

2018

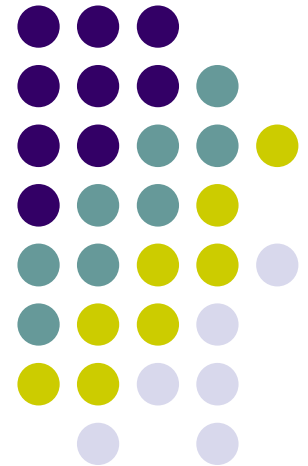
Practical Parallel Computing (実践的並列コンピューティング) No. 4

Shared Memory Parallel Programming with OpenMP (2)

Toshio Endo

School of Computing & GSIC

endo@is.titech.ac.jp



“mm” sample: Matrix Multiply



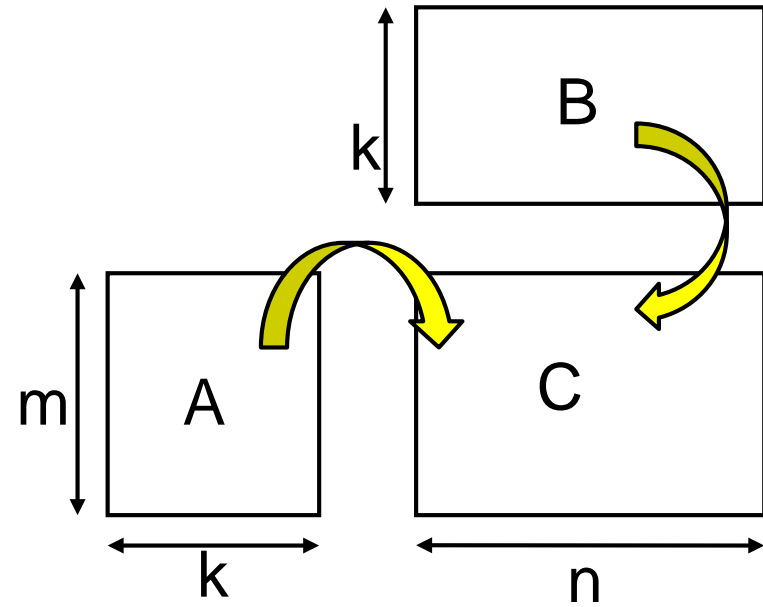
Available at [~endo-t-ac/ppcomp/18/mm/](https://endo-t-ac/ppcomp/18/mm/)

A: a $(m \times k)$ matrix, B: a $(k \times n)$ matrix

C: a $(m \times n)$ matrix

$$C \leftarrow A \times B$$

- Algorithm with a triple for loop
- Supports variable matrix size.
 - Each matrix is expressed as a 1D array by *column-major* format
- Execution: `./mm [m] [n] [k]`





Matrix Multiply Algorithm

```
for (i = 0; i < m; i++) {  
    for (j = 0; j < n; j++) {  
        for (l = 0; l < k; l++) {  
            Ci,j += Ai,l * Bl,j;  
        } } }  
←For each row in C  
←For each column in C  
←For dot product
```

- The innermost statement is executed for mnk times
- Compute Complexity: $O(mnk)$
 - Computation speed (Flops) is obtained as $\frac{2mnk}{t}$, where t is execution time

The innermost statement includes
2 (floating point) calculation

- [Q] What if loop order is changed?
 - IJL order in above. JLI order in mm sample
 - Number of operations does not change. But how is the speed?

Variable Length Arrays in (Classical) C Language



- `int a[n];` raises an error. How do we do?
- `void *malloc(size_t size);`
⇒ Allocates a memory region of *size* bytes from “heap region”, and returns its head pointer
- When it becomes unnecessary, it should be discarded with `free()` function

A fixed length array

```
int a[5];  
  
... a[i] can be used ...
```

A variable length array

```
int *a;  
a = (int *)malloc(sizeof(int)*n);  
  
... a[i] can be used ...  
  
free(a);
```

array length

⌘ Exceptionally, C99 specification includes variable length arrays

How We Do for Multiple Dimensional Arrays



`int a[m][n];` raises an error. How do we do?

