

2017

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

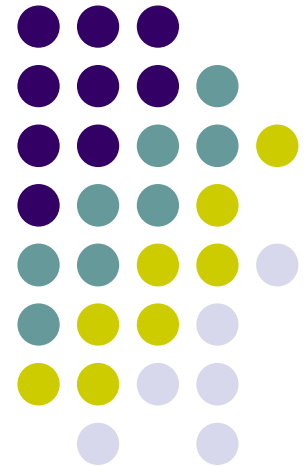
No. 6

Shared Memory Parallel Programming with OpenMP (4)

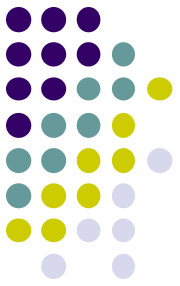
Toshio Endo

School of Computing & GSIC

endo@is.titech.ac.jp



Considerations in Parallel Programming



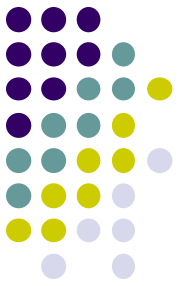
Step1: How we can make “correct” parallel software

- Is dependency preserved?
- No race condition?

Step2: How we can make “fast” parallel software

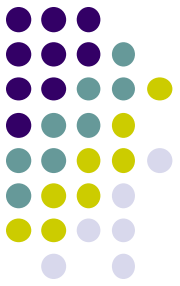
- Is bottleneck small?
- Are tasks well balanced between threads?

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

Dependency between Computations



- If partial computations C1 and C2 are **independent**, we can parallelize them
- If they are **dependent**, we cannot

C1: Read a, b and Write c
C2: Read d, e and Write f
C3: Read c and Write g
C4: Read e and Write h
C5: Read i and Write h

Which computations are independent?

C1&C2 → **independent**

C1&C3 → **dependent**

- c is **written** by C1, **read** by C3(!)

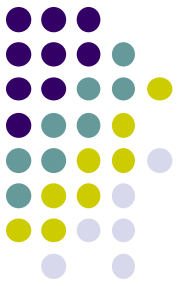
C2&C4 → **independent**

- e is **read** by C2&C4
- Read vs. Read is Ok

C4&C5 → **dependent**

- h is **written** by C4&C5
- Write vs. Write is NG

Dependency and Parallelism in Stencil Computations (1)

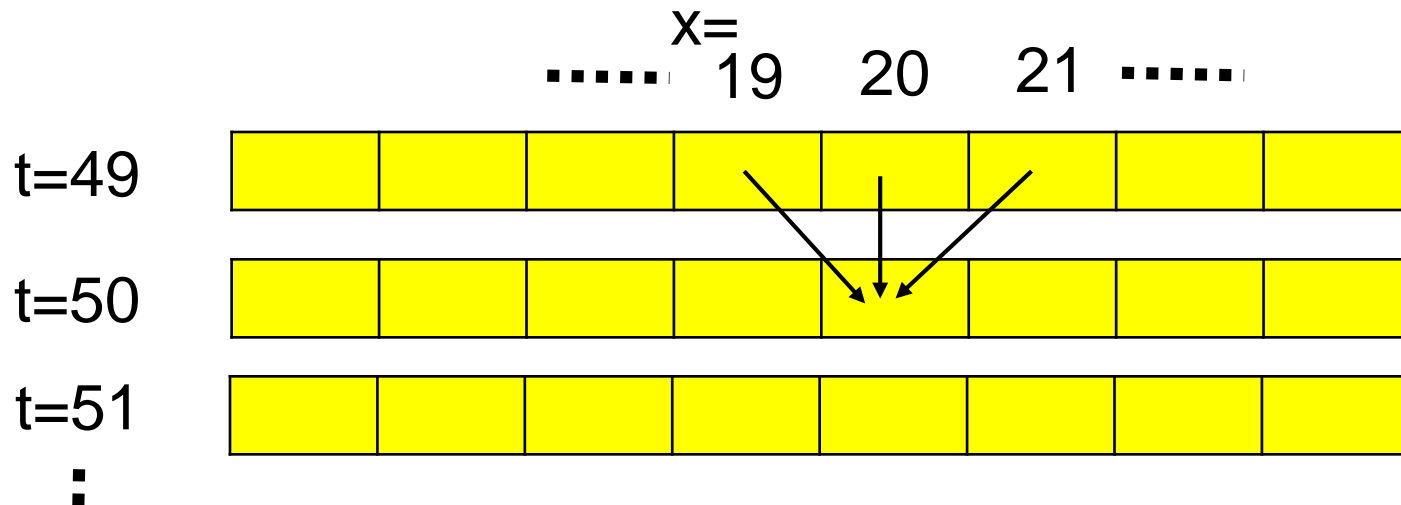


Consider a stencil computation:

