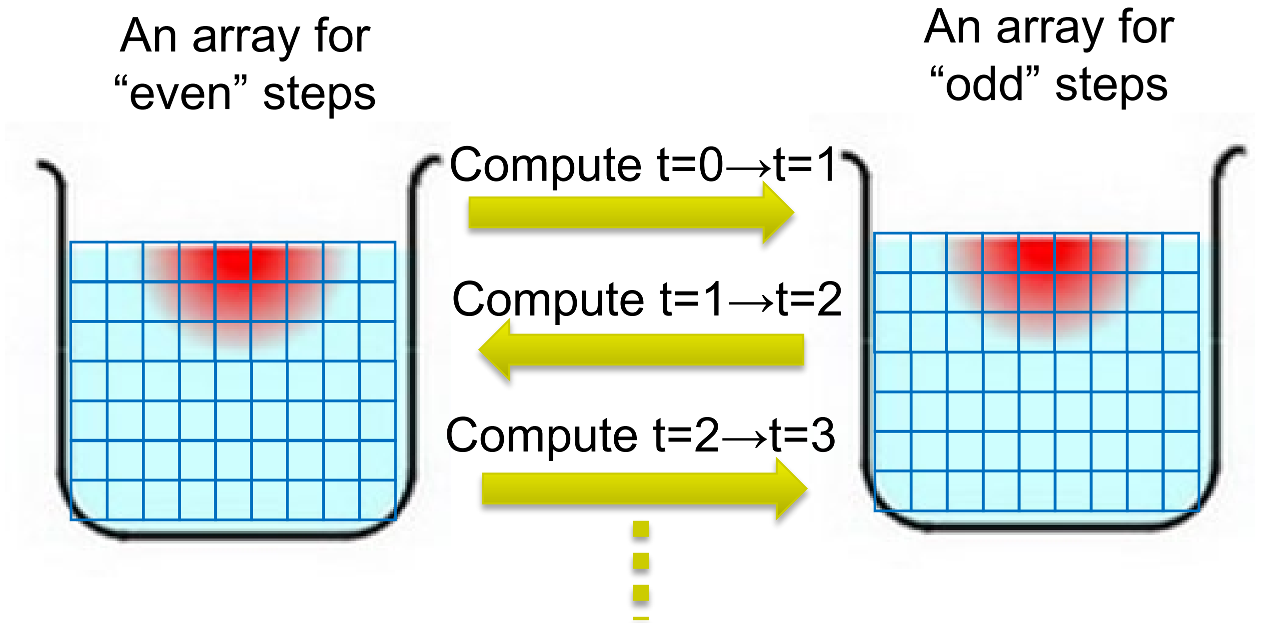

A simple way is to make arrays for all time steps

float data[NT+1][NY][NX]

- This uses too much memory
- Do we need all of (NT+1) arrays?

# Double Buffering Technique

- It is sufficient to have "current" array and "next" array.
- It is better to use only "Double buffers"




The diffusion sample program uses

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

# How Do We Parallelize "diffusion"?

calc() takes long time, complexity is O(nx ny nt)

It mainly uses "for" loops

➔ #pragma omp parallel for is useful! But…

There are 3 (t, x, y) loops. Which should be parallelized?

[Hint1] Parallelizing either of spatial loop (x, y) would be good. Then spaces are divided into multiple threads

→ [Q] Parallelizing t loop is a not good idea. Why?

[Hint2] Take care of "pitfall in nested loops" (see slides in previous class)

# Towards “Correct” Parallel Programming

There are several types of bugs in parallel programming

- Bugs in compile time

- Bugs in run time
  - Bugs that abort execution (cf. segmentation fault)
  - Silent bugs → Hardest to find!

All bugs should be avoided!