Not in a straightforward way. Instead, we do either of:

(1) Use a pointer of pointers

- We *malloc* m 1D arrays for every row (each has n length)
- We *malloc* 1D array of m length to store the above pointers


(2) Use a 1D array with length of $m \times n$

(*mm sample uses this method*)

- To access an array element, we should use `a[i*n+j]` or `a[i+j*m]`, instead of `a[i][j]`

Express a 2D array using a 1D array




“I want
to use ...”

a 2D array $a[m][n]$

m

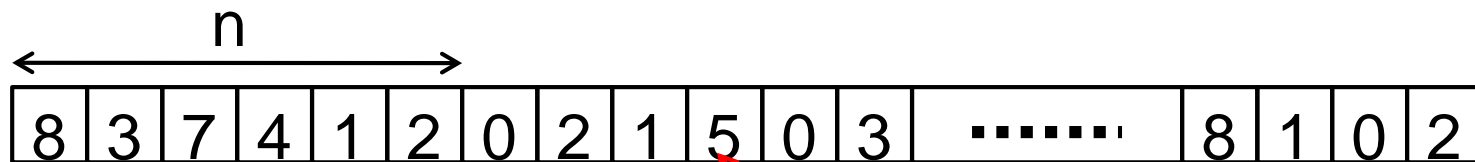
8	3	7	4	1	2
0	2	1	5	0	3
1	8	6	4	2	1
3	4	8	1	0	2

n

$a[1][3]$

Expressions in C language

```
int *a; a = malloc(sizeof(int)*m*n);
```



$a[1*n+3]$

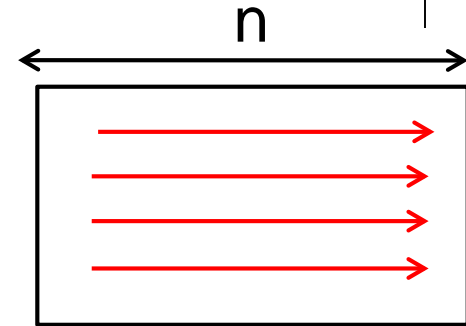
In this case, an element $a_{i,j}$ is $a[i*n+j]$

Two Data Formats

Row major format

- More natural for C programmers

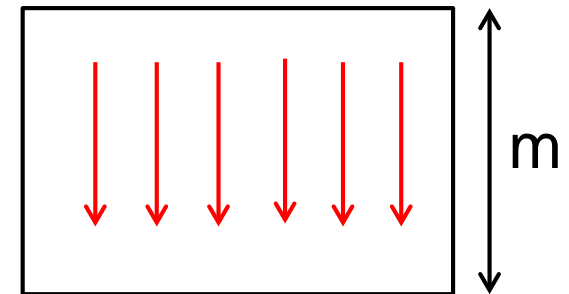
$$a_{i,j} \Rightarrow a[i*n+j]$$



Column major format

- BLAS library
- **mm sample**

$$a_{i,j} \Rightarrow a[i+j*m]$$



- We have more choices for 3D, 4D... arrays

[Q] Does the format affect the execution speed?

OpenMP Version of mm (mm-omp)



- One of loops is parallelized

```
#pragma omp parallel private(i,l)
```

```
#pragma omp for
```

```
for (j = 0; j < n; j++) {      ← j loop is parallelized
    for (l = 0; l < k; l++) {
        for (i = 0; i < m; i++) {
            C[i+j*ldc] += A[i+l*lda] * B[l+j*ldb];
        } } }
```

What is “private” option for?

