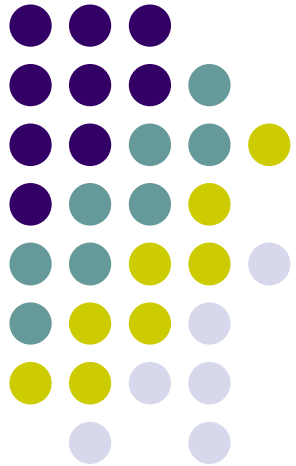
2019 Practical Parallel Computing (実践的並列コンピューティング) No. 6

Shared Memory Parallel Programming with OpenMP (4)

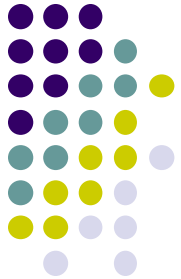


Toshio Endo

School of Computing & GSIC

endo@is.titech.ac.jp

Considerations in Parallel Programming



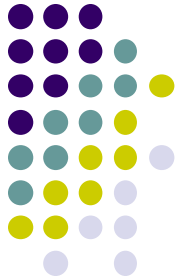
Step1: How we can make "correct" parallel software

- Is dependency preserved?
- No race condition?

Step2: How we can make "fast" parallel software

- Is bottleneck small?
- Are tasks well balanced between threads?

Towards "Correct" Parallel Software



- We have learned several OpenMP syntaxes to make computations parallel
  - #pragma omp parallel
  - #pragma omp for
  - #pragma omp task

- But it is programmer's responsibility to check whether the parallelization is correct or not

Can We Do in Parallel?

[Q1] Is it ok to execute C1 and C2 in parallel?

C1: y = 10; C2: z = 20;

※y, z are shared variables

→Yes ☺ Execution order of C1&C2 does not affect results

C1: y = 10; then → C2: z = 20;

C2: z = 20; then → C1: y = 10;

y = 10 and z = 20
Same results

[Q2] Is it ok to execute C3 and C4 in parallel?

C3: x = 10; C4: x = 20;

※x is a shared variable

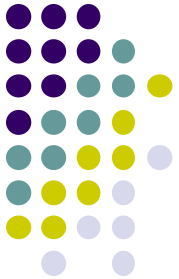
→No! ☹ If execution order is changed, we see different results

C3: x = 10; then → C4: x = 20;

C4: x = 20; then → C3: x = 10;

x = 20? x = 10?
Different results!

Dependency between Computations



We define following sets for computation C

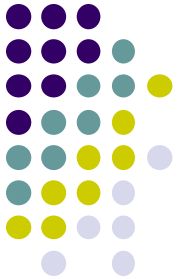
- Read set R(C): the set of variables read by C
- Write set W(C): the set of variables written by C
  - Ex) C: x = y+z ➔ R(C) = {y, z}, W(C) = {x}

We define dependency between C1 and C2

- If (W(C1) ∩ R(C2) ≠ ∅), C1 and C2 are dependent (write vs read)
- If (R(C1) ∩ W(C2) ≠ ∅), C1 and C2 are dependent (read vs write)
- If (W(C1) ∩ W(C2) ≠ ∅), C1 and C2 are dependent (write vs write)
- Otherwise, C1 and C2 are independent
  - ※ read vs read cases are independent

If C1 and C2 are independent, parallelization of C1 and C2 is safe ☺

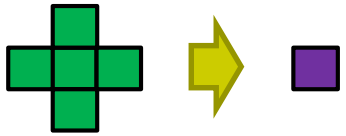
Dependency and Parallelism in Stencil Computations (1)



Consider 1D stencil computation:

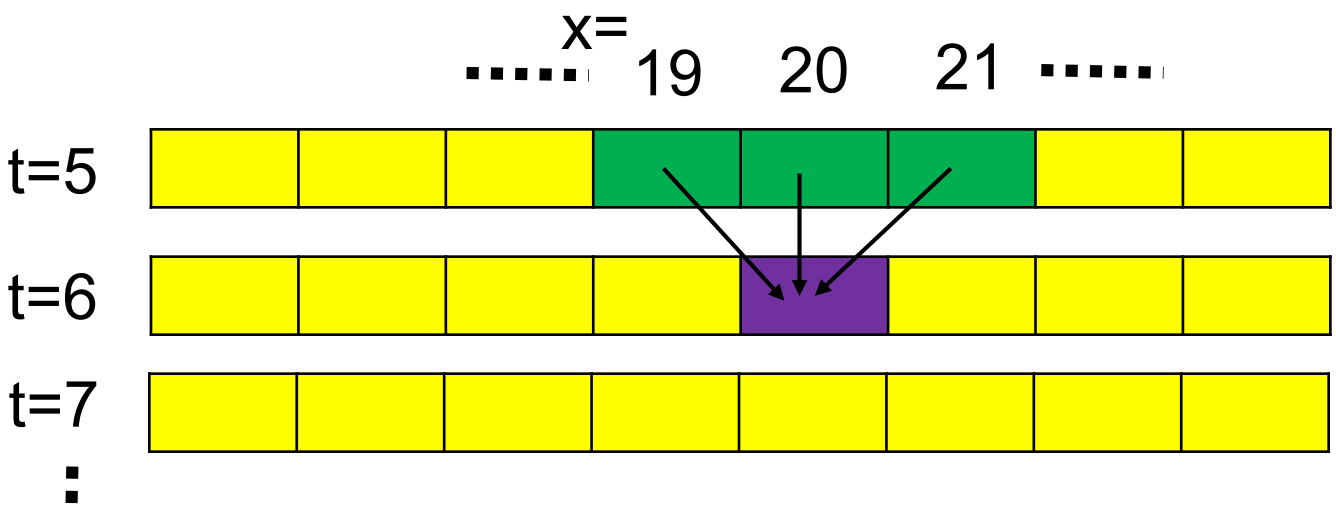
```
for (t = 1; t < NT; t++)
    for (x = 1; x < NX-1; x++)
        f_{t,x} = (f_{t-1,x-1} + f_{t-1,x} + f_{t-1,x+1}) / 3.0   /*  C_{t,x} */
```

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



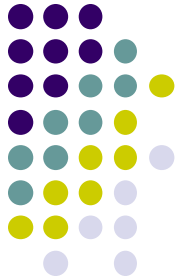
We let $C_{t,x}$ be computation of a single point $f_{t,x}$

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



※ This figure omits double buffering technique

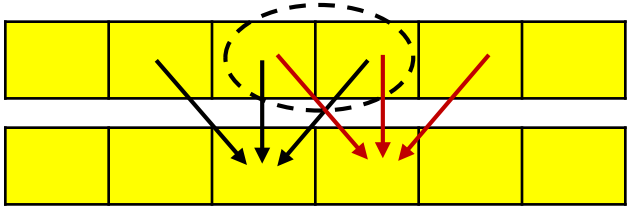
Dependency and Parallelism in Stencil Computations (2)



- Can we compute $f_{6,20}$ and $f_{6,21}$ in parallel? *(t is same, x is different)*
  - $R(C_{6,20})=\{f_{5,19}, f_{5,20}, f_{5,21}\}$, $W(C_{6,20})=\{f_{6,20}\}$
  - $R(C_{6,21})=\{f_{5,20}, f_{5,21}, f_{5,22}\}$, $W(C_{6,21})=\{f_{6,21}\}$

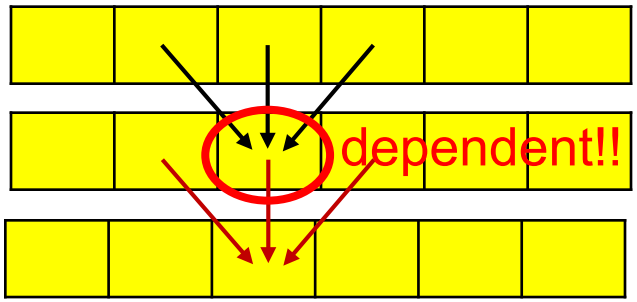
  → They are independent ☺ (for all pairs of x)

Read vs. Read is Ok



- Can we compute $f_{6,20}$ and $f_{7,20}$ in parallel? *(t is different)*
  - $R(C_{6,20})=\{f_{5,19}, f_{5,20}, f_{5,21}\}$, $W(C_{6,20})=\{f_{6,20}\}$
  - $R(C_{7,20})=\{f_{6,19}, f_{6,20}, f_{6,21}\}$, $W(C_{7,20})=\{f_{7,20}\}$

