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

Part 1: OpenMP

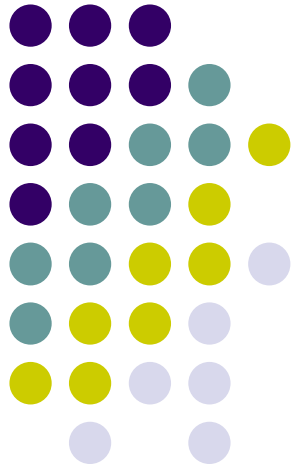
No 3: Bottleneck, Race Condition etc.

Apr 22, 2024

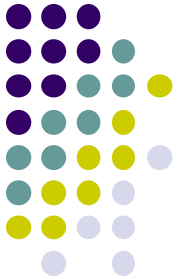
Toshio Endo

School of Computing & GSIC

endo@is.titech.ac.jp

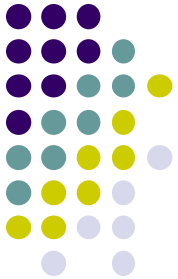


Overview of This Course



- Part 0: Introduction
  - 2 classes
- Part 1: OpenMP for shared memory programming
  - 4 classes ← We are here (3/4)
- Part 2: GPU programming
  - OpenACC and CUDA
  - 4 classes
- Part 3: MPI for distributed memory programming
  - 3 classes

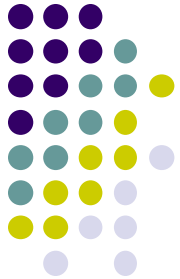
TSUBAME Maintenance



- TSUBAME4 is stopped for maintenance
  - 9:00, Mon, Apr 22 – 17:00, Wed, Apr 24 (plan)
  - TSUBAME4 portal is also stopped

  - Today, we can not use TSUBAME

Today's Topic



- Bottleneck, mutual exclusion, reduction, in OpenMP

  samples:

  - lumm, lumm-omp
  - pi, pi-bad-omp, pi-slow-omp, pi-fast-omp, pi-omp

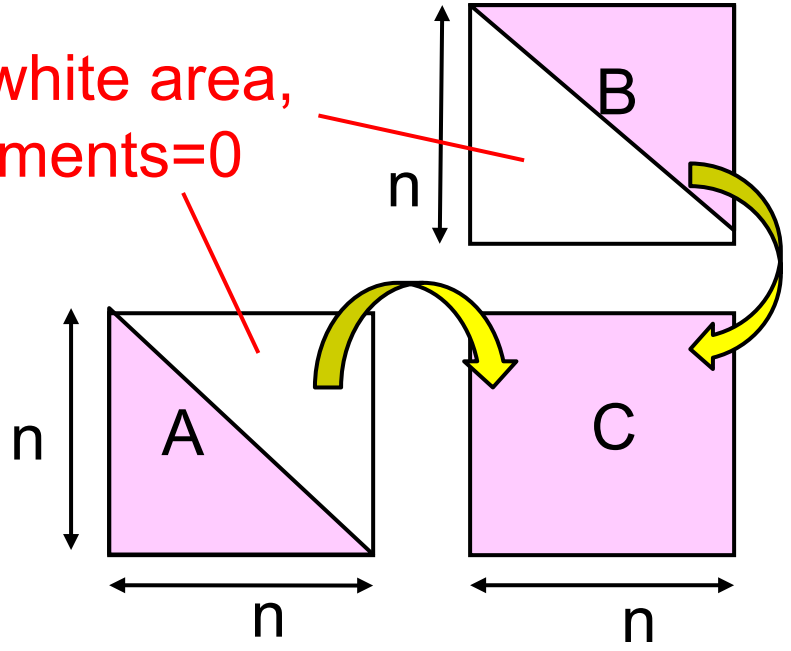
“lumm” sample: LU Matrix Multiply

Available at /gs/bs/tga-ppcomp/24/lumm/

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

} Square matrices

C ← A B

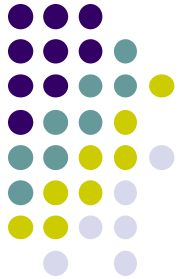


- Execution: ./lumm [n]

lumm is similar to mm sample, but

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

Difference between "mm" and "lumm"

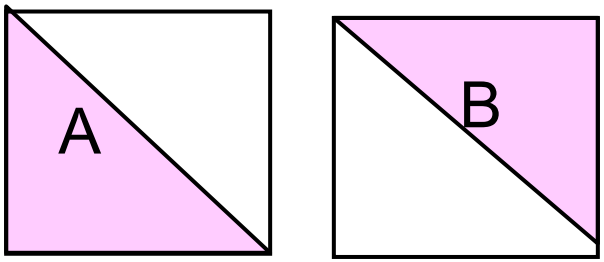


- Matrix multiply

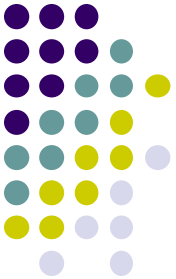
```
for (j = 0; j < n; j++) {
  for (l = 0; l < k; l++) {
    for (i = 0; i < m; i++) {
      Ci,j += Ai,l * Bl,j;
    } } }
```