# When Can We Use “omp for”?

- Loops with some (complex) forms cannot be supported, unfortunately ☹
- The target loop must be in the following form

```
#pragma omp for
  for (i = value; i op value; incr-part)
    body
```

```
 "op" : <, >, <=, >=, etc.
 "incr-part" : i++, i--, i+=c, i-=c, etc.
```

OK ☺: `for (x = n; x >= 0; x-=4) …`

ERROR ☹: `for (i = 0; test(i); i++) …`

ERROR ☹: `for (p = head; p != NULL; p = p->next)`

Errors in compile time

# What are Differences between These Codes?

```
double D[100];
      :
```

Code A

```
#pragma omp parallel for
  for (i = 0; i < 100; i++) {
     D[i] = D[i]+1.0;
  }
```

Code B

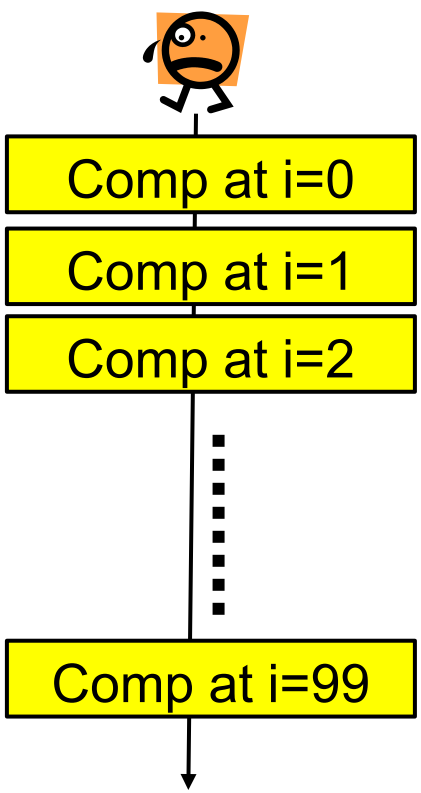
```
#pragma omp parallel for
  for (i = 0; i < 99; i++) {
     D[i+1] = D[i]+1.0;
  }
```

- Both codes can be compiled and executed…
- But only code A is correct ☺ , code B has a bug ☹
  - Code B's results may be wrong

# Sequential Execution and Parallel Execution of Loop

[Sequential]

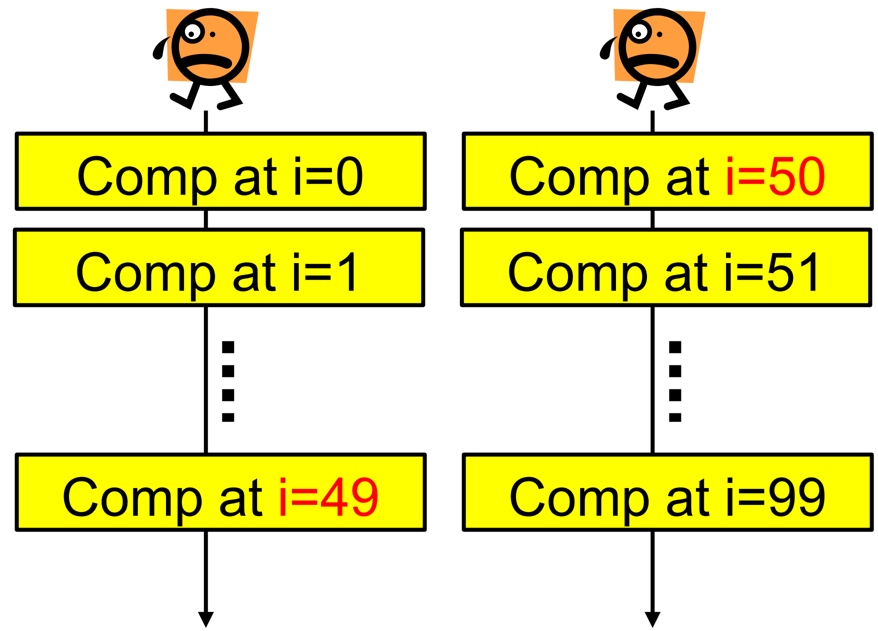
for (i = 0; i < 100; i++) …




[Parallel]

#pragma omp parallel for

for (i = 0; i < 100; i++) …




in case of 2 threads,
i=50 is computed before i=49

# Difference between Two Codes

Code A

```
#pragma omp parallel for
  for (i = 0; i < 100; i++) {
     D[i] = D[i]+1.0;
  }
```

OK

It is ok to reorder 100 computations

Code B

```
#pragma omp parallel for
  for (i = 0; i < 99; i++) {
     D[i+1] = D[i]+1.0;
  }
```

NG

Computations must be done in an order (i=0,1,2…)
➔ Parallelization breaks the order