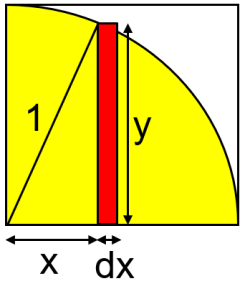
  → They are dependent ☹



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

Partially Dependent Case

- Can we execute C1 and C2 in parallel?
  - Here, *sum* is a shared variable
  - Similar pattern appears in “pi” sample



C1

```
    :
[calculate ans1]
    :
sum += ans1;
```

these parts are independent

dependent

C2

```
    :
[calculate ans2]
    :
sum += ans2;
```

- C1 and C2 are dependent ☹
  - since both W(C1) and W(C2) includes *sum*

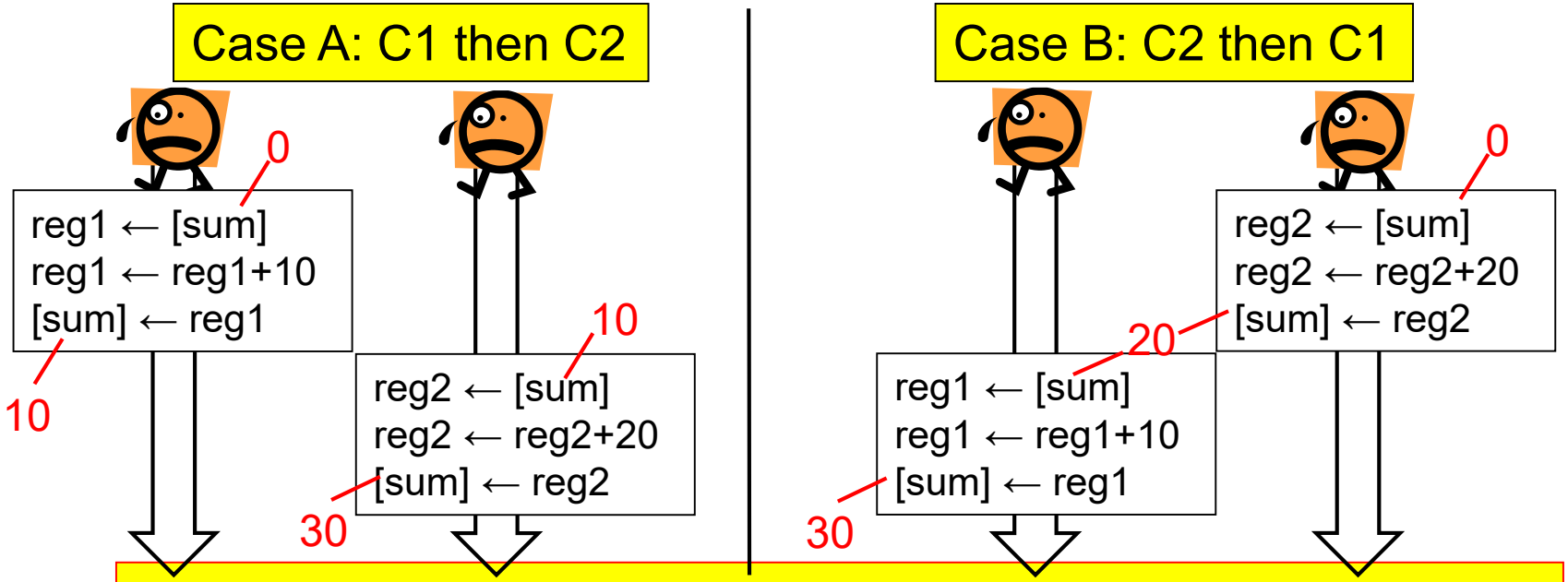
➔ Do we have to abandon parallel execution?