Shared Variables & Private Variables (1)



While OpenMP uses “shared memory model”, **not all are shared**

In default, variables are classified as follows

- Variables declared **out of** parallel region \Rightarrow **Shared variables**
- Variables declared **inside** parallel region \Rightarrow **Private variables**

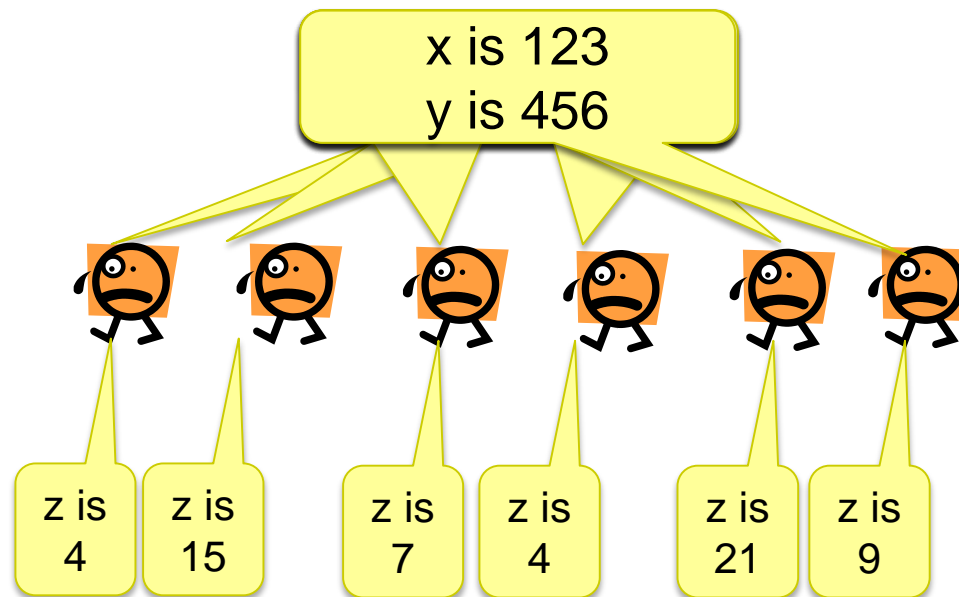
```
{  
    int s = 1000; shared  
    #pragma omp parallel  
    {  
        int i; private  
        i = func(s, omp_get_thread_num());  
        printf( "%d\n" , i);  
    }  
}
```

```
int func(int a, int b)  
{  
    int rc = a+b; private  
    return rc;  
}
```

Shared Variables & Private Variables (2)



We let x , y be shared, and z be private



*Single instance
for each x , y*

*Each thread has
its own instance for z*

- When a thread updates a shared variable, other threads are affected
 - We should be careful and careful!



Pitfall in Nested Loops (1)

- The following sample looks ok, but there is a bug
 - We do not see compile errors, but answers would be wrong ☹️

```
int i, j;  
#pragma omp parallel  
#pragma omp for  
for (i = 0; i < m; i++) {  
    for (j = 0; j < n; j++) {  
        ...  
    }  
}
```

Both i, j are declared
outside parallel region
→ Considered “shared”
It is a problem to share **j**

cf)

Thread A is executing i=5 loop
Thread B is executing i=8 loop



The executions should be independent
Each execution must include
j=0, j=1...j=n-1 correctly

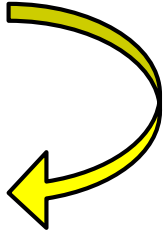
j must be private



Pitfall in Nested Loops (2)

Two modifications (Either is ok)

```
int i;  
#pragma omp parallel for  
for (i = 0; i < m; i++) {  
    int j;    // j is private  
    for (j = 0; j < n; j++) {  
        ...  
    } }  
}
```



