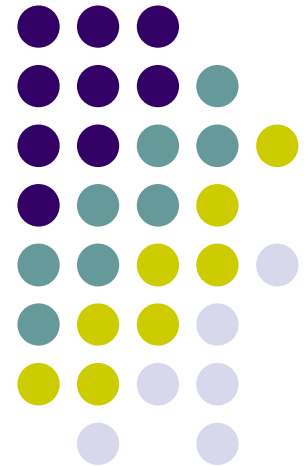


Practical Parallel Computing (実践的並列コンピューティング)

Part1: OpenMP (4)
Apr 28, 2022

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 (4/4)**
- Part 2: **GPU** programming
 - OpenACC and CUDA
 - 4 classes
- Part 3: **MPI** for distributed memory programming
 - 3 classes

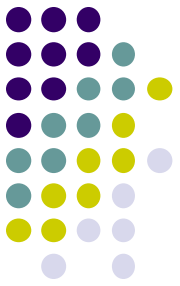


Today's Topic

- 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/hs1/tga-ppcomp/22/lumm/>

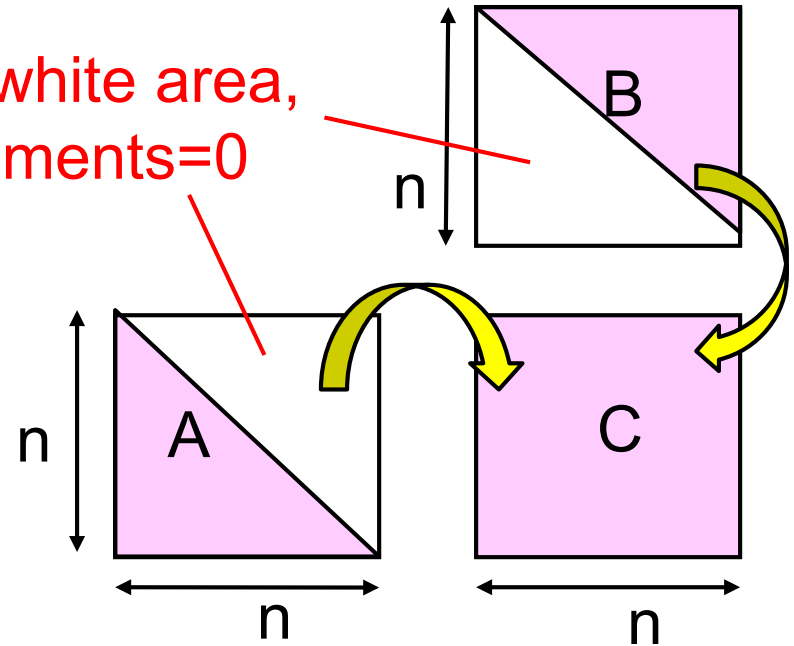


A: a (n×n) matrix
B: a (n×n) matrix
C: a (n×n) matrix

Square matrices

$C \leftarrow A B$

- 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”

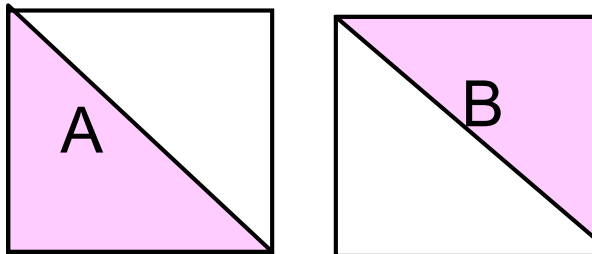


- Matrix multiply

```
for (j = 0; j < n; j++) {  
  for (l = 0; l < k; l++) {  
    for (i = 0; i < m; i++) {  
       $C_{i,j} += A_{i,l} * B_{l,j};$   
    } } }
```

⇒ $2n^3$ computation

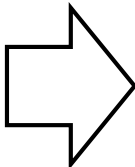
If we know $A_{i,l} = 0$ or $B_{l,j} = 0$, we can skip computation





Computation in “lumm”

- LU Matrix multiply

```
for (j = 0; j < n; j++) {  
  for (l = 0; l <= j; l++) {  
    for (i = l; i < m; i++) {  
      Ci,j += Ai,l * Bl,j;  
    } } }   $(2/3)n^3$  computation
```

Comparing time between
“mm 2000 2000 2000” and “lumm 2000”