# Dependency between Computations

We define following sets for computation C

- Read set R(C): the set of variables read by C
- Write set W(C): the set of variables written by C
  - Ex) C: x = y+z ➔ R(C) = {y, z}, W(C) = {x}

We define dependency between C1 and C2

- If ($W(C1) \cap R(C2) \neq \emptyset$), C1 and C2 are dependent (write vs read)
- If ($R(C1) \cap W(C2) \neq \emptyset$), C1 and C2 are dependent (read vs write)
- If ($W(C1) \cap W(C2) \neq \emptyset$), C1 and C2 are dependent (write vs write)
- Otherwise, C1 and C2 are independent
  - ※ read vs read cases are independent

<u>If C1 and C2 are independent, parallelization of C1 and C2 is safe ☺</u>

# Example of Dependency

Code A

```
#pragma omp parallel for
  for (i = 0; i < 100; i++) {
     D[i] = D[i]+1.0;    ← Ai
  }
```

$R(A_i) = \{D[i]\}$, $W(A_i) = \{D[i]\}$

All 100 computations are independent

Code B

```
#pragma omp parallel for
  for (i = 0; i < 99; i++) {
     D[i+1] = D[i]+1.0;  ← Bi
  }
```

$R(B_i) = \{D[i]\}$, $W(B_i) = \{D[i\underline{+1}]\}$

$R(B_{i+1}) \cap W(B_i) = \{D[i+1]\} \neq \emptyset$ → Dependent!

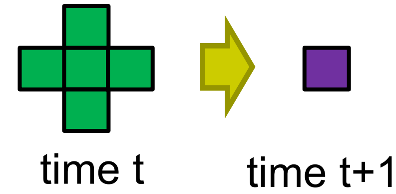
# Dependency and Parallelism in Stencil Computations

Consider 1D stencil computation:

```
for (t = 0; t < NT; t++)
    for (x = 1; x < NX-1; x++)
        f[t+1,x] = (f[t,x-1] + f[t,x] + f[t,x+1]) / 3.0   /*  c[t,x] */
```

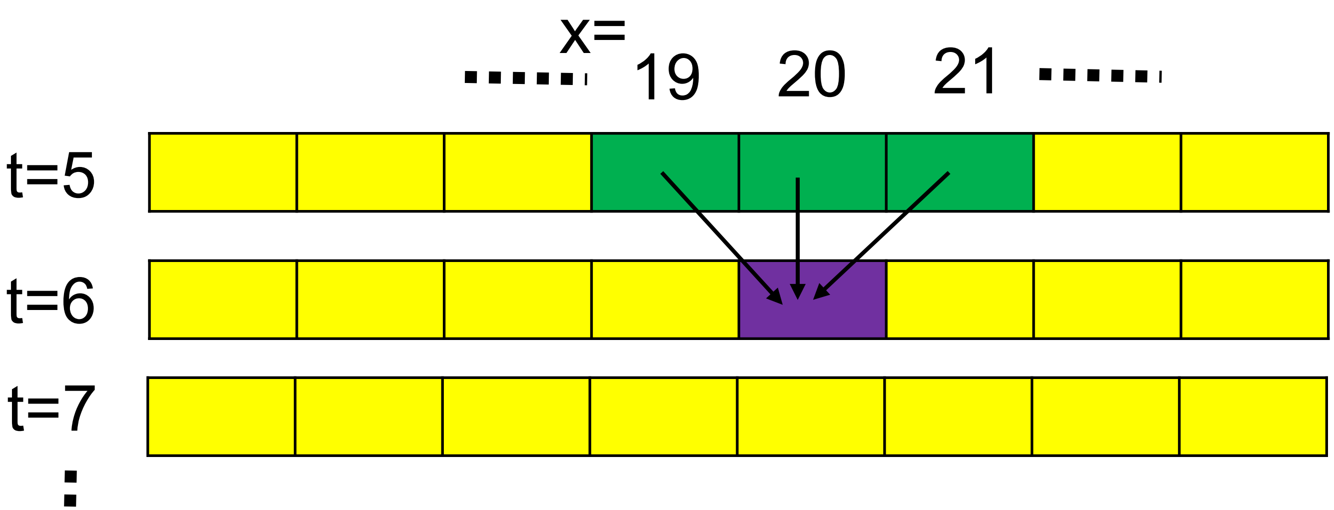
$$f_{t+1,x} = (f_{t,x-1} + f_{t,x} + f_{t,x+1}) / 3.0 \quad /^{*}\ c_{t,x}\ ^{*}/$$

