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

2025 Class No.6
[OpenMP Part] (4)
Bottleneck, Race Condition etc.

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

Overview of This Course

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

Today's Topic

- Bottleneck, mutual exclusion, reduction in OpenMP

  samples:

  - base/lumm, omp/lumm (Optional)
  - base/pi, omp/pi-bad, omp/pi-slow-omp, omp/pi-fast, omp/pi

“lumm” sample: LU Matrix Multiply

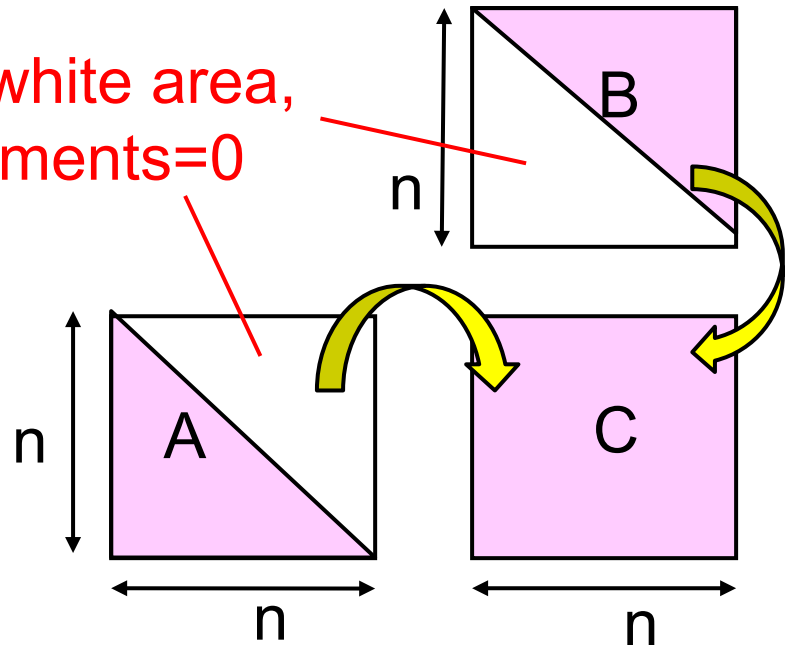
ppcomp-ex/base/lumm/

A: a (n×n) matrix
B: a (n×n) matrix
C: a (n×n) matrix

Square matrices

C ← A B

In white area, elements=0



- Execution: ./lumm [n]

lumm is similar to mm sample, but

- A is a Lower triangular matrix
- B is an Upper triangular matrix

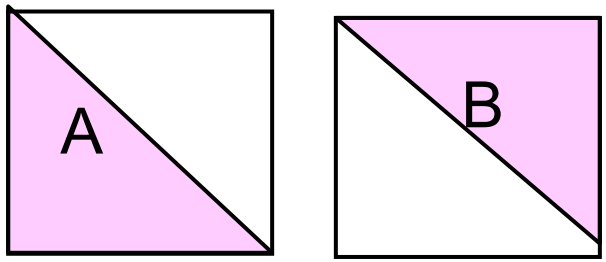
Difference between "mm" and "lumm"

- (usual) Matrix multiply

```
for (j = 0; j < n; j++) {
  for (l = 0; l < k; l++) {
    for (i = 0; i < m; i++) {
      Ci,j += Ai,l * Bl,j;
    } } }
```

$2n^3$ computation

If we know $A_{i,l} = 0$ or $B_{l,j} = 0$, we can skip computation



Computation in "lumm"

- LU Matrix multiply

```
for (j = 0; j < n; j++) {
  for (l = 0; l <= j; l++) {
    for (i = l; i < n; i++) {
      Ci,j += Ai,l * Bl,j;
    } } }
```

(2/3)n^3 computation

Comparing time between
"mm 2000 2000 2000" and "lumm 2000"

	1thread
mm	1.77 (sec)
lumm	0.539 (sec)
mm / lumm	3.28

→ Shorter time in lumm ☺
(around 1/3)

“lumm-omp”: OpenMP version

ppcomp-ex/omp/lumm/

```
#pragma omp parallel private(l,l)
#pragma omp for
for (j = 0; j < n; j++) {
  for (l = 0; l <= j; l++) {
    for (i = l; i < m; i++) {
      Ci,j += Ai,l * Bl,j;
    } } }
```