	1thread
mm	5.20 (sec)
lumm	1.90 (sec)
mm / lumm	2.74

→ Shorter time in lumm 😊



“lumm-omp”: OpenMP version

Available at </gs/hs1/tga-ppcomp/22/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 * Bi,j;
        } }
    }
```

	1thread	2threads	4threads	7threads
mm	5.20 (sec)	2.62	1.31	0.775
lumm	1.90 (sec)	1.22	0.652	0.378
mm / lumm	2.74	2.15	2.01	2.05

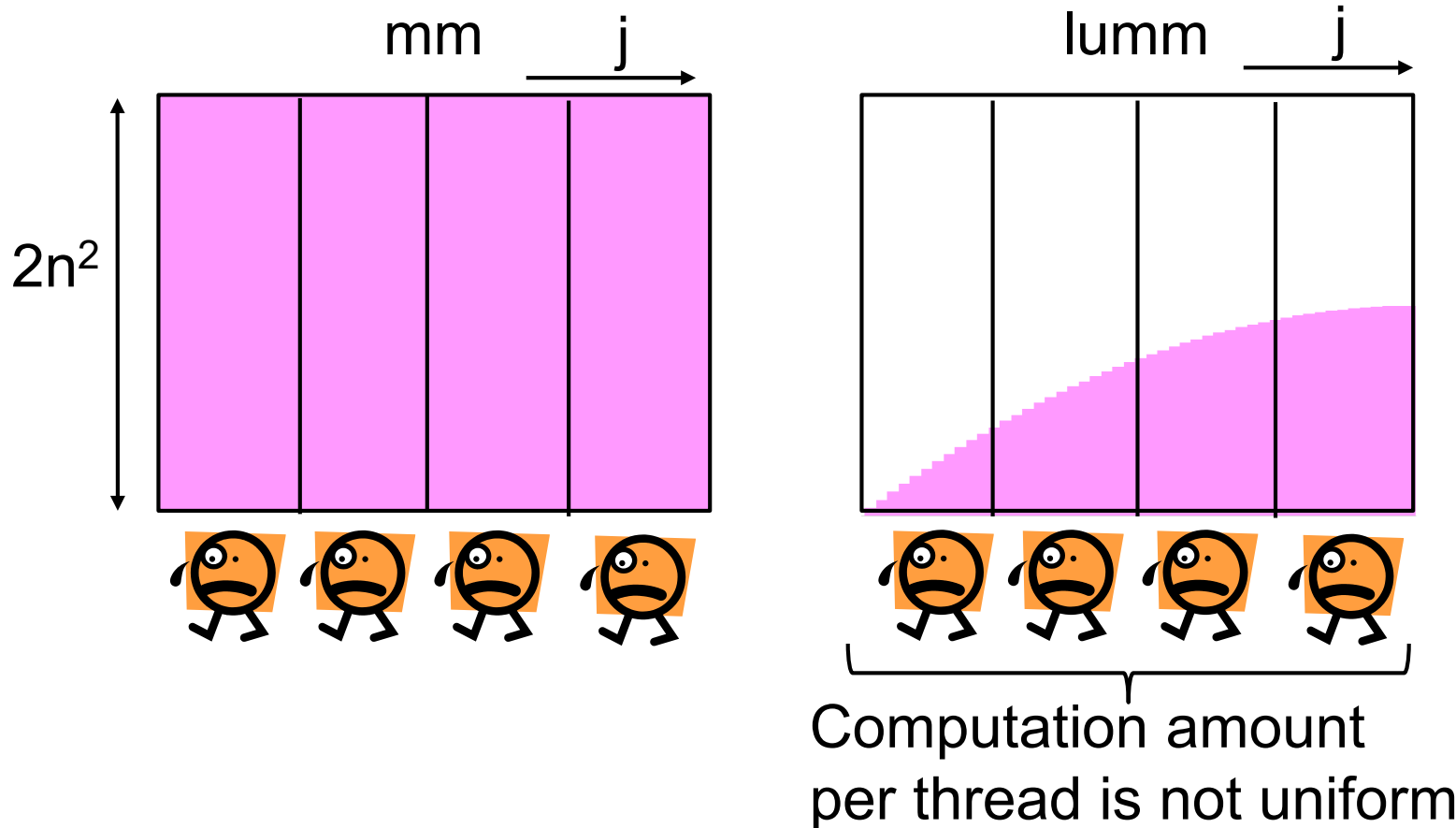


lumm is faster, but the ratio gets worse. Why?



Effects of Load Imbalance

- In lumm, computation amount for each j is not uniform



Towards “Fast” Parallel Software



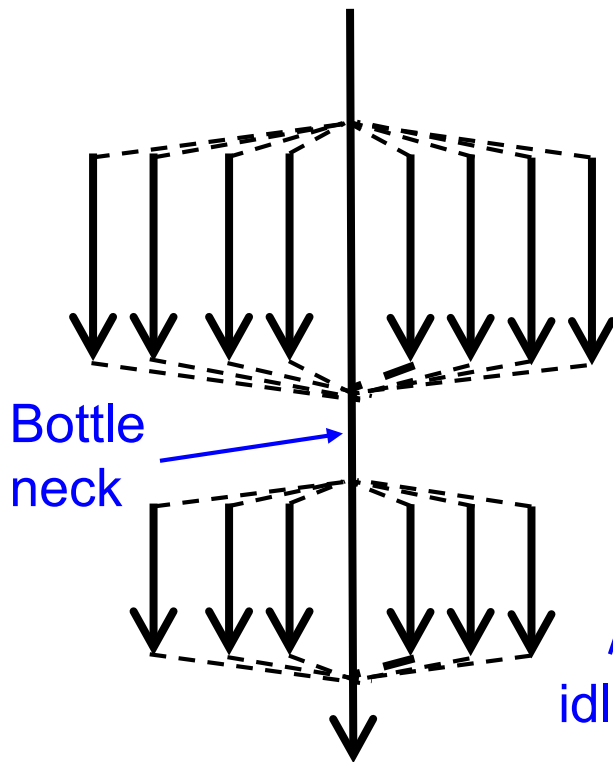
- If the entire algorithm is divided into independent computations (such as mm example), the story is easy
 - But generally, most algorithms include both