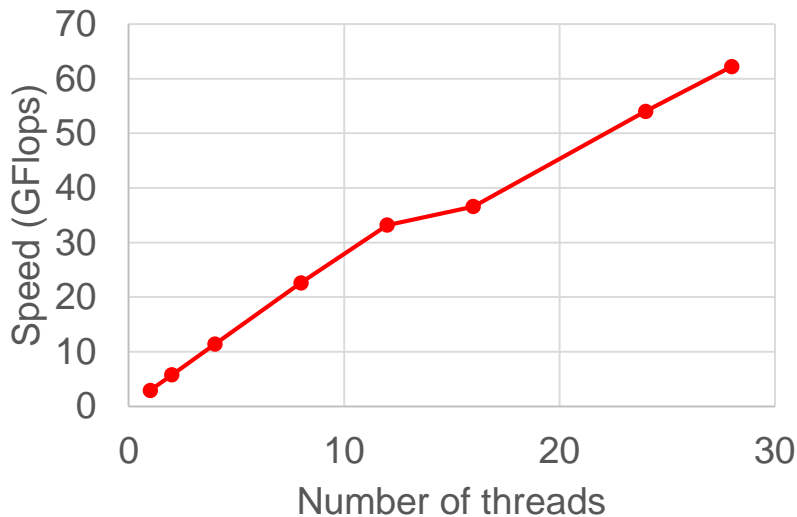
```
int i, j;  
#pragma omp parallel for private(j)  
// j is forcibly private  
for (i = 0; i < m; i++) {  
    for (j = 0; j < n; j++) {  
        ...  
    } }  
}
```

Performance of mm sample

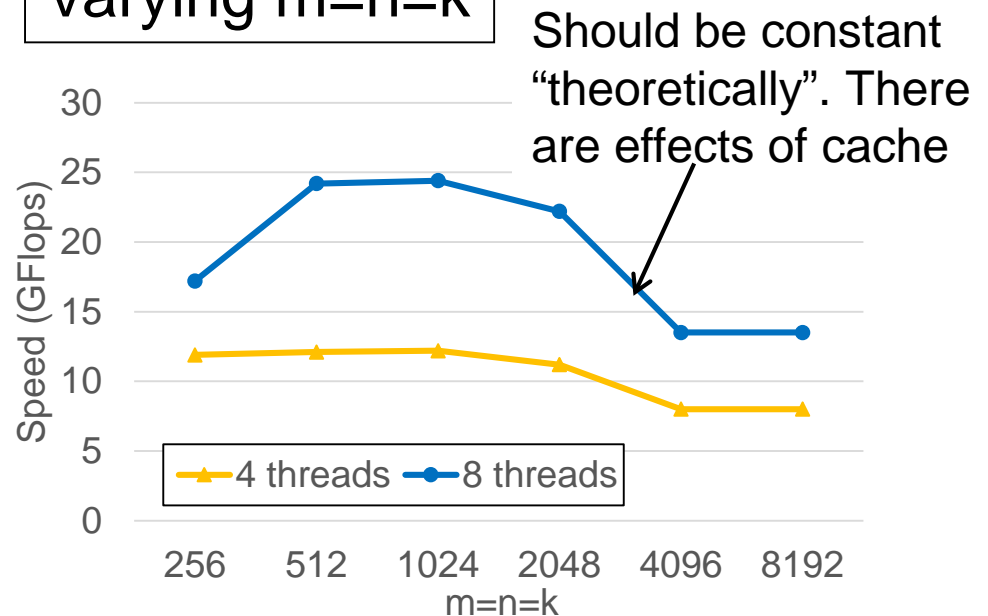


- A TSUBAME3 node (Xeon E5-2680 v4 x2 = 28core)
- Speed is (2mnk/t)

m=n=k=2048,
Varying # of threads



8 threads,
Varying m=n=k



OpenMP Version of mm (Again)



- One of loops is parallelized

```
#pragma omp parallel private(i,l)
```

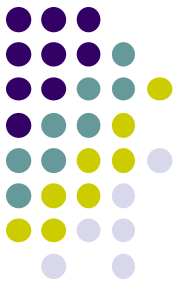
```
#pragma omp for
```

```
for (j = 0; j < n; j++) {           ← j loop is parallelized
    for (l = 0; l < k; l++) {
        for (i = 0; i < m; i++) {
            C[i+j*ldc] += A[i+l*lda] * B[l+j*ldb];
        } } }
```

[Q] What if we parallelize other loops?