What's Wrong if Parallelized? (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

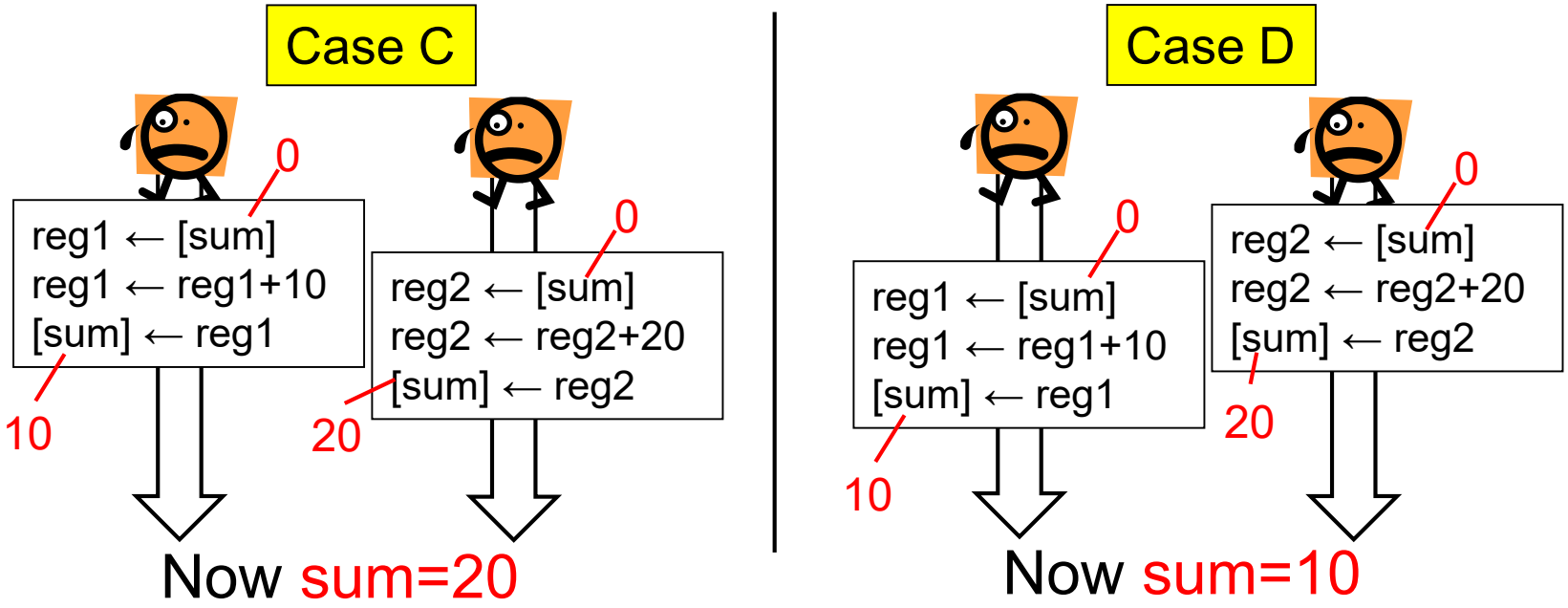
```
reg1 ← [sum]
reg1 ← reg1+10
[sum] ← reg1
```

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



What's Wrong if Parallelized? (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"