 - Computations that converge on specific threads
 - Computations that can be parallelized
- ⇒ The later part raises problems called “**bottleneck**”



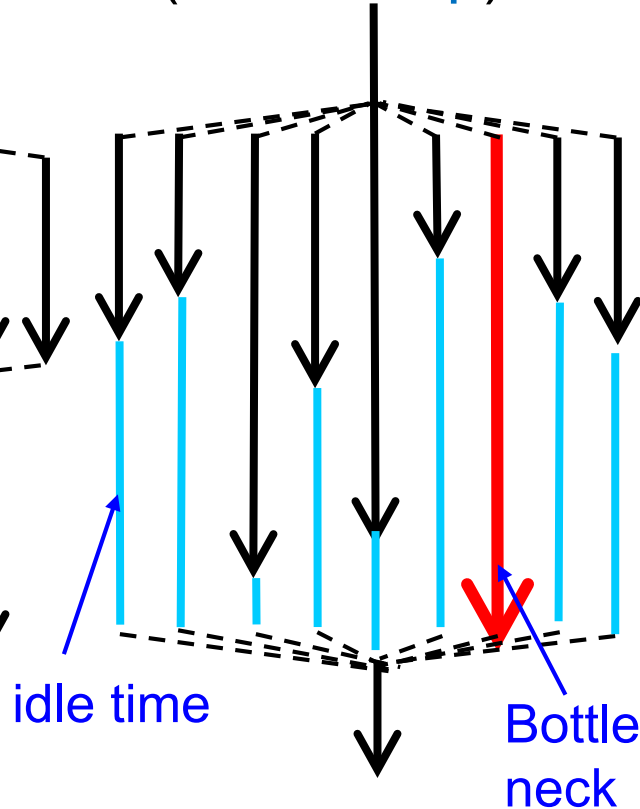


Various Bottlenecks

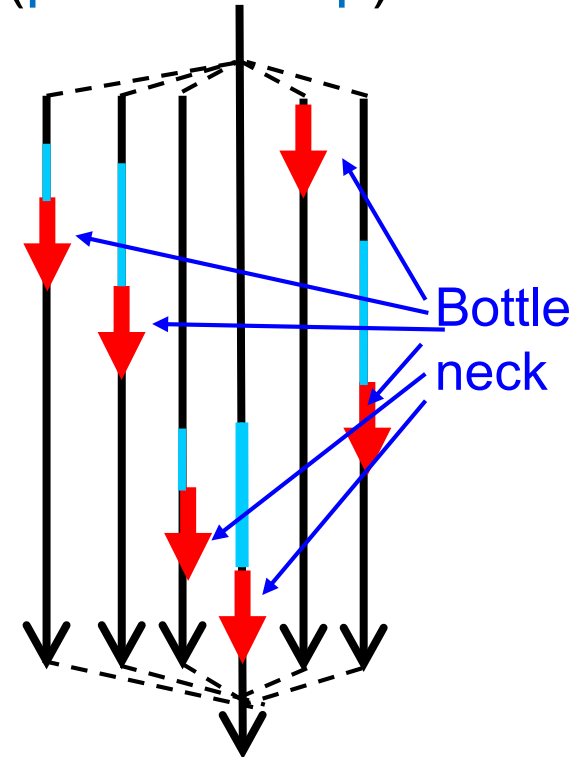
Bottleneck by
sequential part



Bottleneck by
load imbalance
(lumm-omp)



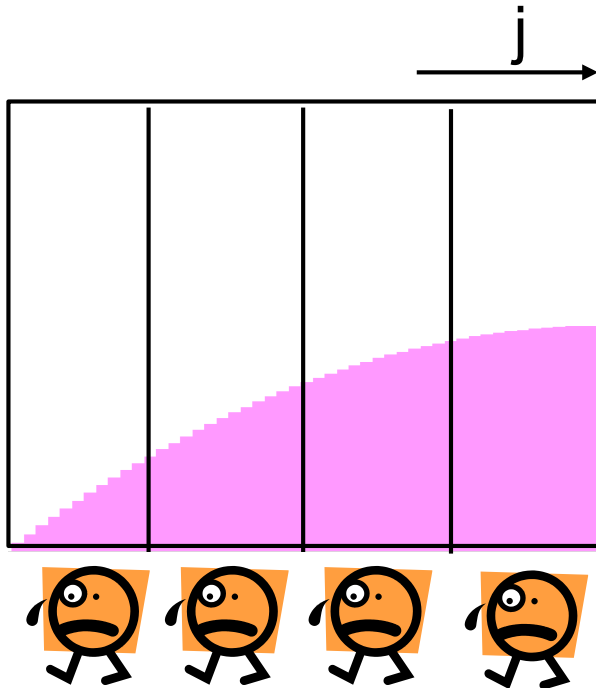
Bottleneck by
critical sections
(pi-slow-omp)



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

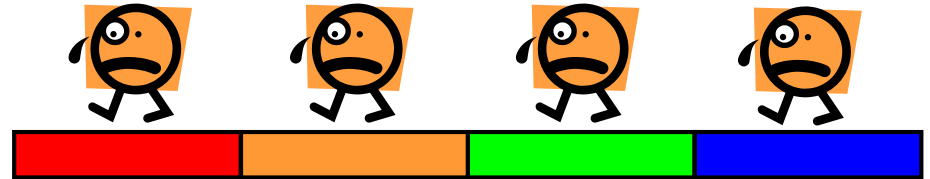


- OpenMP provides several scheduling methods (mapping between iteration and threads)

#pragma omp for `schedule(...)`

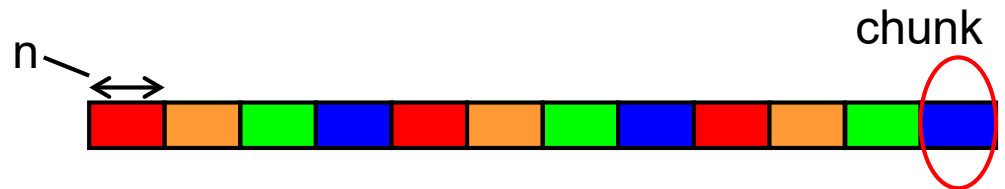
`schedule(static)`

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



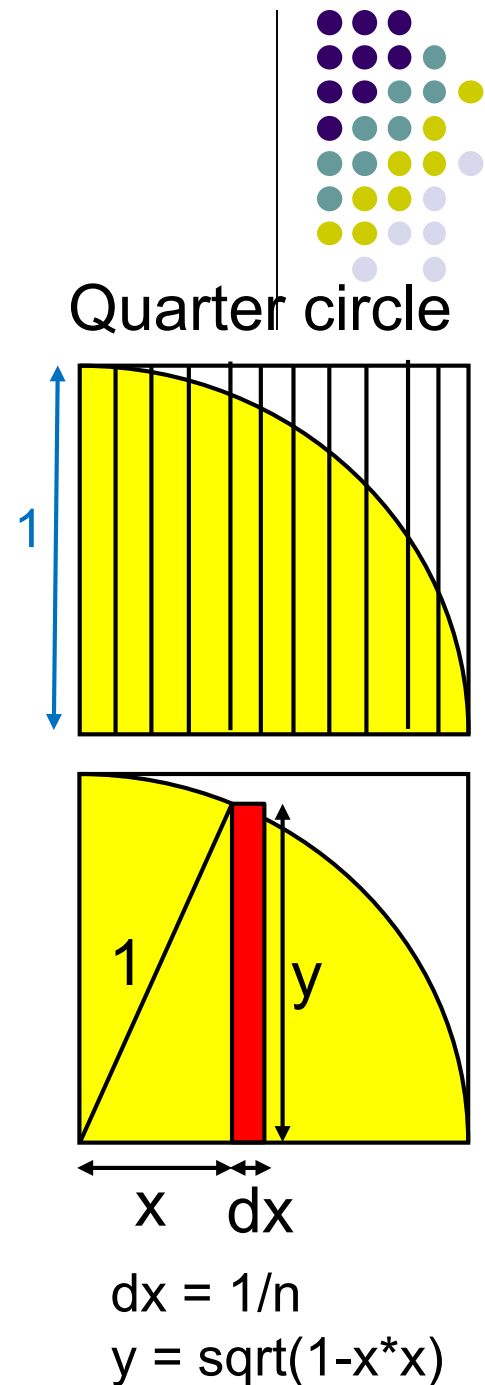