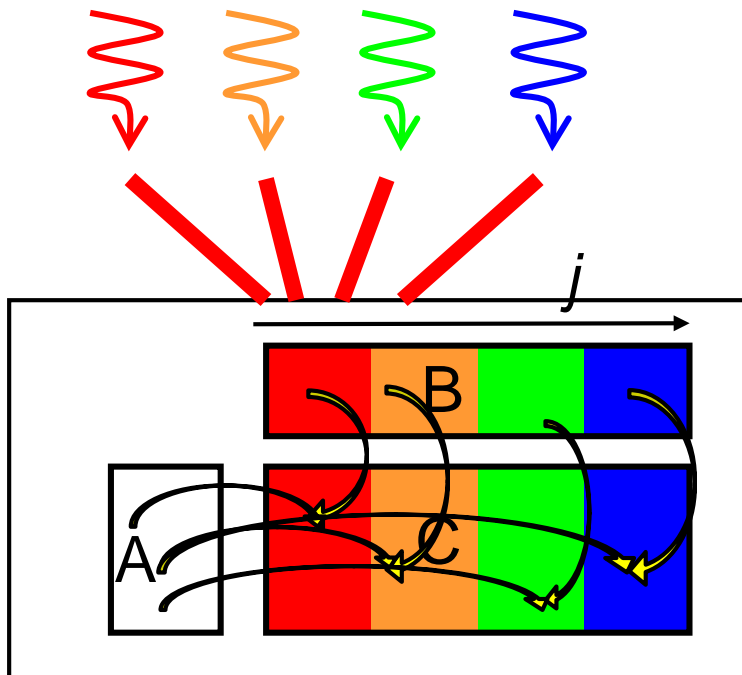
→ *i* loop is ok for correct answers, but may be slow

→ *l* loop causes wrong answers!



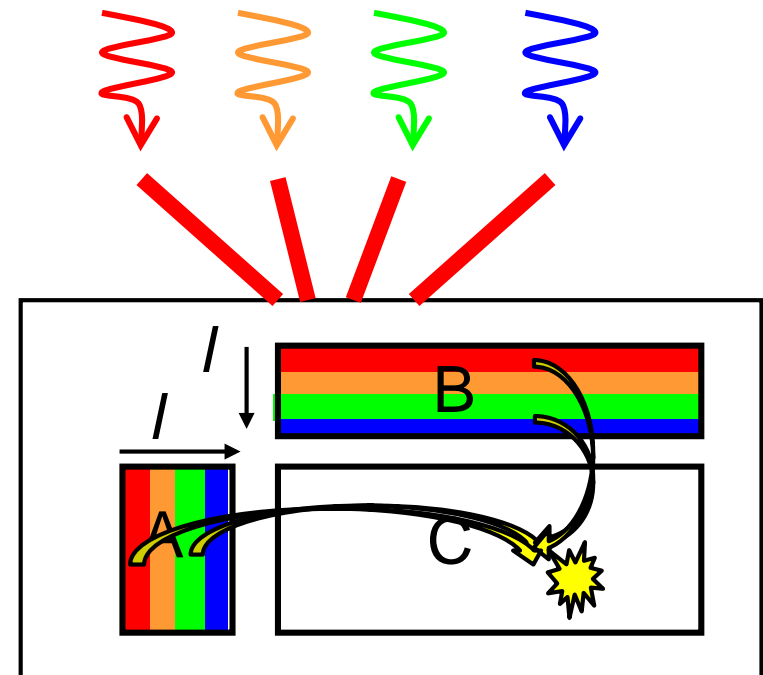
How Multiple Threads Work

Parallelizing j loop



Simultaneous **read**
(in this case, A) is OK
Similarly, parallelizing
 i loop is ok

Parallelizing i loop (??)