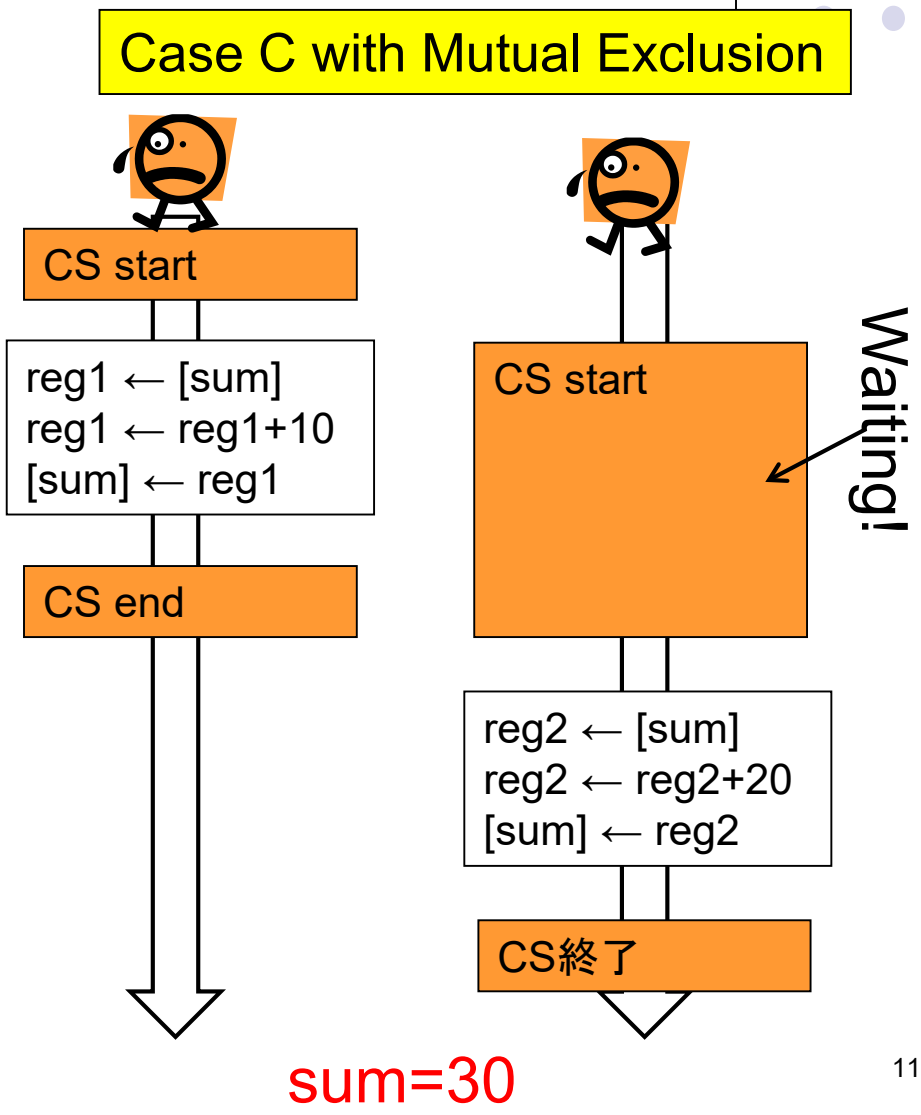
Mutual Exclusion to Avoid Race Condition

Mutual exclusion (mutex):

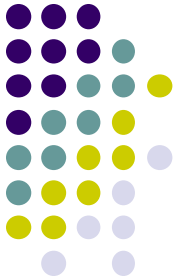
Mechanism to control threads so that only a single thread can enter a “specific region”