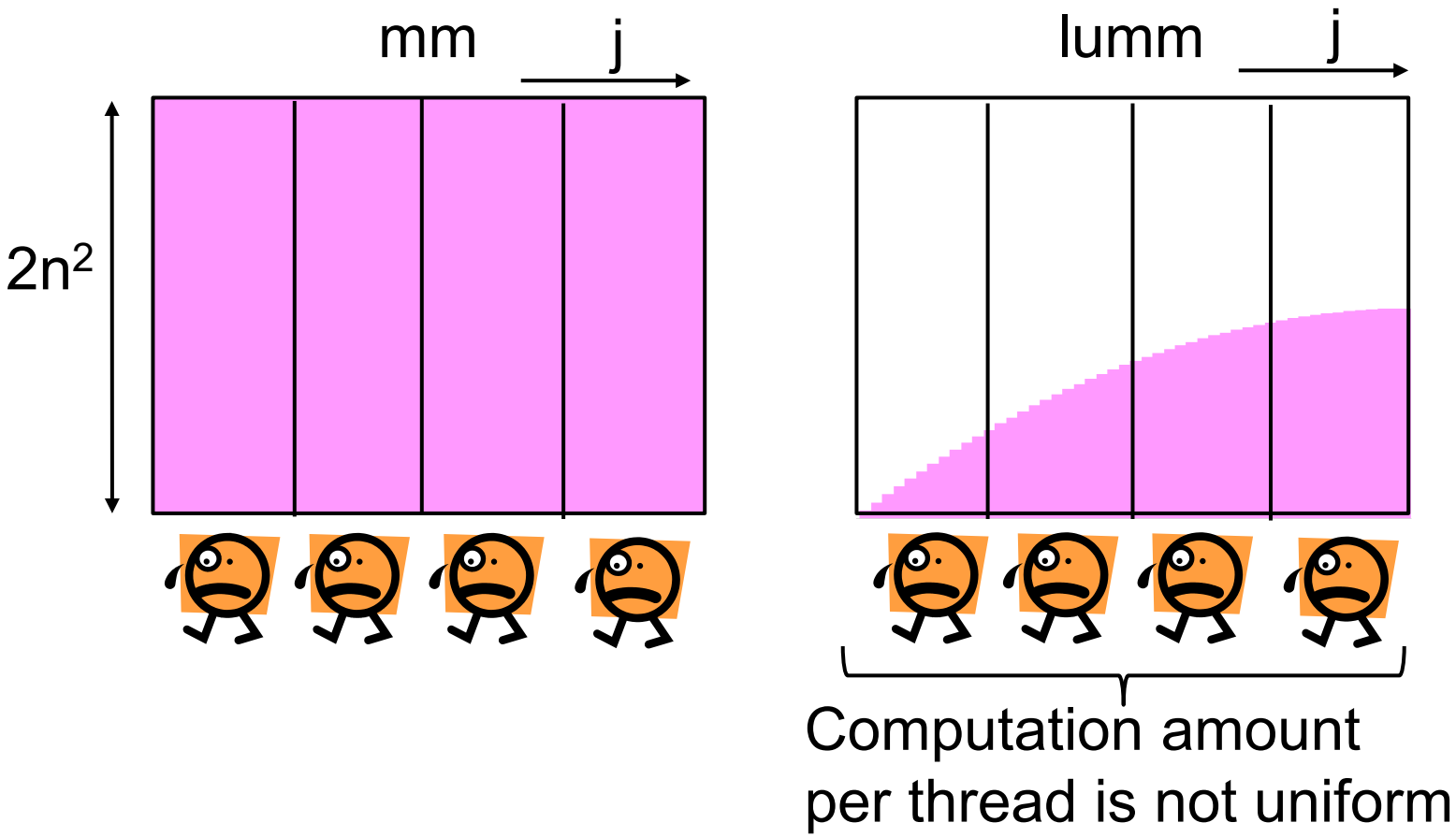
memo:
github’s version is already fast. please edit

	1thread	4threads	8threads
mm-omp	1.77 (sec)	0.470	0.237
lumm-omp	0.539 (sec)	0.210	0.105
mm-omp / lumm-omp	3.28	2.24	2.26

lumm-omp is faster, but the ratio gets worse. Why?