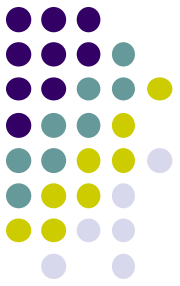
$$f_{t+1,x} = (f_{t,x-1} + f_{t,x} + f_{t,x+1}) / 3.0$$

✂ This is simpler than diffusion sample

- We focus on update of a single point
 - It includes 3 Read and 1 Write

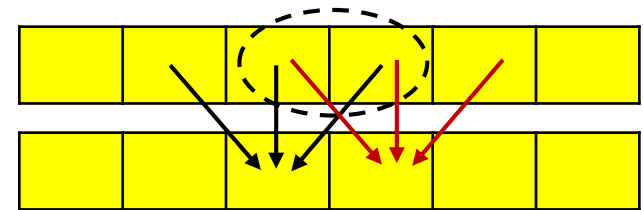


Dependency and Parallelism in Stencil Computations (2)

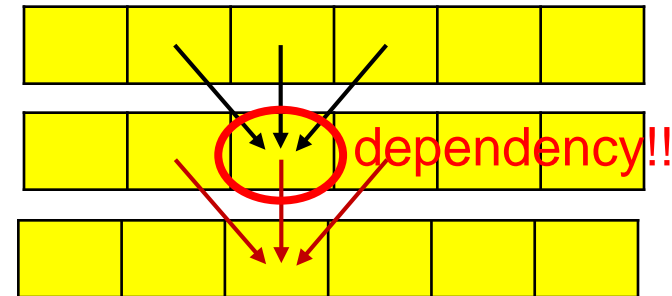


- Can we compute $f_{50,20}$ and $f_{50,21}$ in parallel? (t is same, x is different)
 - $f_{50,20}$: Read $f_{49,19}$, $f_{49,20}$, $f_{49,21}$ and Write $f_{50,20}$
 - $f_{50,21}$: Read $f_{49,20}$, $f_{49,21}$, $f_{49,22}$ and Write $f_{50,21}$
 → They are independent 😊 (for all pairs of x)

Read vs. Read is Ok

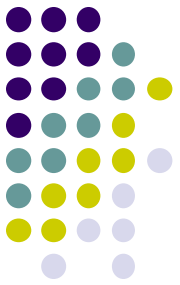


- Can we compute $f_{50,20}$ and $f_{51,20}$ in parallel? (t is different)
 - $f_{50,20}$: Read $f_{49,19}$, $f_{49,20}$, $f_{49,21}$ and Write $f_{50,20}$
 - $f_{51,20}$: Read $f_{50,19}$, $f_{50,20}$, $f_{50,21}$ and Write $f_{51,20}$
 → They are dependent ☹️