※ This is simpler than "diffusion" (2D) sample




We let $c_{t,x}$ be computation of a single point $f_{t+1,x}$

$R(c_{t,x}) = \{f_{t,x-1}, f_{t,x}, f_{t,x+1}\}$, $W(c_{t,x}) = \{f_{t+1,x}\}$



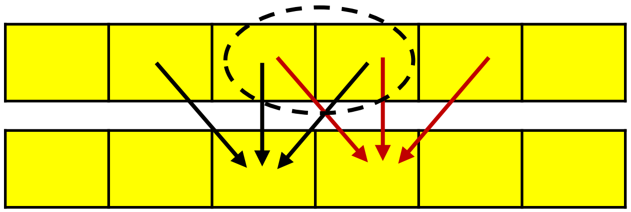

※ This figure omits double buffering technique

# Discussion on Stencil: Case of Spatial Loop

for (t = 0; t < NT; t++)
for (x = 1; x < NX-1; x++) ← Is this loop parallelizable?
$f_{t+1,x} = (f_{t,x-1} + f_{t,x} + f_{t,x+1}) / 3.0$ /* $c_{t,x}$ */

- Can we compute $c_{5,20}$ and $c_{5,21}$ in parallel? *(t is same, x is different)*
  - $R(C_{5,20})=\{f_{5,19}, f_{5,20}, f_{5,21}\}$, $W(C_{5,20})=\{f_{6,20}\}$
  - $R(C_{5,21})=\{f_{5,20}, f_{5,21}, f_{5,22}\}$, $W(C_{5,21})=\{f_{6,21}\}$
  - → They are independent ☺ ,for all pairs of x
  - → x loop can be parallelized

Read vs. Read is Ok



# Discussion on Stencil: Case of Temporal Loop

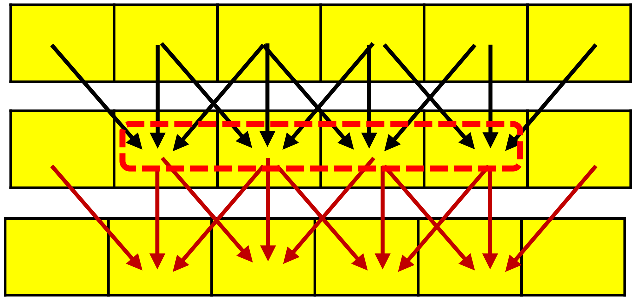
for (t = 0; t < NT; t++)   ← Is this loop parallelizable?

$$\left.\begin{array}{l}\text{for (x = 1; x < NX-1; x++)}\\ \quad f_{t+1,x} = (f_{t,x-1} + f_{t,x} + f_{t,x+1}) / 3.0\end{array}\right\} C_t$$

- Can we compute $C_5$ and $C_6$ in parallel? *(t is different)*
  - $R(C_5)=\{f_{5,0}, \ldots, f_{5,NX-1}\}$, $W(C_5)=\{f_{6,1}, \ldots, f_{6,NX-2}\}$
  - $R(C_6)=\{f_{6,0}, \ldots, f_{6,NX-1}\}$, $W(C_6)=\{f_{7,1}, \ldots, f_{7,NX-2}\}$

  → $R(C_6) \cap W(C_5) = \{f_{6,1}, \ldots, f_{6,NX-2}\} \neq \emptyset$

  → They are dependent ☹



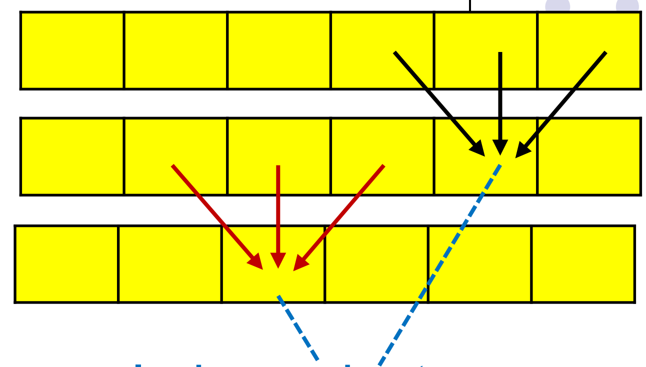
dependent!!