Effects of Load Imbalance

- In lumm, <u>computation amount for each j</u> is not uniform

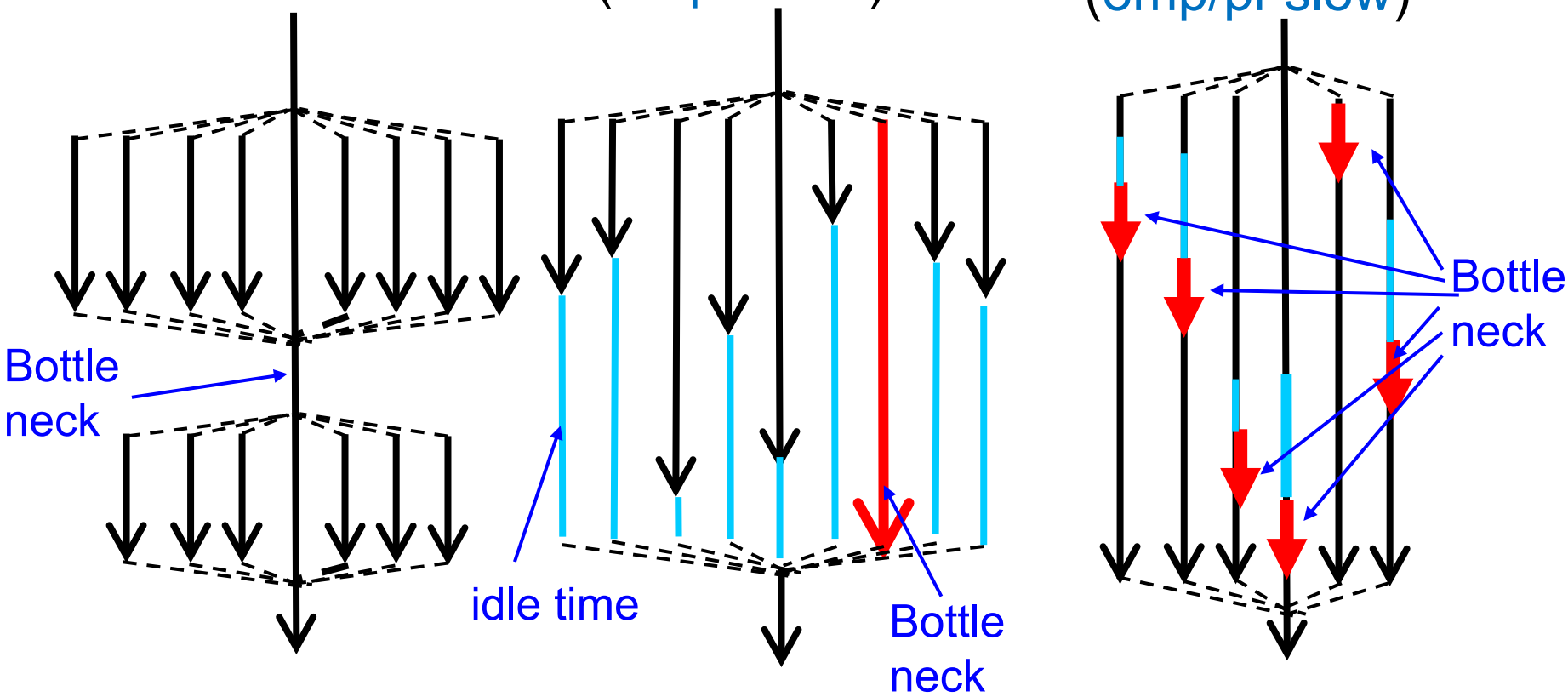


Various Bottlenecks

Bottleneck by sequential part

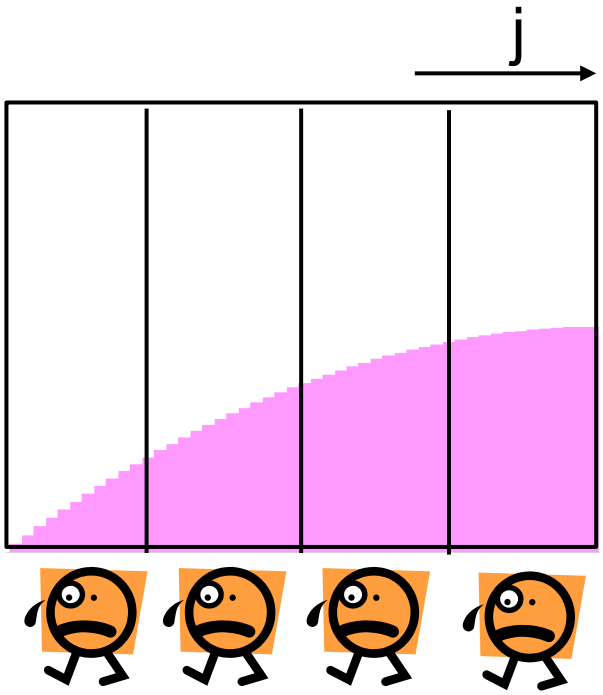
Bottleneck by load imbalance (omp/lumm)

Bottleneck by critical sections (omp/pi-slow)



Moreover, There are architectural bottlenecks

Improvement of lumm-omp



- Imbalance is caused by the default rule of “omp for”
  - “block distribution”
- Rule of “omp for” can be changed by schedule option

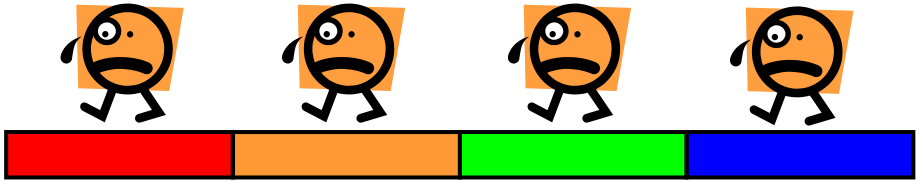
#pragma omp for schedule (…)

Changing "schedule" of omp for

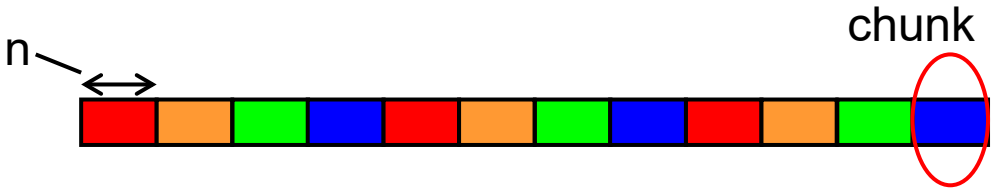
- OpenMP provides several scheduling methods (mapping between iteration and threads)

```
#pragma omp for schedule(···)
```

`schedule(static)`
Uniform block distribution (default)



`schedule(static, n)`
Cyclic distribution
n is "chunk" size



`schedule(dynamic, n)`
An Idle thread take a new chunk

`schedule(guided, n)`
Similar to dynamic, but
"chunk size" gets gradually smaller

[Q] In lumm-omp, #pragma omp for schedule(static,1) works better than #pragma omp for. Why?

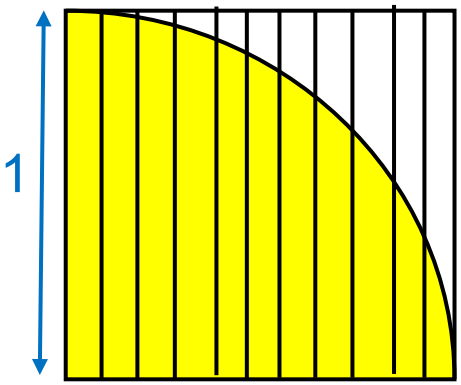
“pi” sample

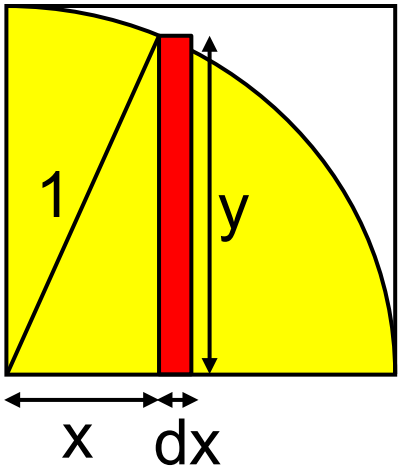
Compute an approximation of π = 3.14159…
(circumference/diameter)

- ppcomp-ex/base/pi/
- Method
  - SUM ← Approximation of the yellow area
  - π ← 4 x SUM
- Execution : ./pi [n]
  - n: Number of division
  - Cf) ./pi 100000000
- Compute complexity : O(n)

Note: This program is only for a simple sample.
π is usually computed by different algorithms.