[Q] In lumm, `#pragma omp for schedule(static,1)` works good. Why?

“pi” sample

Compute an approximation of $\pi = 3.14159\dots$
(circumference/diameter)

- Available at </gs/hs1/tga-ppcomp/22/pi/>
- Method
 - $SUM \leftarrow$ Approximation of the yellow area
 - $\pi \leftarrow 4 \times SUM$
- Execution: `./pi [n]`
 - n: Number of division
 - Cf) `./pi 100000000`
- Compute complexity: $O(n)$

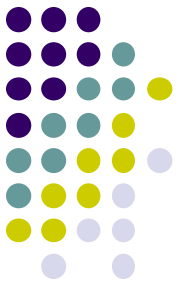
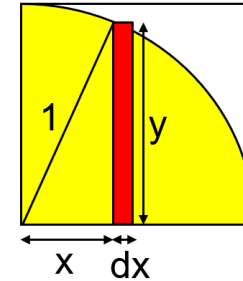
*Note: This program is only for a simple sample.
 π is usually computed by different algorithms.*



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)

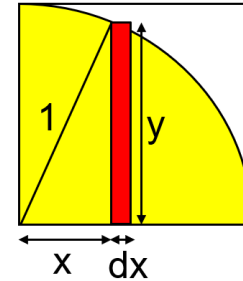
```
double pi(int n) {  
    int i;  
    double sum = 0.0;  
    double dx = 1.0 / (double)n;
```