In Assignment [O1]
- it is OK to parallelize x-loop or y-loop
- it is NG to parallelize t-loop

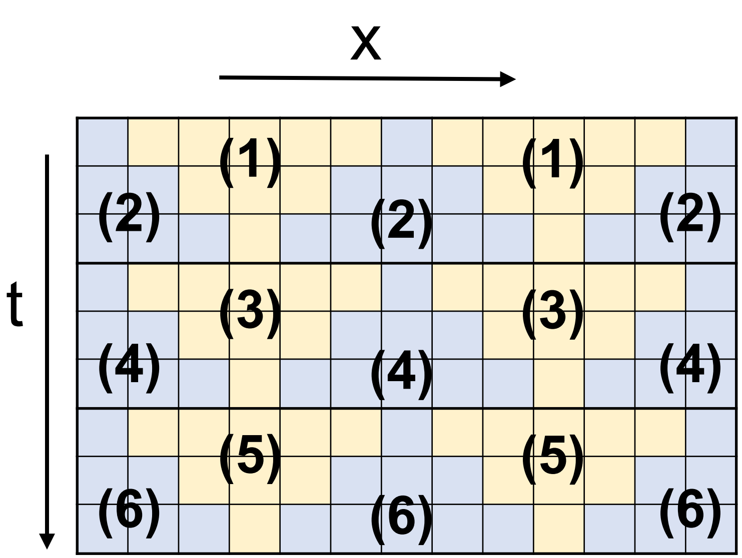
# Advanced Topic: More Speed in Stencil

We see dependency more in detail:

- $c_{6,20}$ depends on $c_{5,19}$ , $c_{5,20}$, $c_{5,21}$
  - The same point or its direct neighbor
- But not on $c_{5,22}$




Independent
➔ Can be parallelized
➔ Can be reordered




Temporal blocking technique:
After computations in (1) finish, we can start (2)
"Trapezoids" in the same stage can be parallelized
➔ Speed is improved for better access locality