- 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;
    }
```

An example available at ~endo-t-ac/ppcomp/19/pi-good-omp/

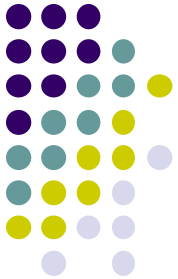
cf) ./pi 100000000

- Computes integral by multiple threads
- The algorithm uses "*sum* += …"
- The answer is 3.1415…

Compare several versions. What are differences?

- pi-bad-omp: Bad answer ☹ due to race condition
- pi-good-omp: Correct answer ☺, but slow ☹ (why?)
- pi-omp / pi-fast-omp: Correct ☺ and fast ☺

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 can be parallelized
  - Computations that cannot (or hardly) be parallelized

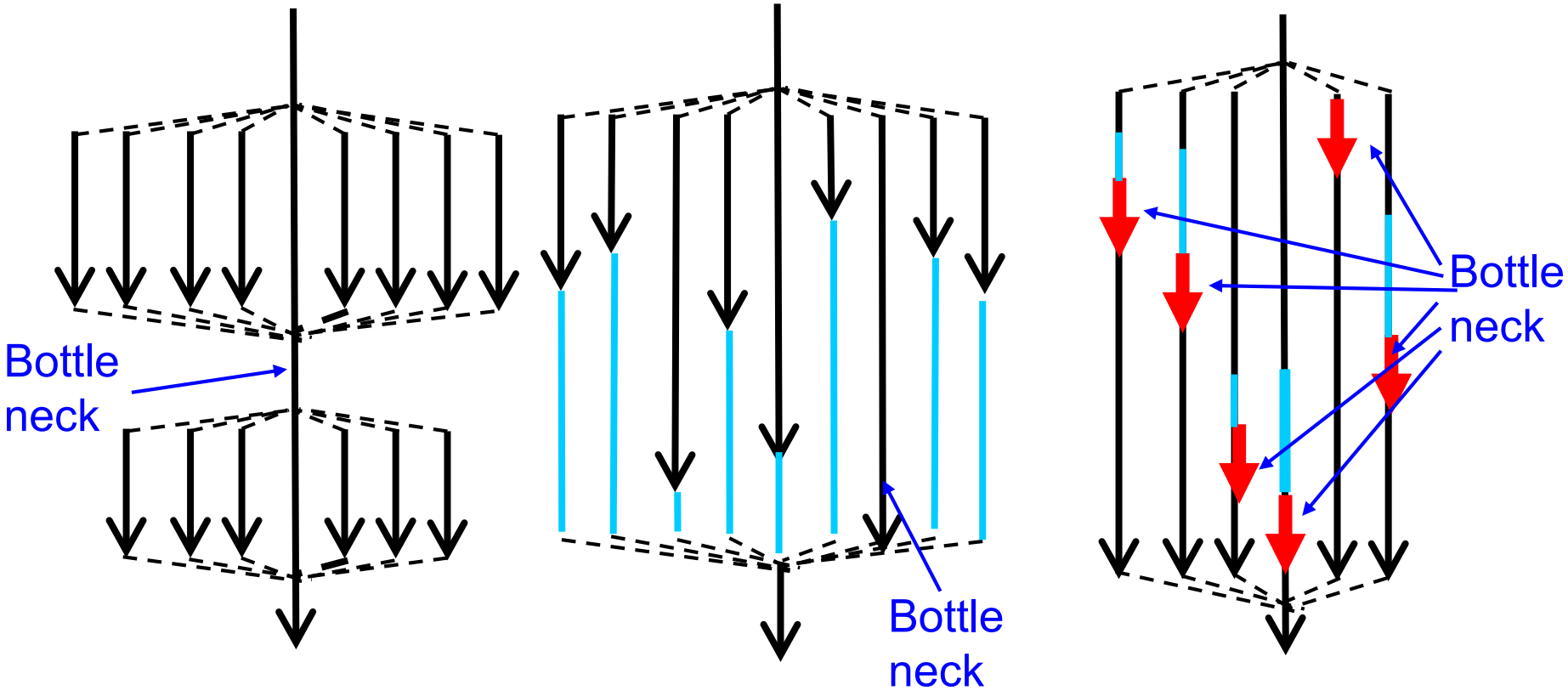
⇒ The later part raises problems called "bottleneck"

Various Bottlenecks

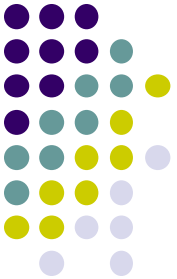
Bottleneck by sequential part

Bottleneck by load imbalance

Bottleneck by critical sections



Moreover, There are architectural bottlenecks



Amdahl's Law

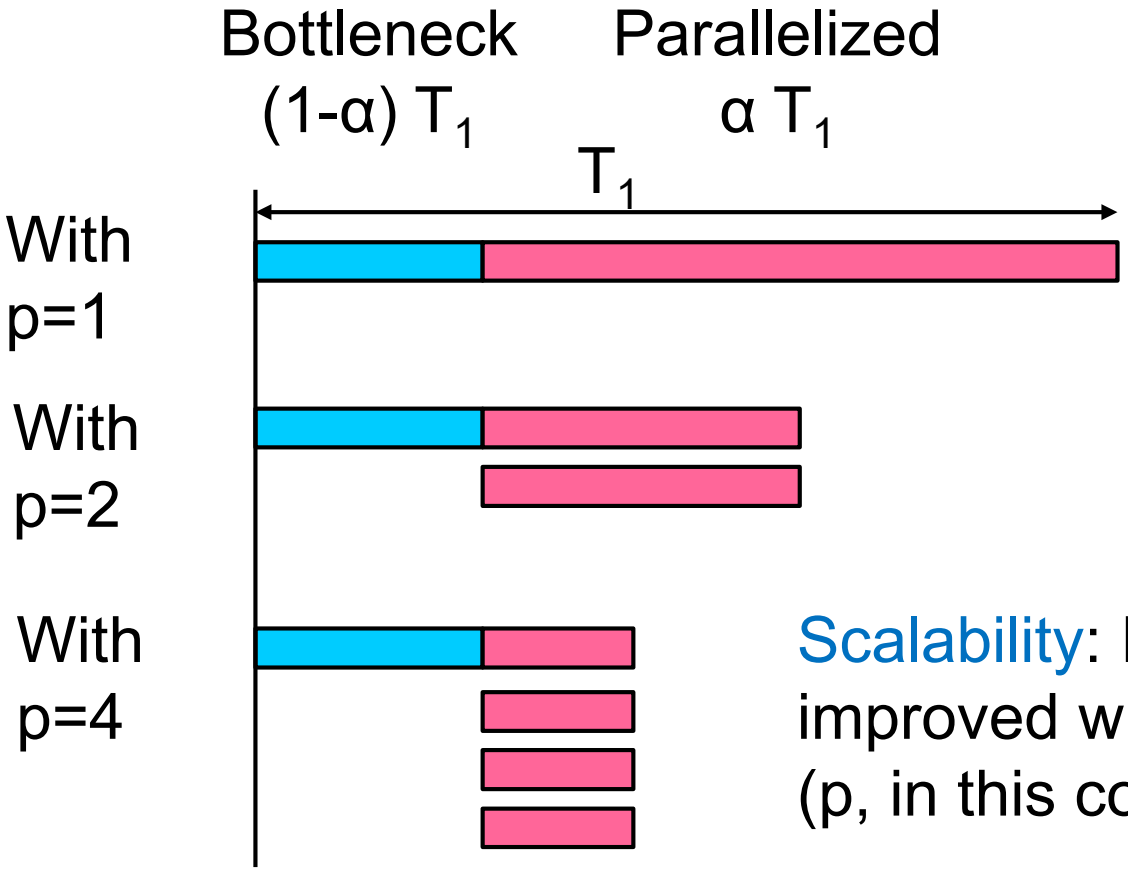
- We consider an algorithm. Then we let
  - T_1 : execution time with 1 processor core
  - α: ratio of computation that can be parallelized
  - 1-α : ratio that CANNOT be parallelized (bottleneck)

⇒ 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

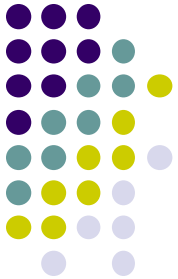


Scalability: How performance is improved with larger resources (p, in this context)

Amdahl's law tells us

- if we want scalability with p∼10, α should be >0.9
- if we want scalability with p∼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-good-omp

export OMP_NUM_THREADS= [p]

./pi 100000000

p	pi-omp pi-fast-omp	pi-good-omp
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)

Slower! ☹