```
#pragma omp parallel  
#pragma omp for
```

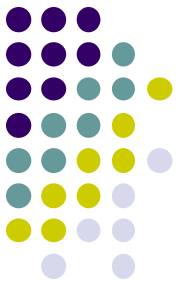
} ok???

```
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; }
```



- Can we use `#pragma omp for`?
- 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;
```

$R(C1) = \{\text{sum}, dx\}$, $W(C1) = \{\text{sum}\}$

$R(C2) = \{\text{sum}, dx\}$, $W(C2) = \{\text{sum}\}$

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

- $W(C1) \cap W(C2) \neq \emptyset \rightarrow$ **Dependent!**

\rightarrow Do we have to abandon parallel execution?





Some 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 at </gs/hs1/tga-ppcomp/22/>

What's Wrong in pi-bad-omp? (1)

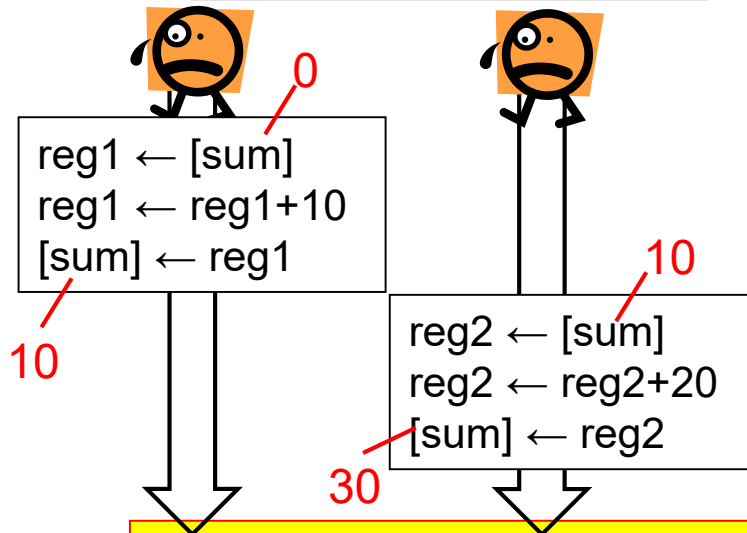


- Now we simply consider `C1: sum += 10;` & `C2: sum += 20;`
- We assume “`sum = 0`” initially
- [Q] Does execution order of C1 & C2 affect the results?
 - Note: “`sum += 10`” is compiled into machine codes like

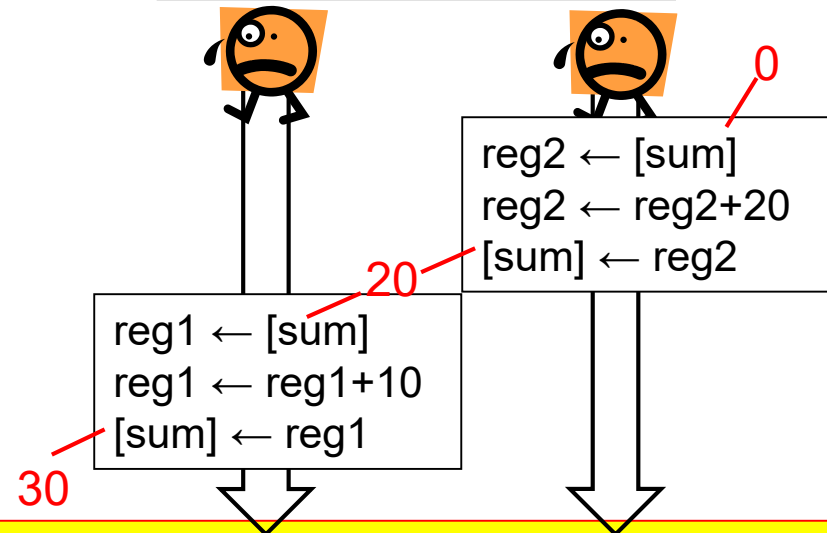
```
reg1 ← [sum]
reg1 ← reg1+10
[sum] ← reg1
```