Quarter circle





dx = 1/n
y = sqrt(1-x*x)

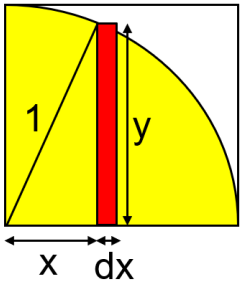
Algorithm of “pi”

```
double pi(int n) {
    int i;
    double sum = 0.0;
    double dx = 1.0 / (double)n;




    for (i = 0; i < n; i++) {
        double x = (double)i * dx;
        double y = sqrt(1.0 - x*x);
        sum += dx*y;
    }

    return 4.0*sum; }
```



Simple Parallelization of “pi” ??

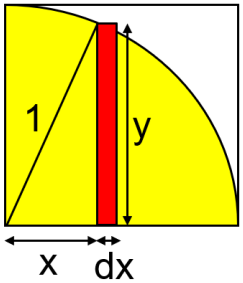
```
double pi(int n) {
    int i;
    double sum = 0.0;
    double dx = 1.0 / (double)n;

#pragma omp parallel
#pragma omp for

    for (i = 0; i < n; i++) {
        double x = (double)i * dx;
        double y = sqrt(1.0 - x*x);
        sum += dx*y;
    }

    return 4.0*sum; }
```

➔ ok???



- This version has a bug!

Several Versions of pi Sample

- base/pi : sequential version

Followings use OpenMP

- omp/pi-bad :
  - *"#pragma omp parallel for"* is simply used

  → This has **a bug** that produces incorrect results
- omp/pi-slow : results are correct, but slow
- omp/pi-fast : results are correct and faster
- omp/pi : similar to pi-fast-omp but uses "reduce" option

do "make" and execute by "./pi 100000000"

Caution: since pi-slow is very slow, please try ./pi 1000000 (smaller N)

Execution Sample

base/pi

approximation of π

```
./pi 100000000
Result=3.141592673589216: Pi took 0.378869 sec --> 0.264 Gsamples/sec
Result=3.141592673589216: Pi took 0.378757 sec --> 0.264 Gsamples/sec
Result=3.141592673589216: Pi took 0.378708 sec --> 0.264 Gsamples/sec
Result=3.141592673589216: Pi took 0.378767 sec --> 0.264 Gsamples/sec
Result=3.141592673589216: Pi took 0.379036 sec --> 0.264 Gsamples/sec
```

omp/pi-bad

```
export OMP_NUM_THREADS=8
./pi 100000000
Result=0.561908397695098: Pi took 0.736543 sec --> 0.136 Gsamples/sec
Result=0.549626144994676: Pi took 0.762682 sec --> 0.131 Gsamples/sec
Result=0.561650379038136: Pi took 0.731312 sec --> 0.137 Gsamples/sec
Result=0.526911533756178: Pi took 0.612839 sec --> 0.163 Gsamples/sec
Result=0.414727757933077: Pi took 0.489859 sec --> 0.204 Gsamples/sec
```

Results are far from π!
Also they change largely

slower than base/pi

Can We Parallelize the loop in pi?

- Let us consider computations with different i

C1 (i=i1)

```
x = (double)i * dx;
y = sqrt(1.0 - x*x);
sum += dx*y;
```

these parts are independent

dependent

C2 (i=i2)

```
x = (double)i * dx;
y = sqrt(1.0 - x*x);
sum += dx*y;
```

R(C1) = {sum,dx}, W(C1) = {sum}

R(C2) = {sum,dx}, W(C2) = {sum}

※ Here, private variables x, y and loop counter i are safely omitted

- $W(C1) \cap W(C2) \neq \emptyset$ ➔ C1 and C2 are Dependent!

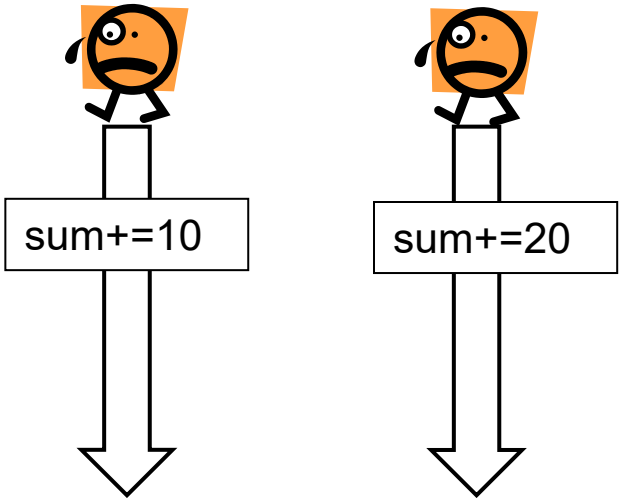
➔ Do we have to abandon parallel execution?