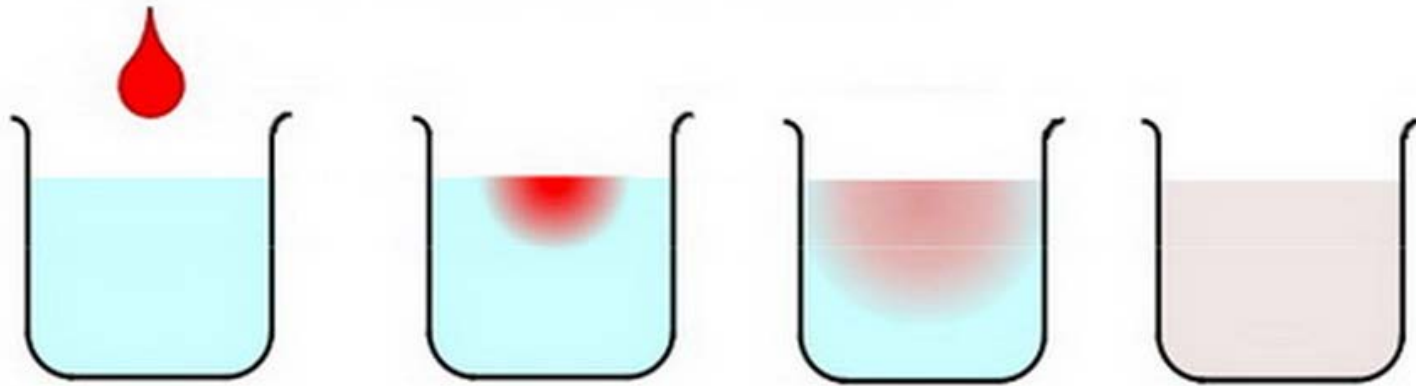
Possible simultaneous **write**
→ “Race condition” problem
may occur.
Answers may be wrong !!

“diffusion” Sample Program (1)



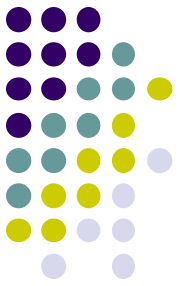
An example of diffusion phenomena:

- 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
 - cf) Weather forecast compute wind speed, temperature, air pressure...



“diffusion” Sample Program (2)

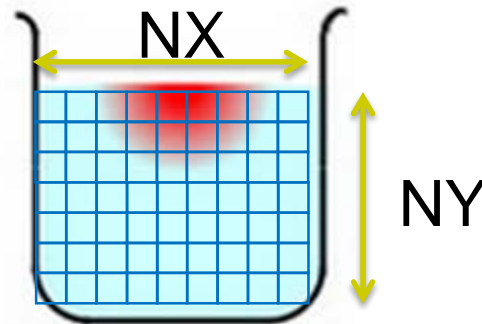
Available at [~endo-t-ac/ppcomp/18/diffusion/](https://github.com/endo-t-ac/ppcomp/18/diffusion/)

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

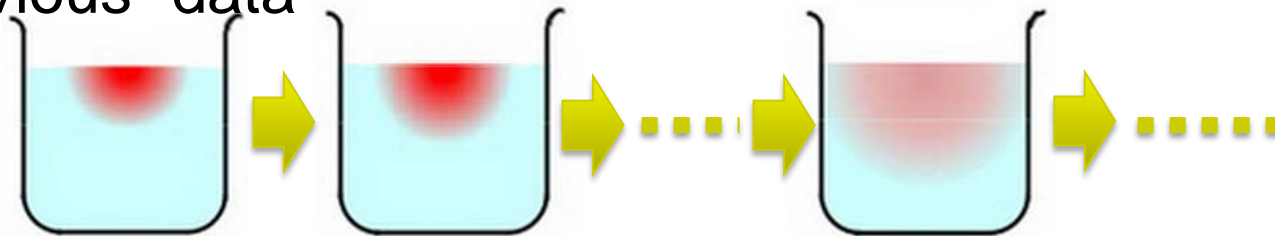


Data Structures in diffusion

- 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 $t=0$

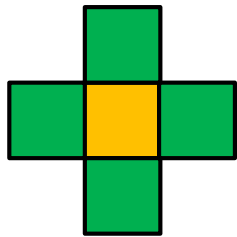
$t=1$

$t=20$



Stencil Computations

- A data point (x,y) at time t is computed using following data at time $t-1$ (previous data)
 - point (x,y)
 - “Neighbor” points of (x,y)



time $t-1$



time t

Points at
boundary
require special
treatments

- Computations of similar type is called “**stencil computation**”
- The followings must be given beforehand
 - All data at time step 0 (**Initial condition**)
 - Data in “boundary” points for every time step (**Boundary condition**)