※ `reg1, reg2...` are registers,
which are thread private

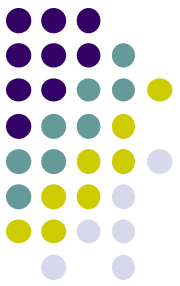
Case A: C1 then C2



Case B: C2 then C1

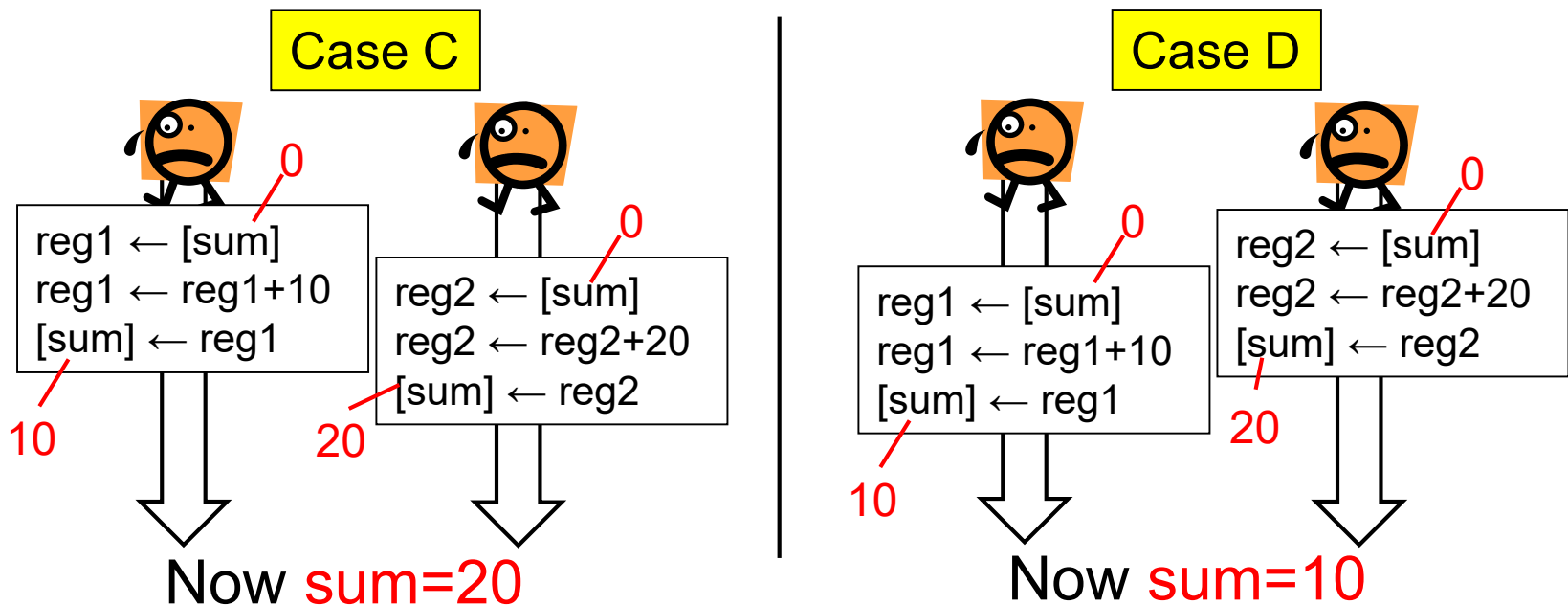


The results are same: sum=30. Ok to parallelize???



What's Wrong in pi-bad-omp? (2)

- **No!!!** The results can be **different** if C1 & C2 are executed (almost) simultaneously

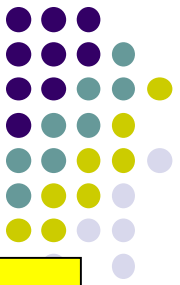


The expected result is 30, but we may get bad results

Such a bad situation is called “**Race Condition**”

➔ Please try “**pi-bad-omp**”

Mutual Exclusion to Avoid Race Condition



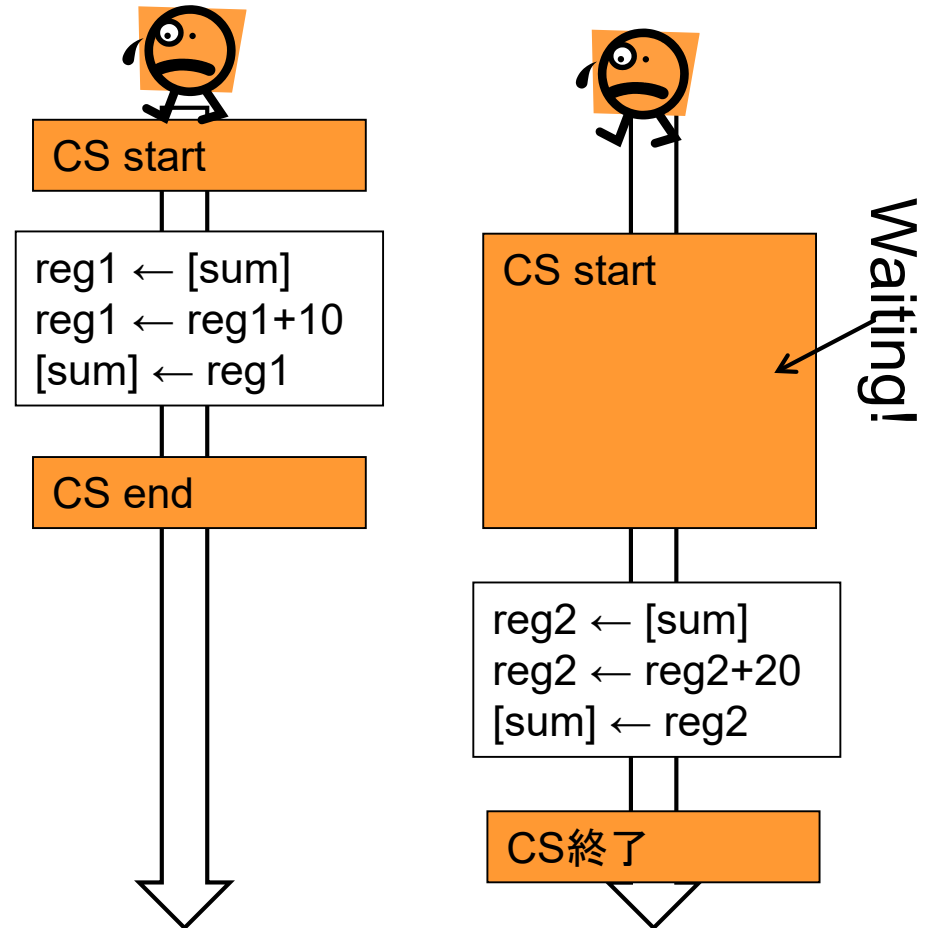
Mutual exclusion (mutex):

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

Case C with Mutual Exclusion



sum=30



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 100000000

- Computes integral by multiple threads
- The algorithm uses “*sum* += ...”
- The answer is 3.1415...