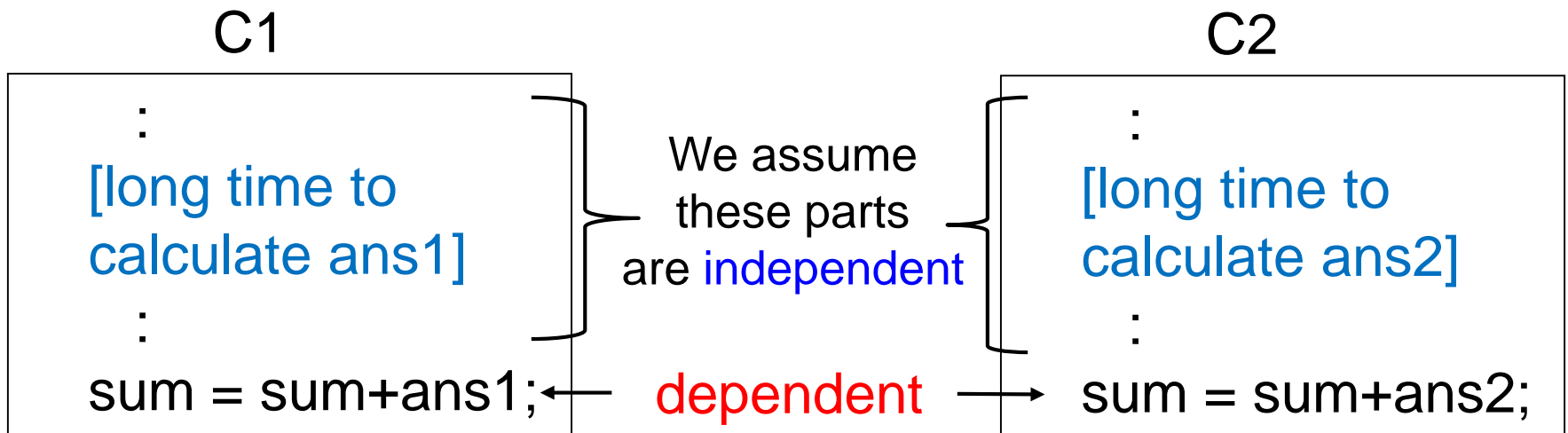
In Assignment [O1]

- it is OK to parallelize x-loop or y-loop
- it is NG to parallelize t-loop



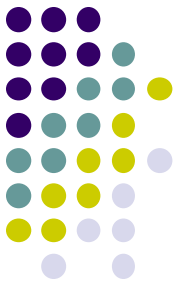
Partially Dependent Case

- Can we execute C1 and C2 in parallel?
 - Here, *sum* is a shared variable



- C1 and C2 are **dependent**, since both **write** *sum*
→ The answer is **no**. But do we have to abandon parallel execution?

What's Wrong if Parallelized? (1)



- What happens if C1, C2 are executed **in sequential**



```
sum = 0;  
C1 (ans1=10)  
C2 (ans2=20)
```

After execution, **sum = 30**

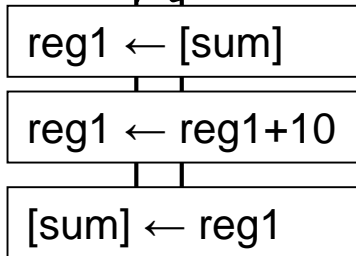
- To discuss parallel execution, let's consider
 - In parallel, execution timing is non-deterministic
 - “sum = sum + 10” is compiled into machine codes like
 - reg1 \leftarrow [sum]
 - reg1 \leftarrow reg1+10
 - [sum] \leftarrow reg1

What's Wrong if Parallelized? (2)



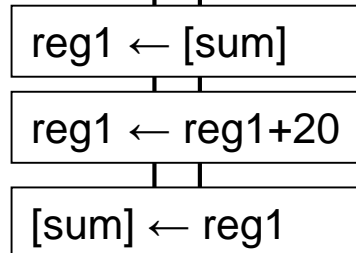
Parallel Execution: Case A

Execution of C1



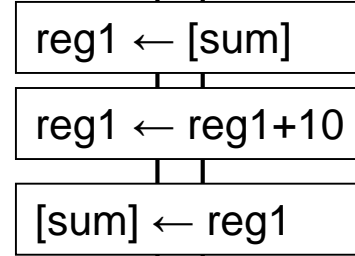
Here, **sum=30**

Execution of C2



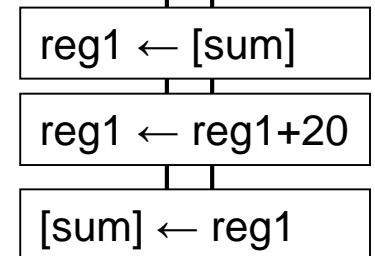
Parallel Execution: Case B

Execution of C1



Here, **sum=20**

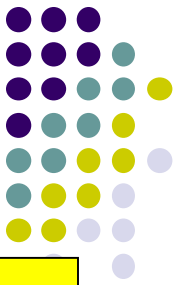
Execution of C2



Different
result!