Race Condition Problem (1)

- To discuss the problem in omp/pi-bad, we consider a simpler program

  export OMP_NUM_THREADS=2

```
int sum = 0;
#pragma omp parallel      // 2 threads
{
   if (omp_get_thread_num() == 0)
      sum += 10;    // C0 (by thread 0)
   else
      sum += 20;    // C1 (by thread 1)
}
```



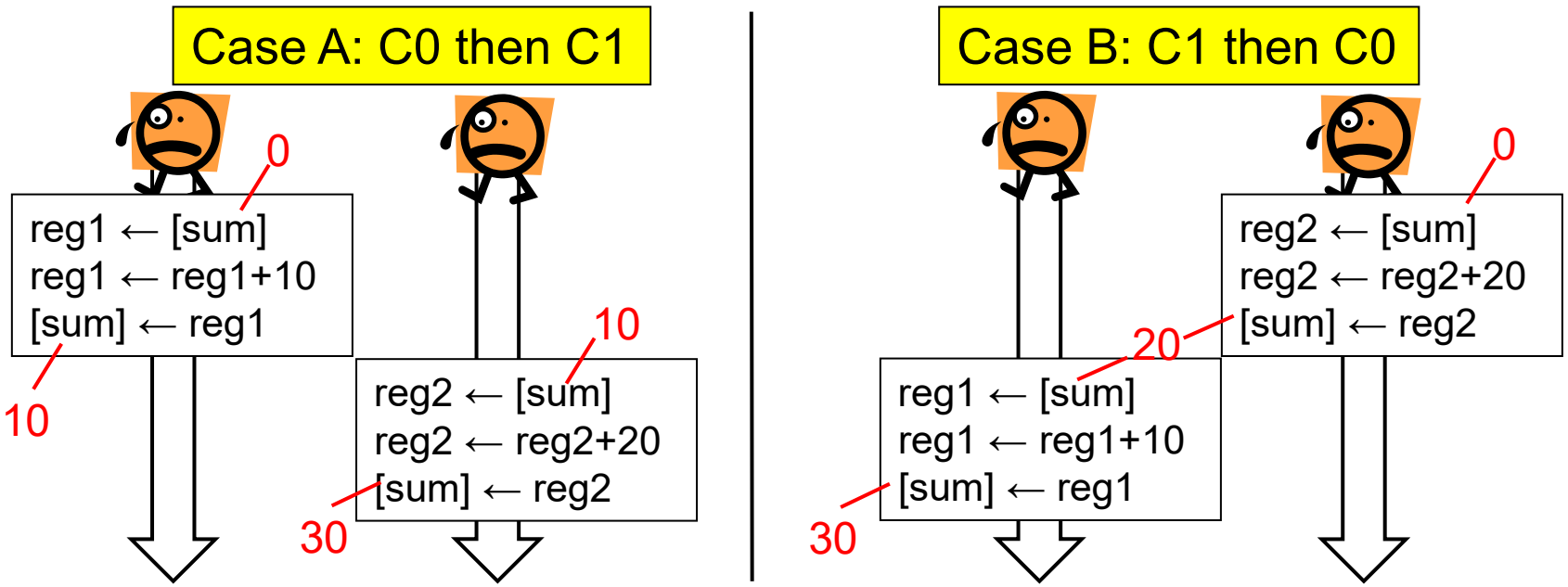
After this program, we expect “sum == 30”. It it ok??

Race Condition Problem (2)

- [Q] Does execution order of C0 and C1 affect the results?
  - Note: "sum += 10" is compiled into machine codes like

```
reg1 ← [sum]
reg1 ← reg1+10
[sum] ← reg1
```

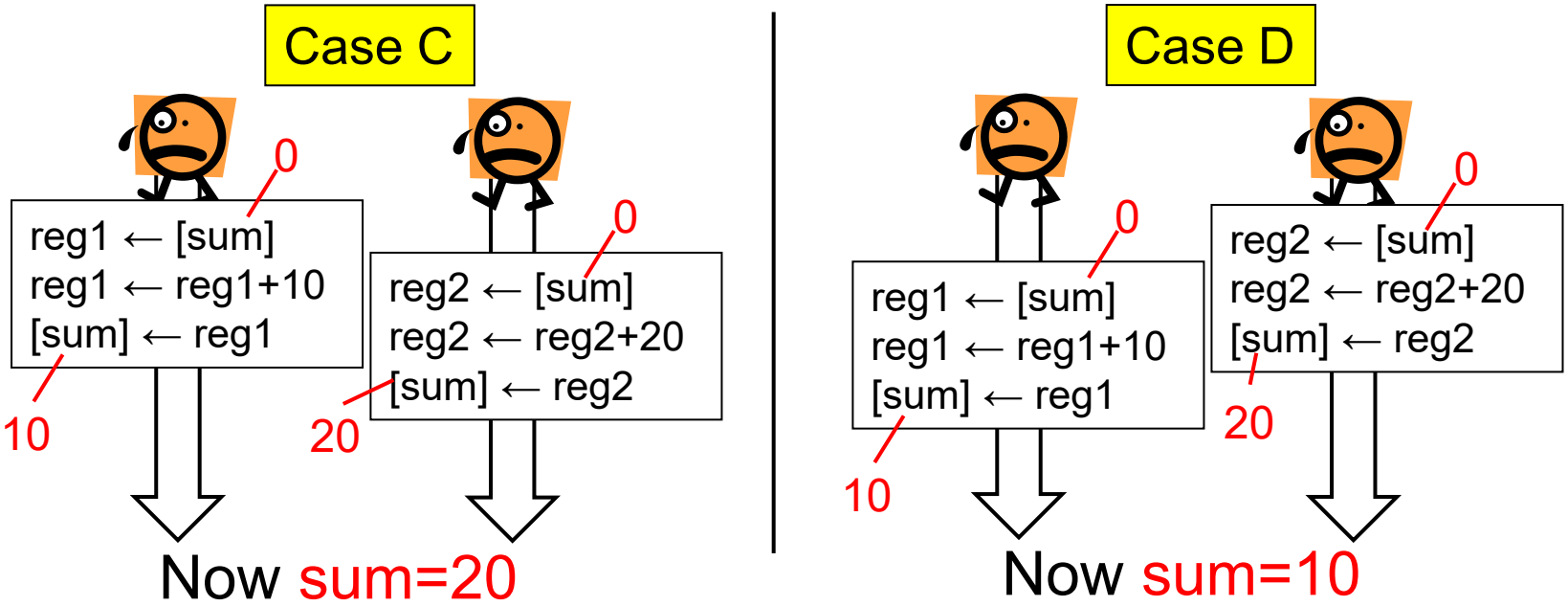
※ reg1, reg2… are CPU registers, which are thread private