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

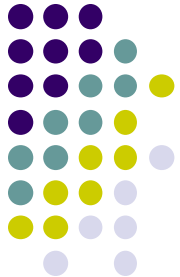
Reducing Bottlenecks

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

- Some directions
  - 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-good-omp” is slow, since each thread enters a critical section too frequently

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

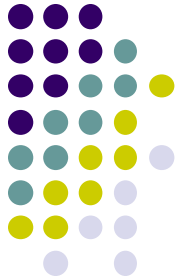
Step 1: Each thread accumulates values into private “local_sum”

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

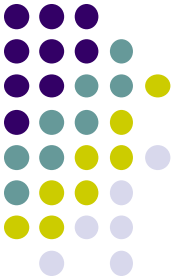
What We Have Learned in OpenMP Part



- 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 node
- In MPI part, we will go over the wall of a node

Assignments in OpenMP Part (Abstract)



Choose *one of* [O1]—[O3], and submit a report

Due date: May 9 (Thursday)

[O1] Parallelize “diffusion” sample program by OpenMP.
(~endo-t-ac/ppcomp/19/diffusion/ on TSUBAME)

[O2] Parallelize “sort” sample program by OpenMP.
(~endo-t-ac/ppcomp/19/sort/ on TSUBAME)

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

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

Next Class:

- Part 2: Distributed Memory Parallel Programming with MPI (1)

Information

Lecture

- Slides are uploaded in OCW
  - www.ocw.titech.ac.jp → search “2019 practical parallel computing”
- Assignments information/submission site are in OCW-i
  - Login portal.titech.ac.jp → OCW/OCW-i
- Inquiry
  - ppcomp@el.gsic.titech.ac.jp
- Sample programs
  - Login TSUBAME, and see ~endo-t-ac/ppcomp/19/ directory

TSUBAME

- Official web including Users guide
  - www.t3.gsic.titech.ac.jp
- Your account information
  - Login portal.titech.ac.jp → TSUBAME portal