But we see **pi-slow-omp** is very slow ☹



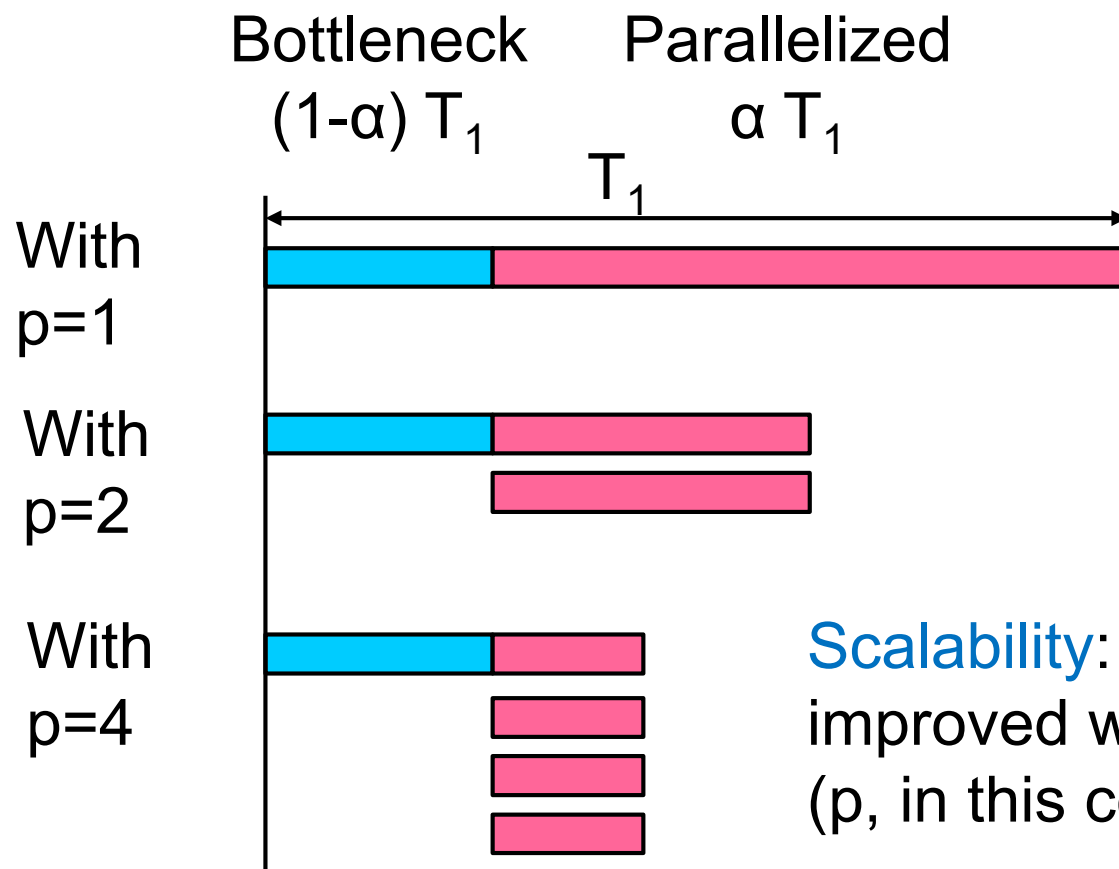
Amdahl's Law

- We consider an algorithm. Then we let
 - T_1 : execution time with 1 processor core
 - α : ratio of computation that can be parallelized
 - $1-\alpha$: 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 we want scalability with $p \sim 10$, α should be >0.9
- if we want scalability with $p \sim 100$, α should be >0.99