# "bsort" Sample Program (Target of [O2])

- Sort an array with bitonic sort algorithm

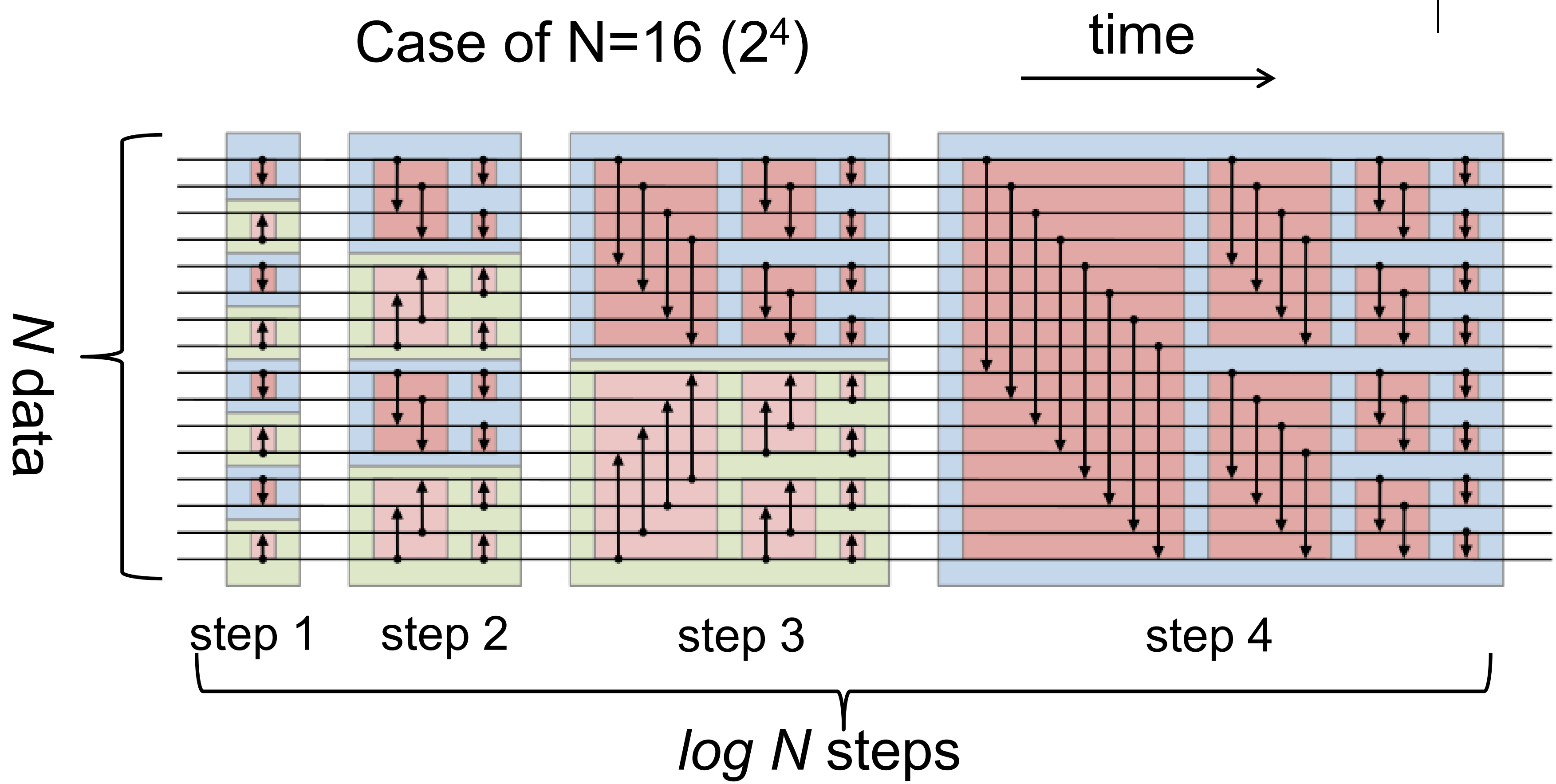
N

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| input | 0.84 | 0.39 | 0.78 | 0.80 | 0.91 | 0.20 | 0.34 | 0.77 |
| output | 0.20 | 0.34 | 0.39 | 0.77 | 0.78 | 0.80 | 0.84 | 0.91 |

In output, for all i, data[i] ≦ data[i+1]

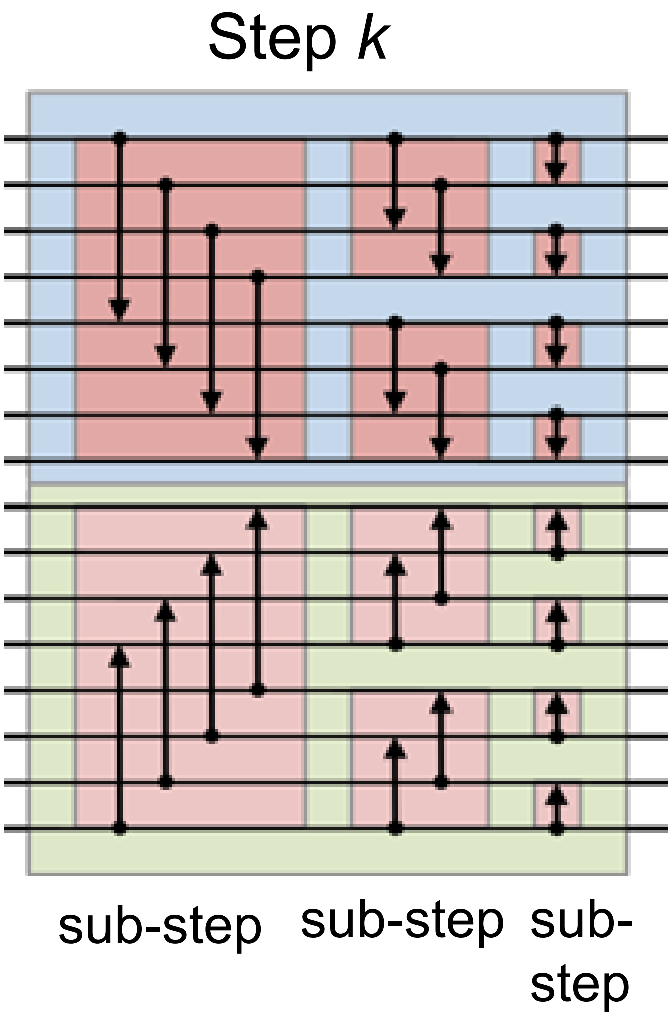
- There are many famous sorting algorithm with different compute complexity:
  - bubble sort and insertion sort, $O(N^2)$
  - bitonic sort [O2], $O(N \log^2 N) = O(N (\log N)^2)$
  - quick sort [O3], $O(N \log N)$
  - merge sort, $O(N \log N)$
  - radix sort