$2n^3$ computation

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

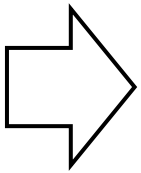


Computation in “lumm”



- 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;
    } } }
```



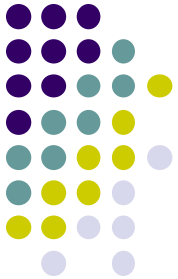
$(2/3)n^3$ computation

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 ☺

“lumm-omp”: OpenMP version



Available at /gs/bs/tga-ppcomp/24/lumm-omp/

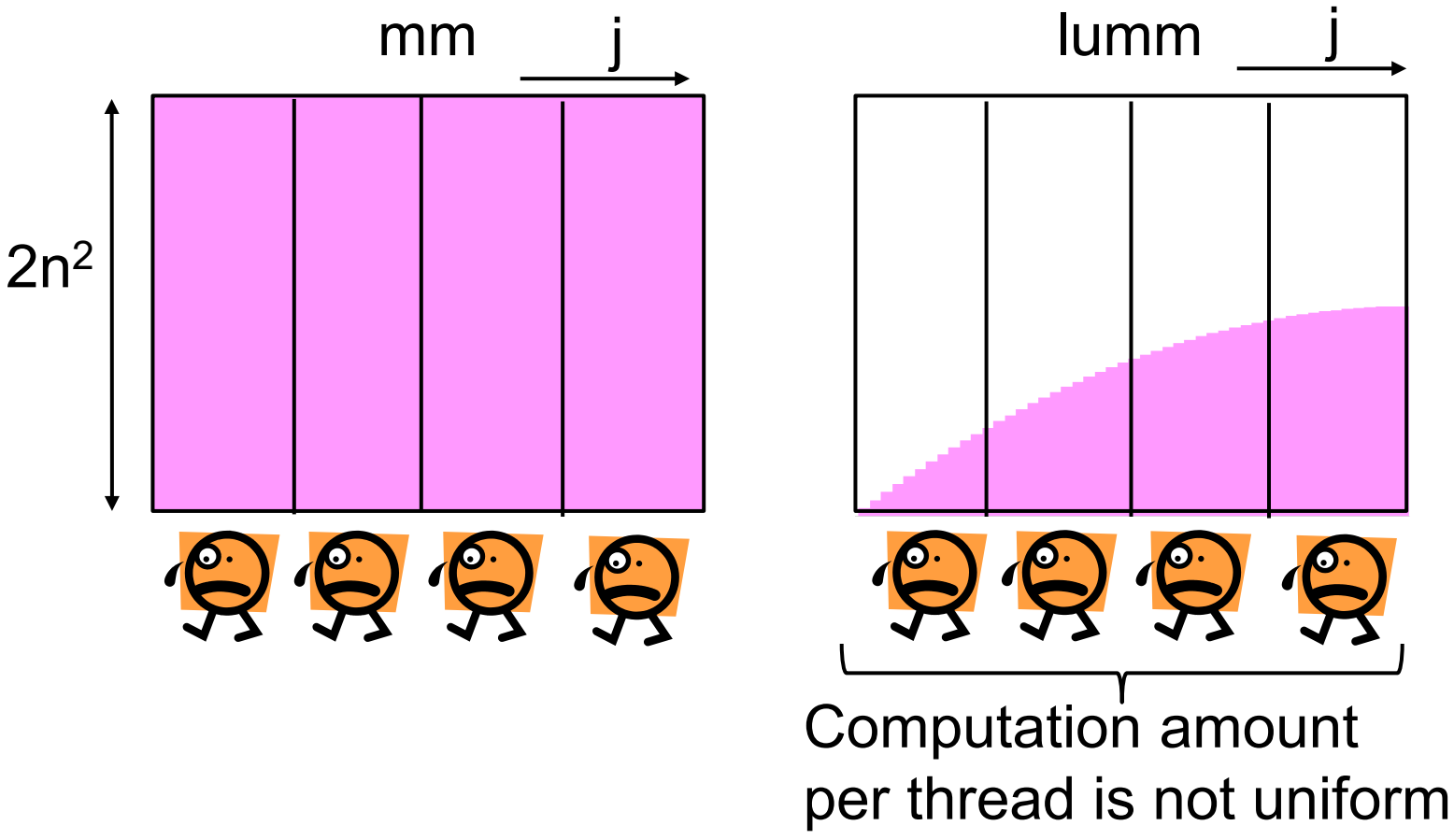
```
#pragma omp parallel private(l,l)
#pragma omp for
for (j = 0; j < n; j++) {
  for (l = 0; l <= j; l++) {
    for (i = l; i < m; i++) {
      Ci,j += Ai,l * Bl,j;
    } } }
```

	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, computation amount for each j is not uniform