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 100000000
```

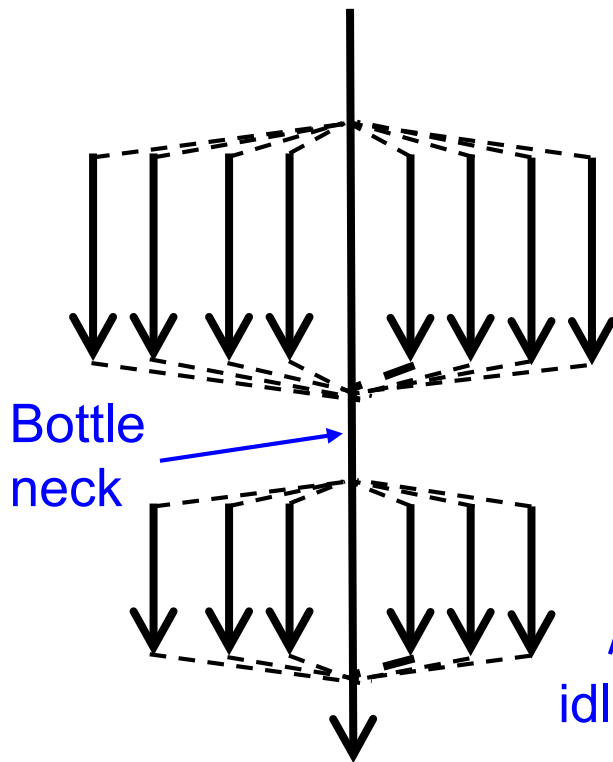
p	pi-omp pi-fast-omp	pi-good-omp	 Slower! 😞
1	0.80 (sec)	1.8 (sec)	
2	0.40 (sec)	9.4 (sec)	
5	0.16 (sec)	10.9~13.0 (sec)	
10	0.08 (sec)	13~16 (sec)	

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

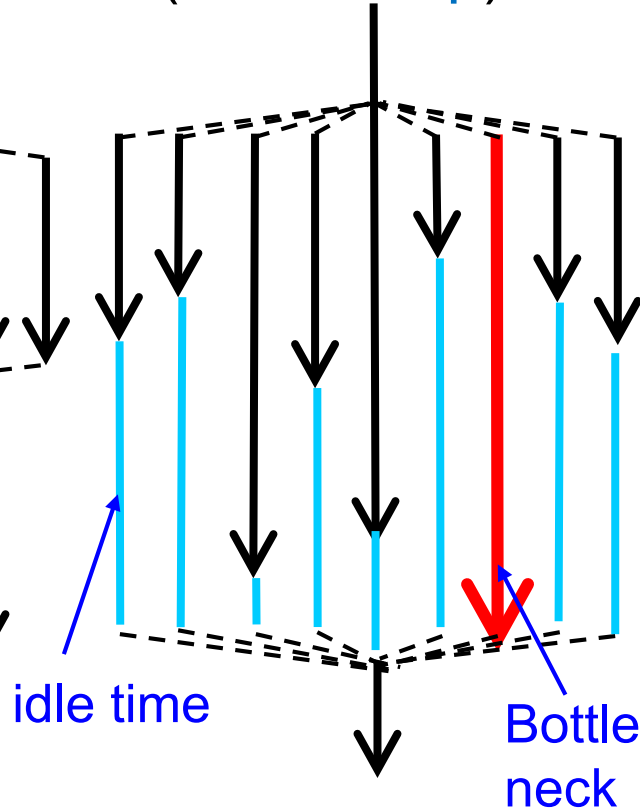
Various Bottlenecks



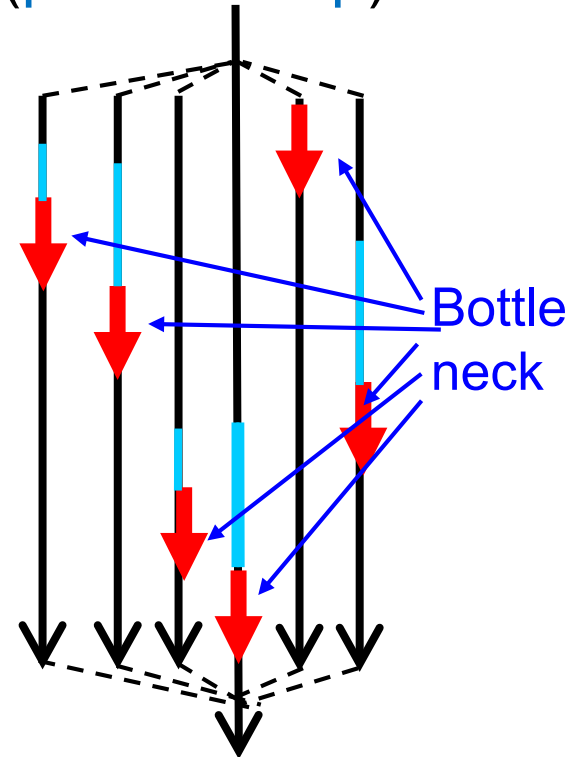
Bottleneck by
sequential part



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 😊

Why is pi-omp (the first omp version) also fast?
“omp for **reduction**(...)” is internally compiled to a similar code as above

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 😊

What We Have Learned in OpenMP Part



- OpenMP: A programming tool for parallel computation by using multiple processor cores
 - Shared memory parallel model
 - `#pragma omp parallel` → Parallel region
 - `#pragma omp for` → Parallelize for-loops
 - `#pragma omp task` → Task parallelism
- We can use multiple processor cores, but only in a single node

Assignments in OpenMP Part (Abstract)



Choose one of [O1]—[O3], and submit a report
Due date: May 12 (Thu)

[O1] Parallelize “diffusion” sample program by OpenMP.

(</gs/hs1/tga-ppcomp/22/diffusion/> on TSUBAME)

[O2] Parallelize “sort” sample program by OpenMP.

(</gs/hs1/tga-ppcomp/22/sort/> on TSUBAME)

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

For more detail, please see OpenMP (1) slides



Next Class:

- Part 2: GPU Programming (1)
 - What GPU programming is
 - Introduction to OpenACC