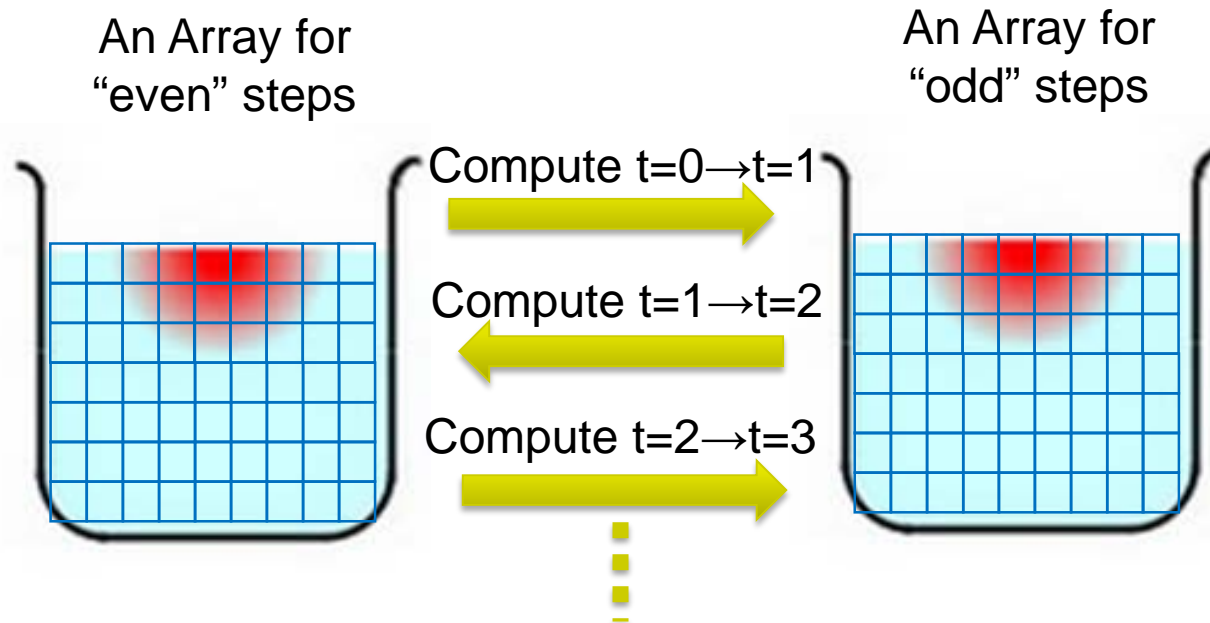


Original meanings of
“stencil”

Double Buffering Technique



- 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 We Parallelize “diffusion” sample (Related to Assignment [O1])



The program mainly uses “for” loops. So “omp parallel for” looks good.

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

[Hint] Parallelizing one of spatial (x, y) would be good. Spaces are divided into multiple threads

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

Assignments in OpenMP Part (Abstract)



Choose one of [O1]—[O3], and submit a report
Due date: May 7 (Monday)

- [O1] Parallelize “diffusion” sample program by OpenMP.
(~endo-t-ac/ppcomp/18/diffusion/ on TSUBAME)
- [O2] Parallelize “sort” sample program by OpenMP.
(~endo-t-ac/ppcomp/18/sort/ on TSUBAME)
- [O3] (Freestyle) Parallelize *any* program by OpenMP.

For more detail, please see No.3 slides or OCW-i.



Next Class:

- OpenMP(3)
 - “task parallelism” for programs with irregular structures
 - sort: Quick sort sample
 - Related to assignment [O2]