Towards "Fast" Parallel Software

- If the entire algorithm is divided into independent computations (such as mm example), the story is easy
- But generally, some algorithms may include
  - Computations that converge on specific threads (as in lumm)
  - Computations that can NOT be parallelized (as in pi)

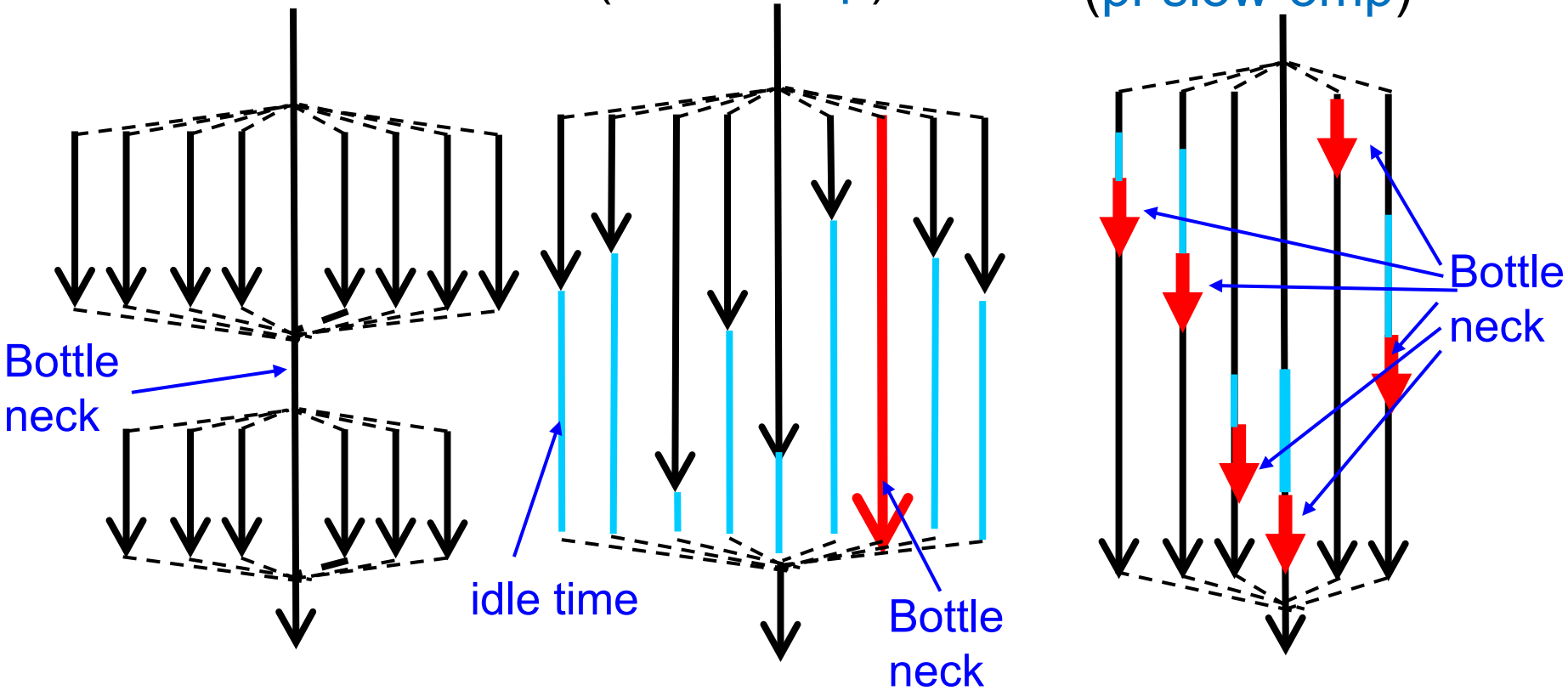
⇒ These parts raise problems called "bottleneck"

Various Bottlenecks

Bottleneck by sequential part

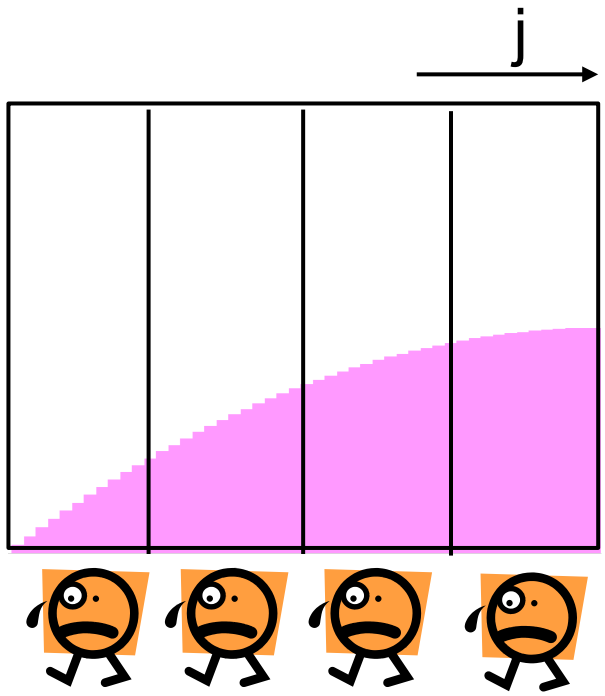
Bottleneck by load imbalance (lumm-omp)

Bottleneck by critical sections (pi-slow-omp)



Moreover, There are architectural bottlenecks

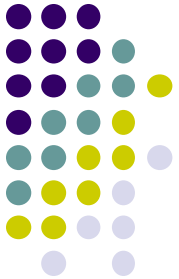