The results are same: sum=30. Ok to parallelize???

Race Condition Problem (3)

- No!!! The results can be different if C0 & C1 are executed (almost) simultaneously



The expected result is 30, but we may get bad results

Such a bad situation is called "Race Condition"

➔ This problem occurs in "pi-bad-omp"

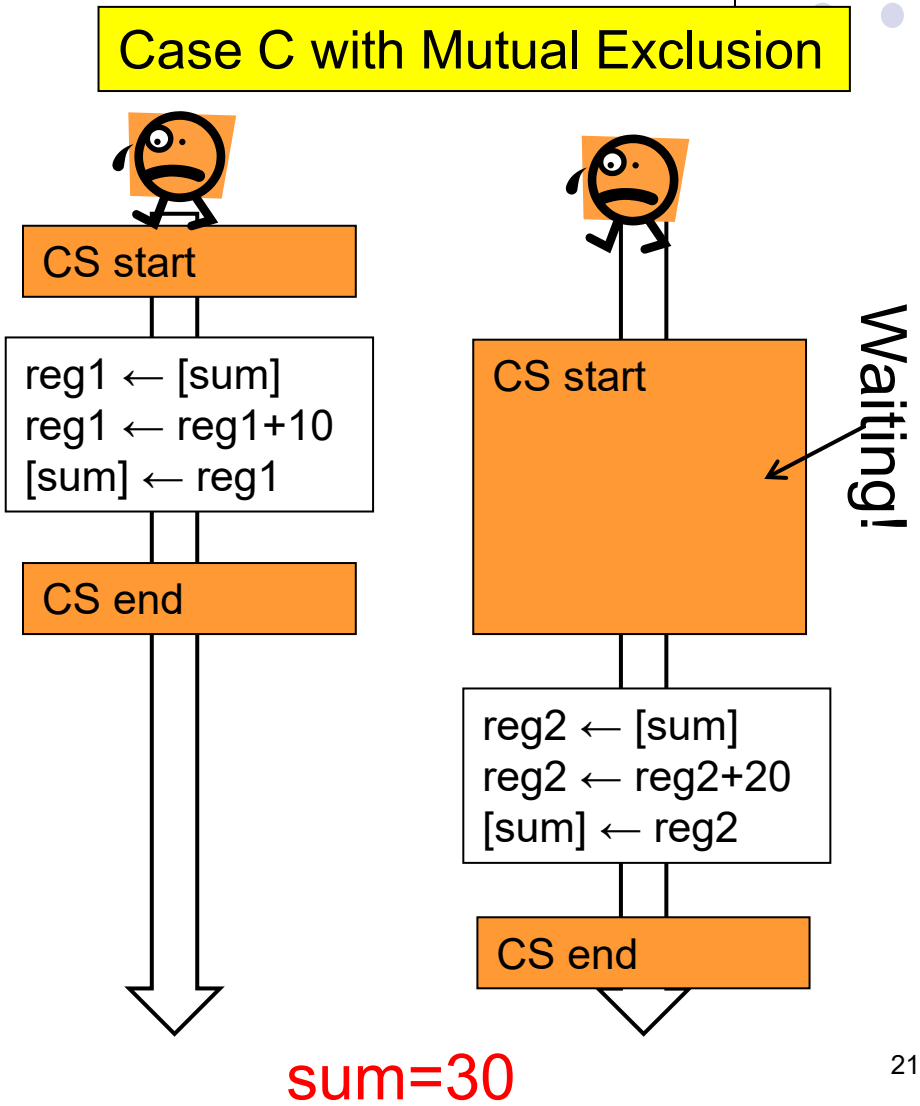
Mutual Exclusion to Avoid Race Condition

Mutual exclusion (mutex):

Mechanism to control threads so that only a single thread can enter a "specific region"