Such a bad situation is called "**Race Condition**"

Mutual Exclusion to Avoid Race Condition



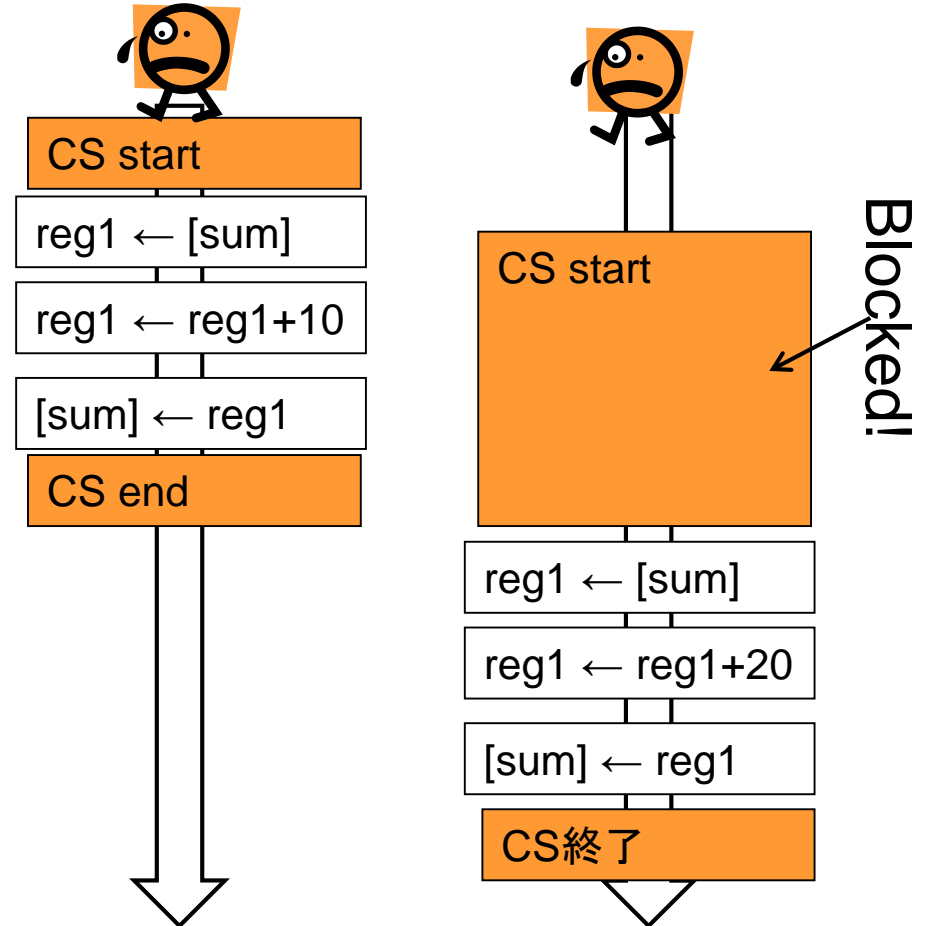
Mutual exclusion (mutex):

Control threads so that only a single thread can enter a “specific region”

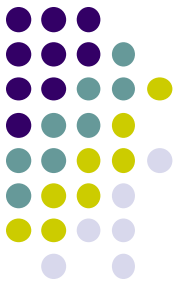
- The region is called **critical section**

⇒ With mutual exclusion, race condition is avoided

Case B with Mutual Exclusion



sum=30



Mutual Exclusion in OpenMP

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

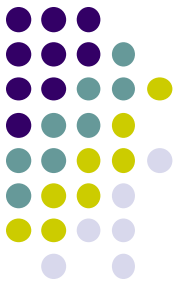
```
int sum = 0;
#pragma omp parallel
{
    [ do something ]
    #pragma omp critical
    {
        sum = sum + myans;
    }
}
```

Examples available at
[~endo-t-ac/ppcomp/17/
count-omp/](http://endo-t-ac/ppcomp/17/count-omp/)

cf) `./count-XXX [n]`
→ Each thread adds 1 to a shared counter for n times
→ Correct answer would be $n \times \text{OMP_NUM_THREADS}$

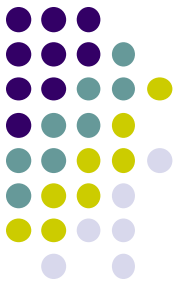
- **count-bad**: Wrong version
- **count-good**: Correct, but slow version with **mutex**
- **count-fast**: Correct and fast version