Improvement of lumm-omp



- 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)

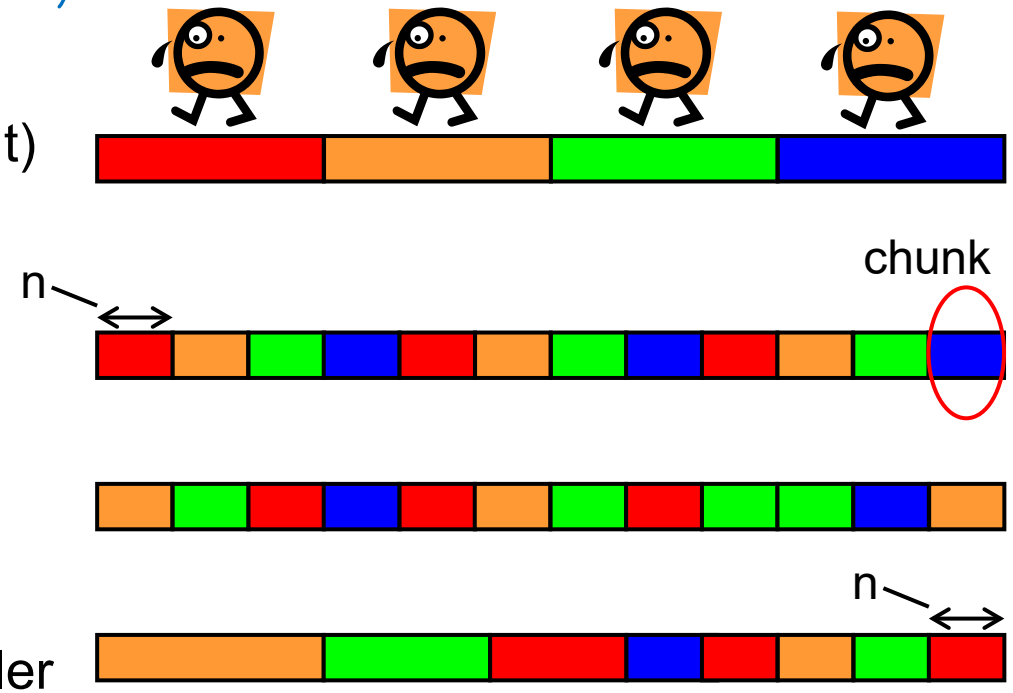
  `#pragma omp for schedule(···)`

`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. Why?

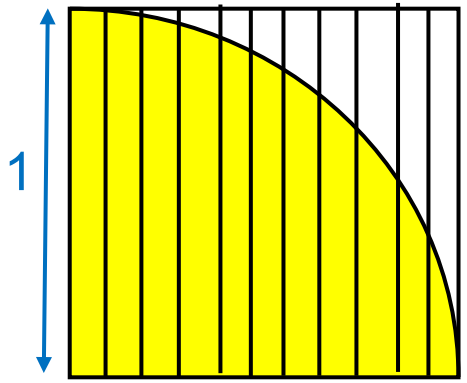
“pi” sample

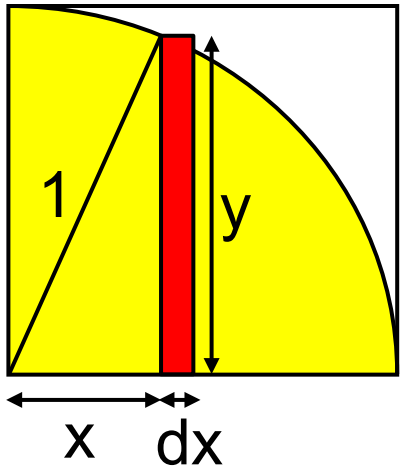
Compute an approximation of π = 3.14159…
(circumference/diameter)

- Available at /gs/bs/tga-ppcomp/24/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

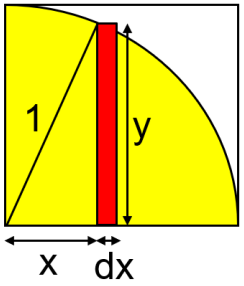




dx = 1/n
y = sqrt(1-x*x)

Algorithm of “pi” (1)

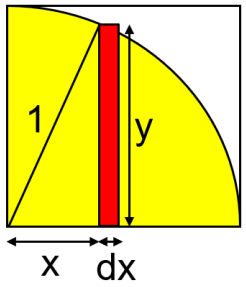
```
double pi(int n) {
    int i;
    double sum = 0.0;
    double dx = 1.0 / (double)n;