- The region is called critical section

⇒ With mutual exclusion, race condition is avoided



Mutual Exclusion in OpenMP

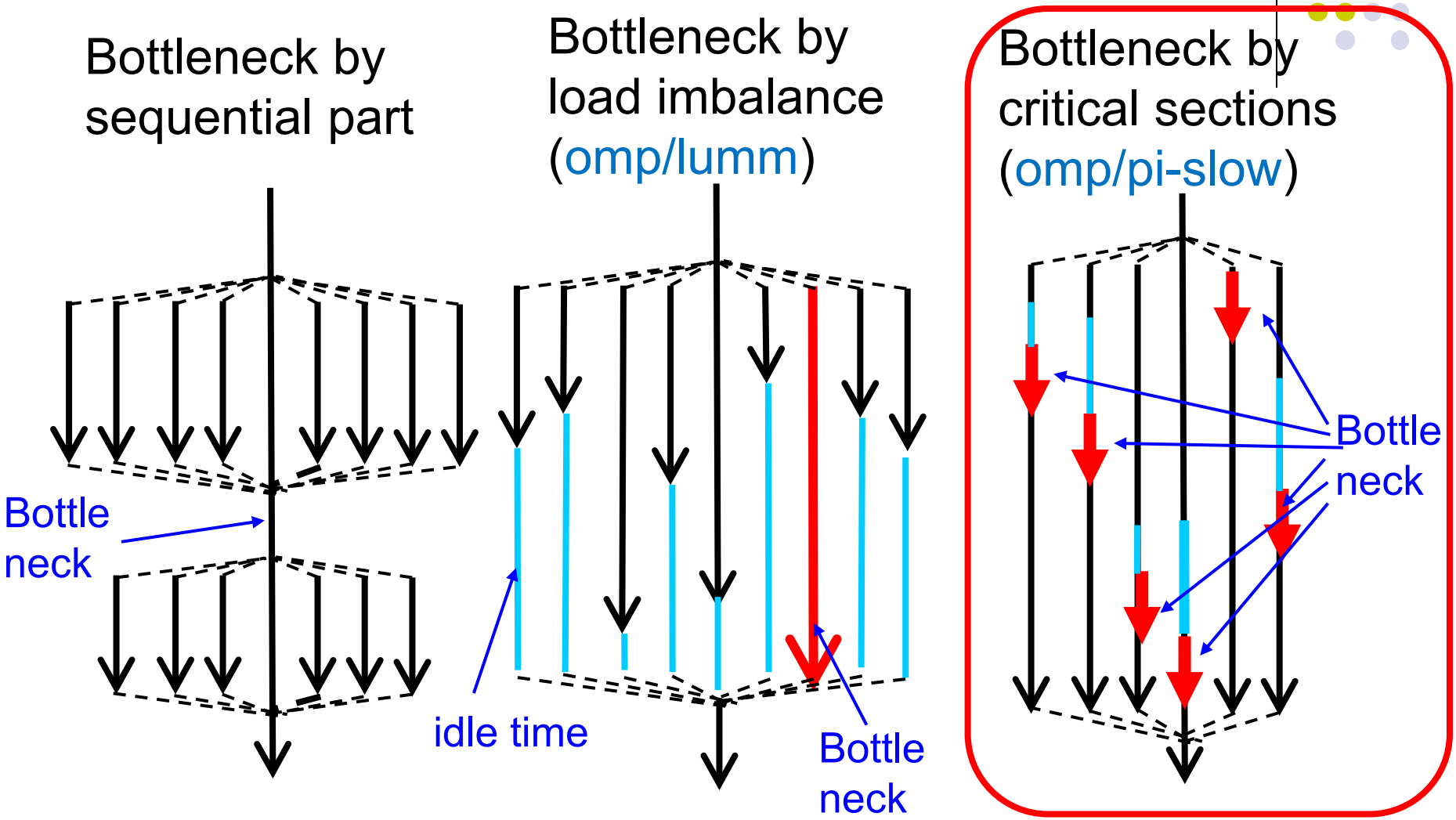
#pragma omp critical makes
the following block/sentence
be critical section

```
   double sum = 0;
#pragma omp parallel
   {
      [ do something ]
#pragma omp critical
      sum += myans;
   }
```

Caution
"omp/pi-slow" is very slow!
./pi 100000000 (10^8) takes too long,
so please use smaller N

cf) ./pi 1000000
(10^6)

We are Seeing Bottlenecks



Amdahl's Law: Modeling Bottlenecks

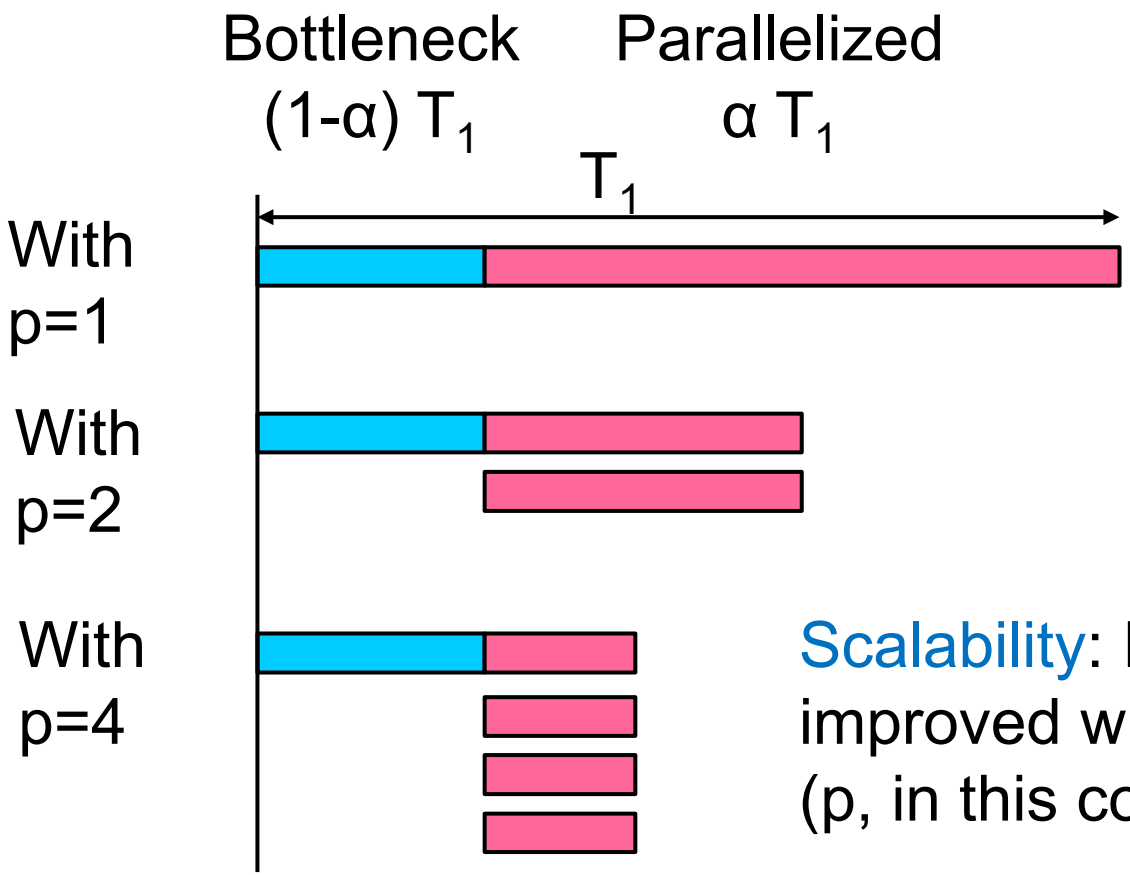
- We consider an algorithm. Then we let
  - T_1 : execution <u>time</u> with <u>1</u> processor core
  - α: ratio of computation that can be <u>parallelized</u>
  - $1-\alpha$: ratio that CANNOT be parallelized (<u>bottleneck</u>)