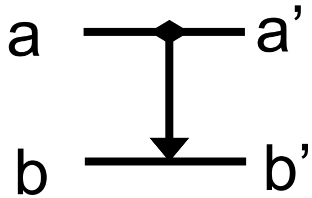
# Overview of bitonic sort



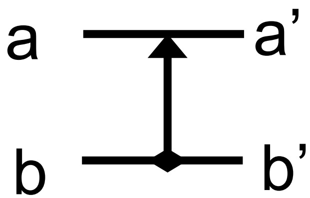

the figure is from Wikipedia

# Basic Operations in bsort




if a ≦ b, a'=a, b'=b
if a>b, a'=b, b'=a




if a ≦ b, a'=b, b'=a
if a>b, a'=a, b'=b

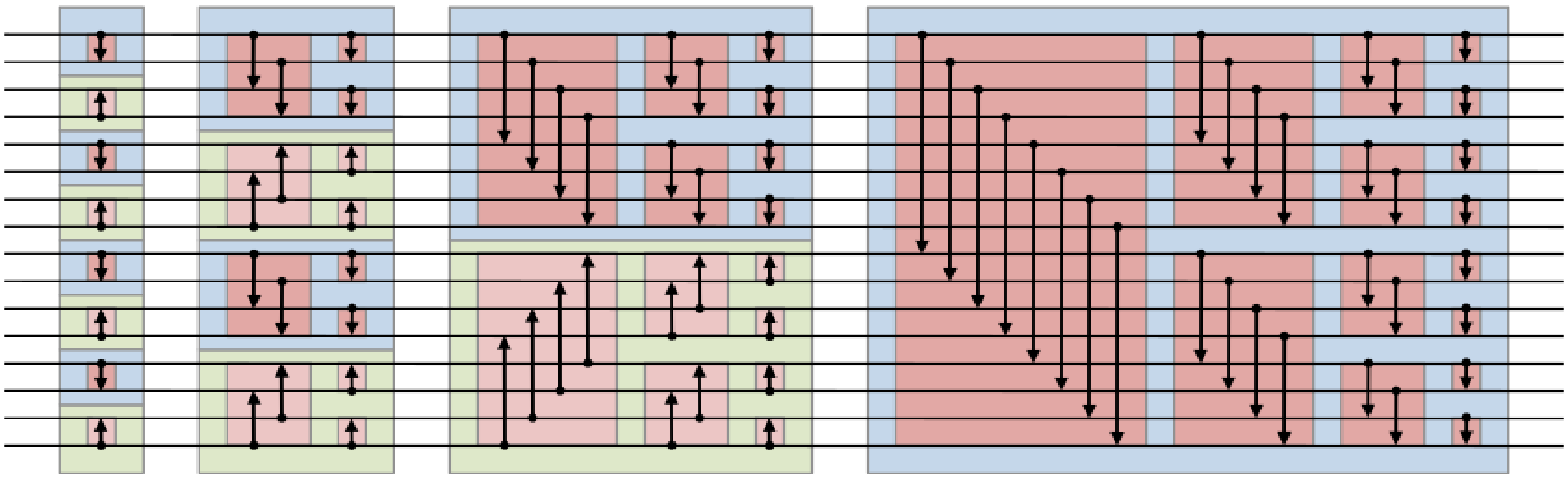
After blue box, data are sorted in ascending order

After green box, data are sorted in descending order

*if interested in its proof, please try Google/ChatGPT*

# Computation Complexity

Sub-step: O(N) ➔ Step k: O(kN)



step 1 step 2 step 3 step 4 = S = log N

Total step number S = log N

Total complexity: O(N + 2N + 3N + ... + S N)

= $O(S^2 N)$ = $O(N \log^2 N)$

# Loops in bsort

- Bitonic sort assumes number of data is power of 2
- We compute N2 ≧ N, N2 is power of 2, and copy data