    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; }
```



Algorithm of “pi” (2)

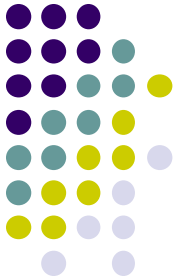
```
double pi(int n) {
    int i;
    double sum = 0.0;
    double dx = 1.0 / (double)n;

#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!
  pi-bad-omp sample
- We have to consider read&write access to sum, a shared variable

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 omitted

- W(C1) ∩ W(C2) ≠ ∅ ➔ Dependent!

➔ Do we have to abandon parallel execution?

Several Versions of pi Sample

- pi: sequential version

Followings use OpenMP

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

  → It has a bug that produces incorrect results
- pi-slow-omp: results are correct, but slow
- pi-fast-omp: results are correct and faster
- pi-omp: same as pi-fast-omp but uses "reduce" option

All are in /gs/bs/tga-ppcomp/24/

Race Condition Problem (1)

- To discuss the problem in pi-bad-omp, 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)
}
```

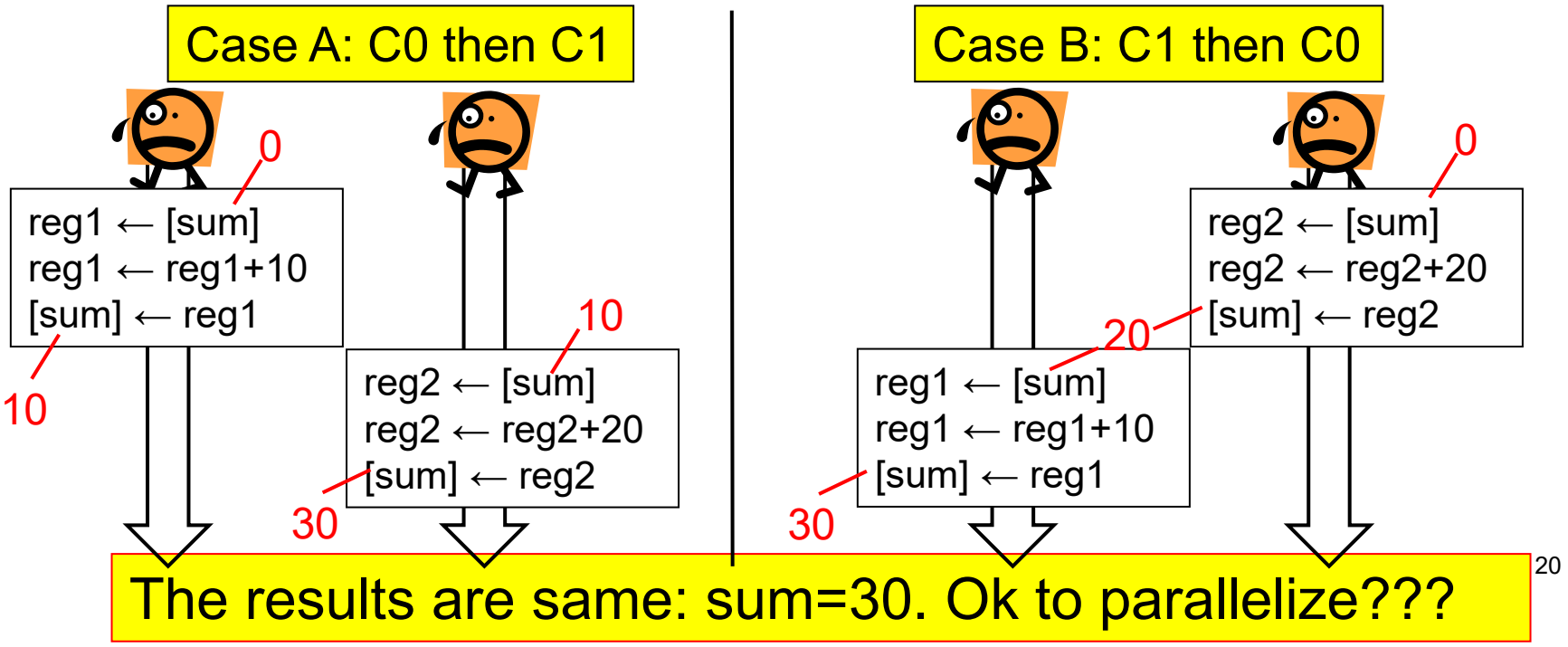
After this program, does “sum” equal to 30?

Race Condition Problem (2)

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

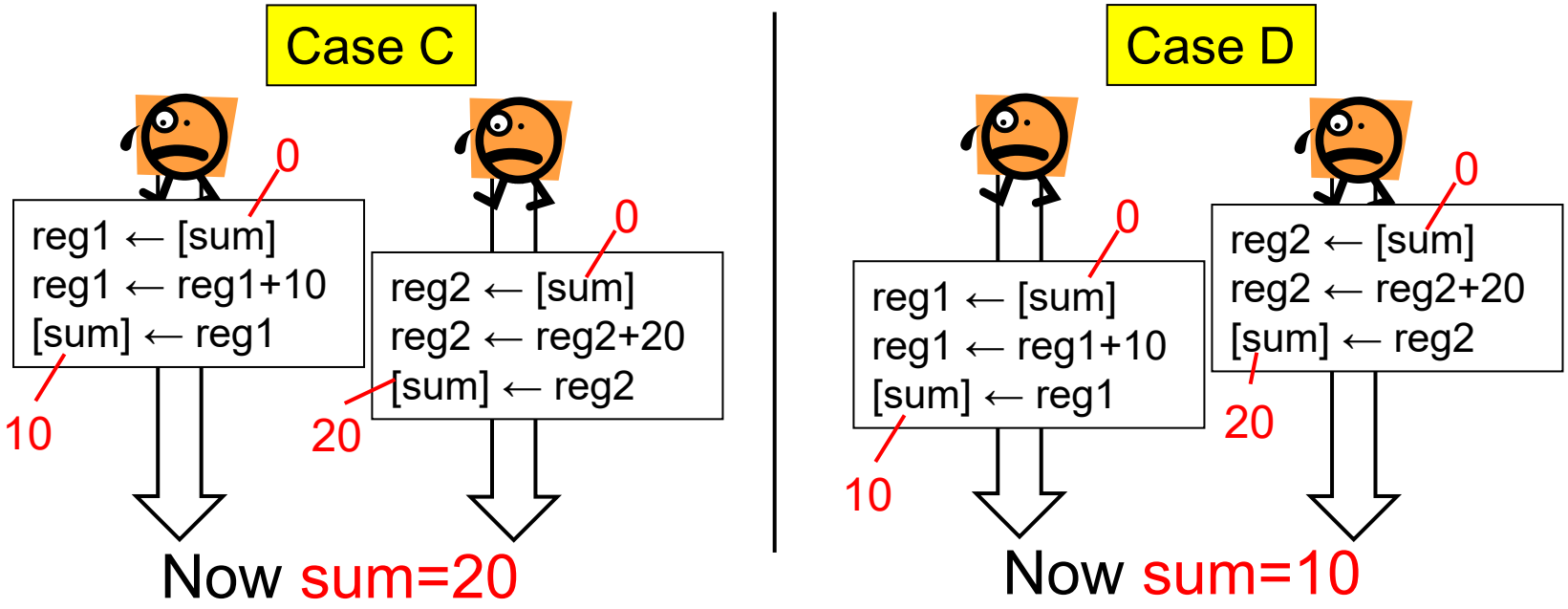
```
reg1 ← [sum]
reg1 ← reg1+10
[sum] ← reg1
```

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



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"

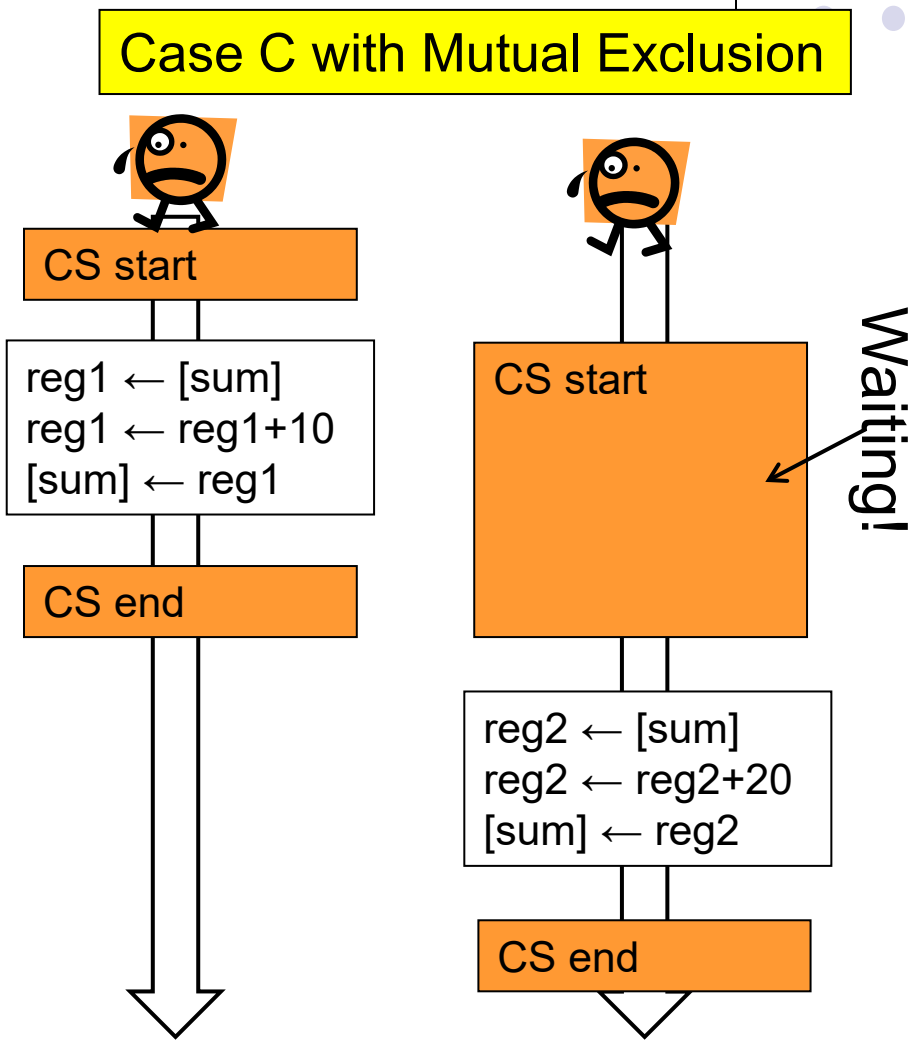
Mutual Exclusion to Avoid Race Condition

Mutual exclusion (mutex):

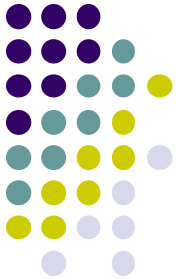
Mechanism to control threads so that only a single thread can enter a "specific region"

- 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;
   }
```