$\Rightarrow$ Estimated execution time with p processor cores is $T_p = ((1 - \alpha) + \alpha / p)\, T_1$

Due to bottleneck, there is limitation in speed-up no matter how many cores are used

$T_\infty = (1-\alpha)\, T_1$

An Illustration of Amdahl's Law



Scalability: How performance is improved with larger resources (p, in this context)

Amdahl's law tells us

- if α=0.9, we only achieve up to 10x speed up with ∞ cores
- if α=0.99, we only achieve up to 100x speed up with ∞ cores

The Fact is Harder Than Theory

- According to Amdahl's law, T_p is monotonically decreasing

→ Is large p always harmless ??

Performance comparison of pi-omp and pi-slow-omp

export OMP_NUM_THREADS= [p]

./pi 1000000000 # 10^9

p	omp/pi-fast omp/pi	omop/pi-slow
1	2.44 (sec)	5.59 (sec)
4	0.61	~60
8	0.305	~150
16	0.153	250~400

Slower! ☹

Reducing bottleneck is even more important (than Amdahl's law tells)

Reducing Bottlenecks

- Approaches for reducing bottlenecks depend on algorithms!
  - We need to consider, consider
  - Some algorithms are essentially difficult to be parallelized

- Some directions
  - Improving load balance
  - Reducing access to shared variables
  - Reducing length of dependency chains
    - called “critical path”
  - Reducing parallelization costs
    - entering/exiting “omp parallel”, “omp critical”… is not free

lumm sample

pi sample

:

Cases of "pi" Sample

- "omp/pi-slow" is slow, since each thread enters a critical section too frequently

→ To improve this, another omp/pi-fast version introduces private variables

Step 1: Each thread accumulates values into private "local_sum"

Step 2: Then each thread does "sum += local_sum" in a critical section once per thread

→ omp/pi-fast is fast ☺

→ But program is more complex ☹

Please see omp/pi-fast/pi.c

omp/pi-fast Algorithm

```
   double sum = 0.0;   // share var

#pragma omp parallel
   {
      double local_sum = 0.0;  // private var
#pragma omp for
      for (i = 0; i < n; i++) {
        double x = (double)i * dx;
        double y = sqrt(1.0 - x*x);
        local_sum += dx*y;
      }

#pragma omp critical  // critical section
      sum += local_sum;
   }
```

- This uses critical section, but much less often than pi-slow!

Reduction Computations in "omp for"

- *"Summation in a for-loop"* is one of typical computations
  → called reduction computations
- In OpenMP, they can be integrated to "omp for"

```
    double sum = 0.0;

#pragma omp parallel
#pragma omp for reduction (+:sum)
    for (i = 0; i < n; i++) {
        double x = (double)i * dx;
        double y = sqrt(1.0 - x*x);
        sum += dx*y;
    }
```

Operator is one of +, -, *, &&, ||, max, min, etc

Name of reduction variable

→ omp/pi is fast, like omp/pi-fast ☺
→ Also, programming is easier ☺

Why is “omp for reduction” Fast?

Why is omp/pi with reduction also fast?
“omp for reduction(…)” is internally compiled to a similar code to pi-fast-omp

```
    double sum = 0.0;

#pragma omp parallel


#pragma omp for reduction (+:sum)
    for (i = 0; i < n; i++) {
        double x = (double)i * dx;
        double y = sqrt(1.0 - x*x);
        sum += dx*y;
    }
```

automatically compiled

```
    double sum = 0.0;

#pragma omp parallel
    {
        double local_sum = 0.0;
#pragma omp for
        for (i = 0; i < n; i++) {
            double x = (double)i * dx;
            double y = sqrt(1.0 - x*x);
            local_sum += dx*y;
        }
#pragma omp critical
        sum += local_sum;
    }
```

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

Next Part: OpenACC (ACC) Part

- Class #7
  - Introduction to GPU programming and OpenACC
  - Parallelization of loops
  - Assignment in OpenACC part
- Class #8
  - Discussion on diffusion sample wit OpenACC
  - Data transfer cost