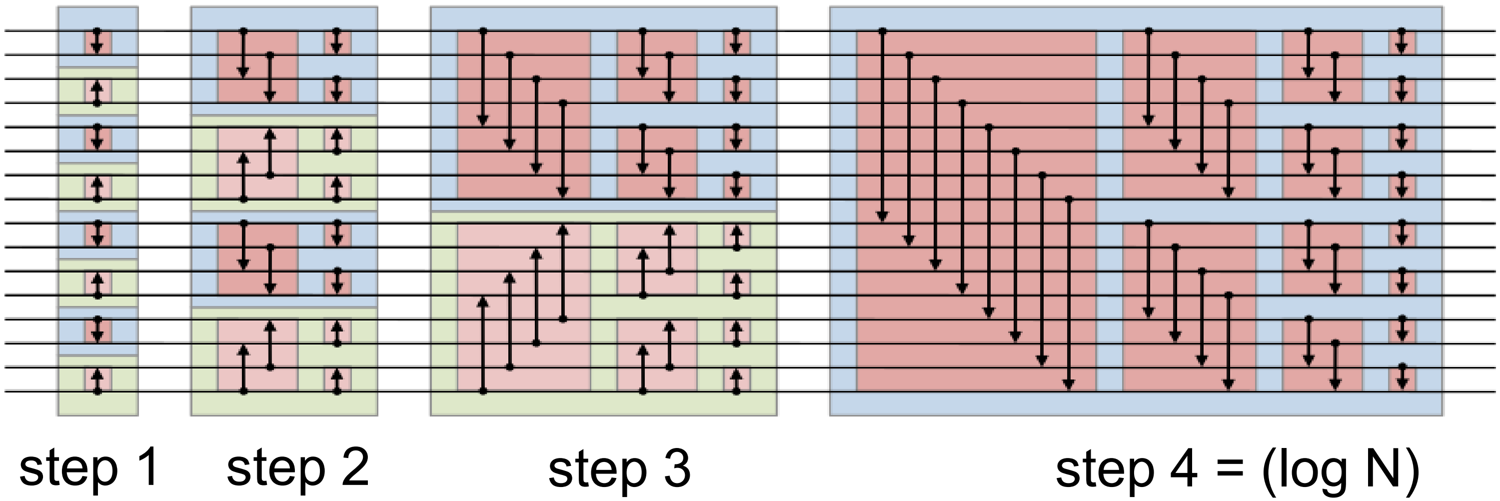
loop structure of sort()

1<<i = $2^i$. (<< is bit-shift)

```
for (i = 1; (1<<i) <= N2; i++) {  // step loop
  for (j = i-1; j >= 0; j--) {    // sub-step loop
      :
      for (k = 0; k < N2; k++) {
          // compare 2 data and swap if appropriate
      }
  }
}
```

Program is a bit complicated for bit operations,
but you can use as it is ☺

# How Do We Parallelize “bsort”?




- Since steps/sub-steps cannot be reordered, we should NOT parallelize *i, j* loops
- Please confirm that parallelizing *k* loops is safe

# Why Do We Use both bsort and qsort?

- Quick sort with O(N log N) complexity is usually faster than bitonic sort with $O(N \log^2 N)$
- Bitonic sort is easier for parallelization
  - In OpenMP, we can use #pragma omp for
    - OpenMP version of qsort will be explained in next class
  - Also parallelizing bsort with OpenACC/CUDA/MPI is easier
    - qsort is harder, and out of scope in this class ☹
    - cf) Cederman et al. “GPU-Quicksort: A practical Quicksort algorithm for graphics processors”, JEA vol14, 2010

# Assignments in OpenMP Part (Abstract)

Choose one of [O1]—[O4], and submit a report

Due date: May 1 (Thu)

[O1] Parallelize "diffusion" sample program by OpenMP.

[O2] Parallelize "bsort" sample program by OpenMP.

[O3] Parallelize "qsort" sample program by OpenMP.

[O4] (Freestyle) Parallelize *any* program by OpenMP.

For more detail, please see ppcomp25-3 slides

# Plan of OMP Part

- Class #3
  - Introduction to OpenMP
- Class #4 (Today)
  - Data parallelism with for loops
  - diffusion sample [O1], bsort sample [O2]
- Class #5
  - Task parallelism
  - qsort sample [O3]
- Class #6
  - What are bottlenecks, race conditions?