Please try "pi-slow-omp"

cf) ./pi 10000000

But we see pi-slow-omp is very slow ☹

Amdahl's Law

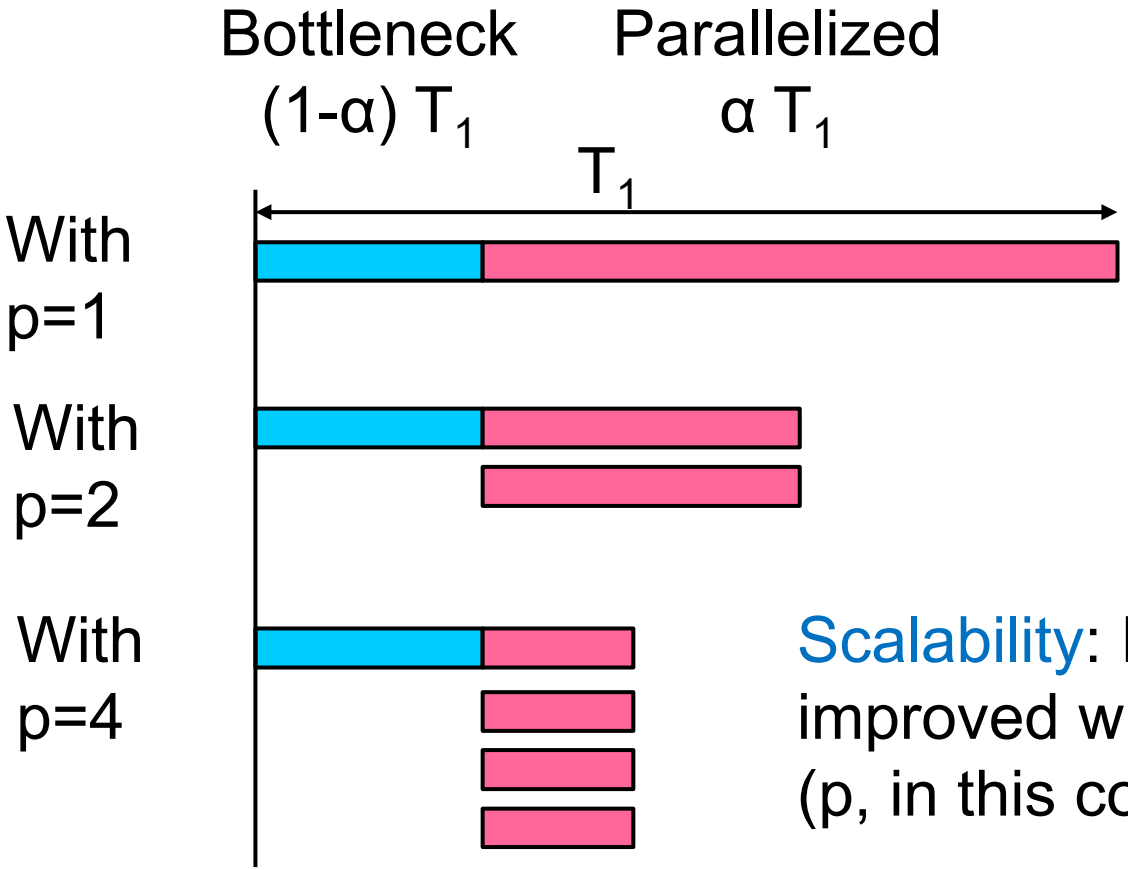
- We consider an algorithm. Then we let
  - T_1 : execution time with 1 processor core
  - α: ratio of computation that can be parallelized
  - 1-α : ratio that CANNOT be parallelized (bottleneck)

⇒ 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

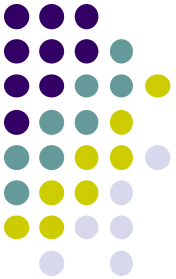


Scalability: How performance is improved with larger resources (p, in this context)

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

Slower! ☹

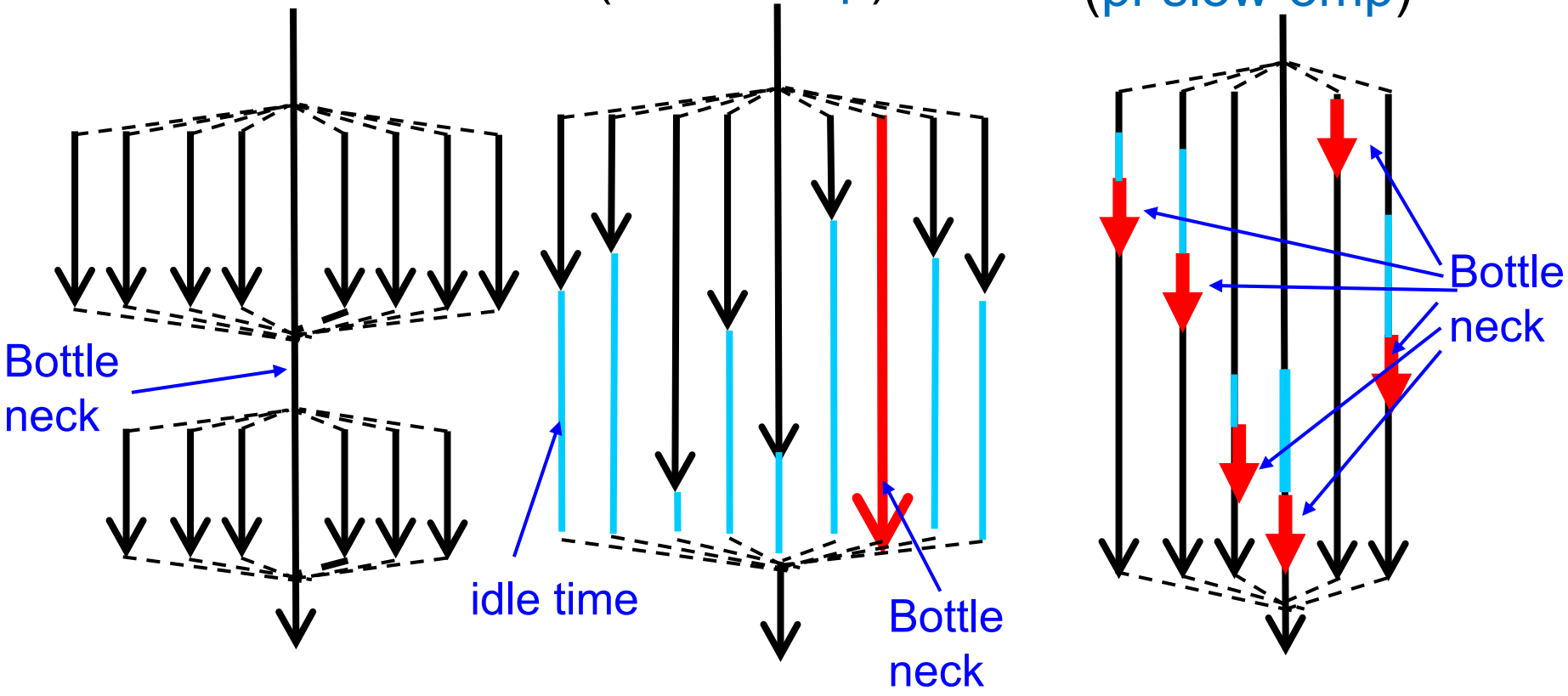
Reducing bottleneck is even more important (than Amdahl's law tells)

Various Bottlenecks

Bottleneck by sequential part

Bottleneck by load imbalance (lumm-omp)

Bottleneck by critical sections (pi-slow-omp)



Moreover, There are architectural bottlenecks

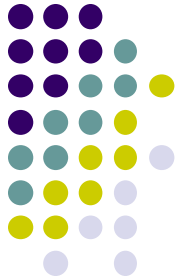
Reducing Bottlenecks

- 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

:

Cases of “pi” Sample



- “pi-slow-omp” is slow, since each thread enters a critical section too frequently

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

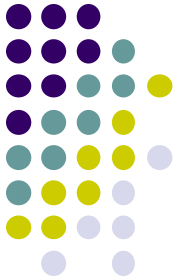
Step 1: Each thread accumulates values into private “local_sum”

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

→ pi-fast-omp is fast and scalable ☺

→ But program is more complex ☹

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

→ pi-omp is fast, like pi-fast-omp ☺

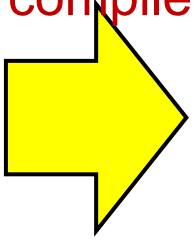
→ Also, programming is easier than pi-fast-omp ☺

Why is Reduction Fast?

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

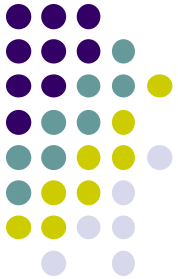
```
    double sum = 0.0;

#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]—[O3], and submit a report

Due date: May 9 (Thu)

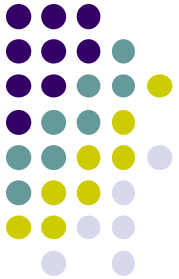
[O1] Parallelize “diffusion” sample program by OpenMP.
(/gs/bs/tga-ppcomp/24/diffusion/ on TSUBAME)

[O2] Parallelize “sort” sample program by OpenMP.
(/gs/bs/tga-ppcomp/24/sort/ on TSUBAME)

[O3] (Freestyle) Parallelize *any* program by OpenMP.

For more detail, please see ppcomp-1-1 slides

Next Class:



- OpenMP (4)
  - “task parallelism” for programs with irregular structures
  - sort: Quick sort sample
    - Related to assignment [O2]