Towards “Fast” Parallel Software



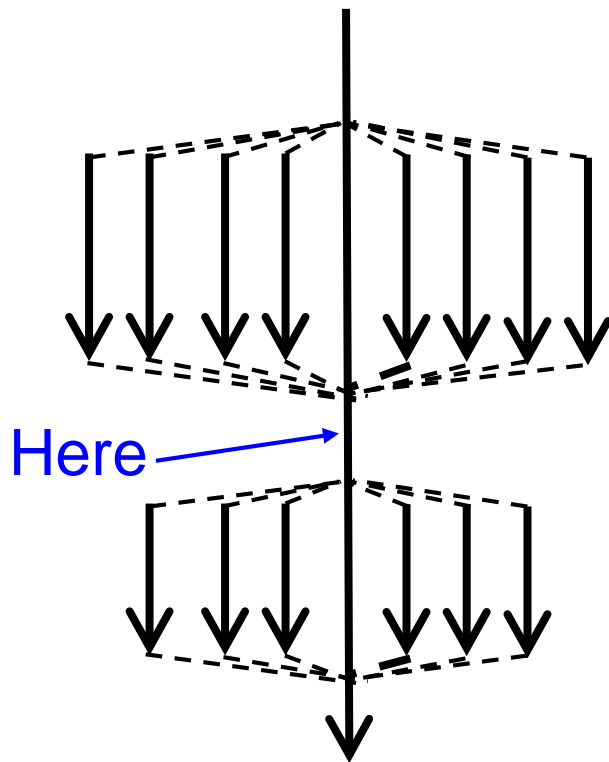
- Most algorithms include both
 - Computations that can be parallelized
 - Computations that cannot (or hardly) be parallelized
- ⇒ The later raises problems called “**bottleneck**”



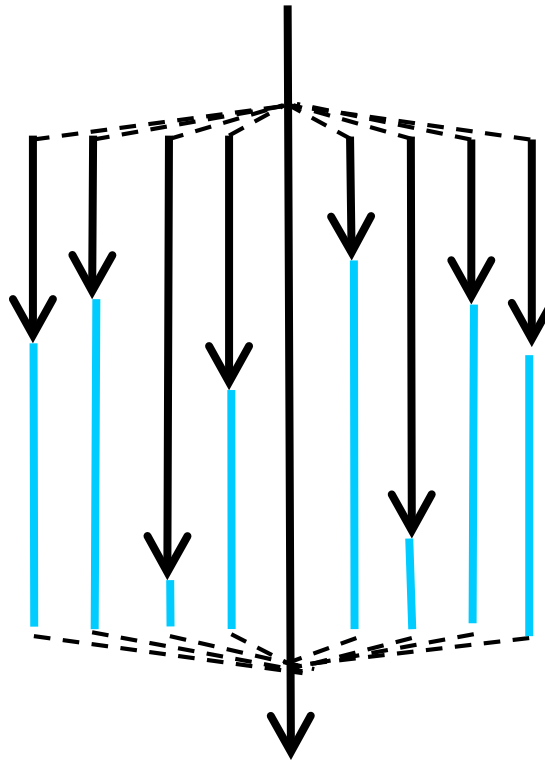


Various Bottlenecks

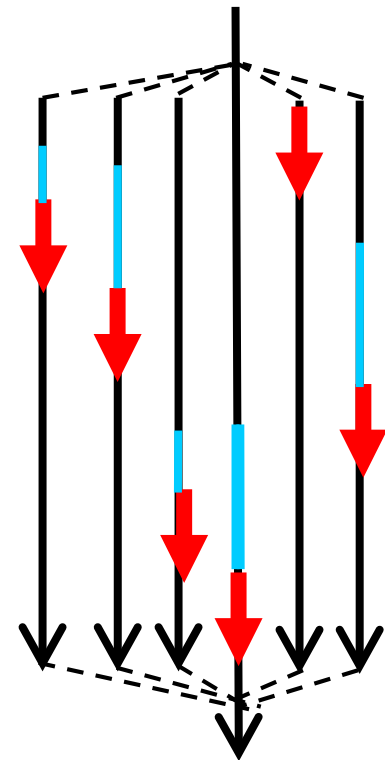
Bottleneck by
sequential part



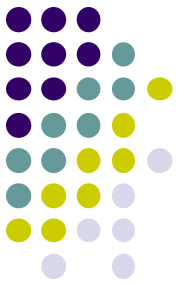
Bottleneck by
load imbalance



Bottleneck by
critical sections



There are more, such as architectural bottlenecks



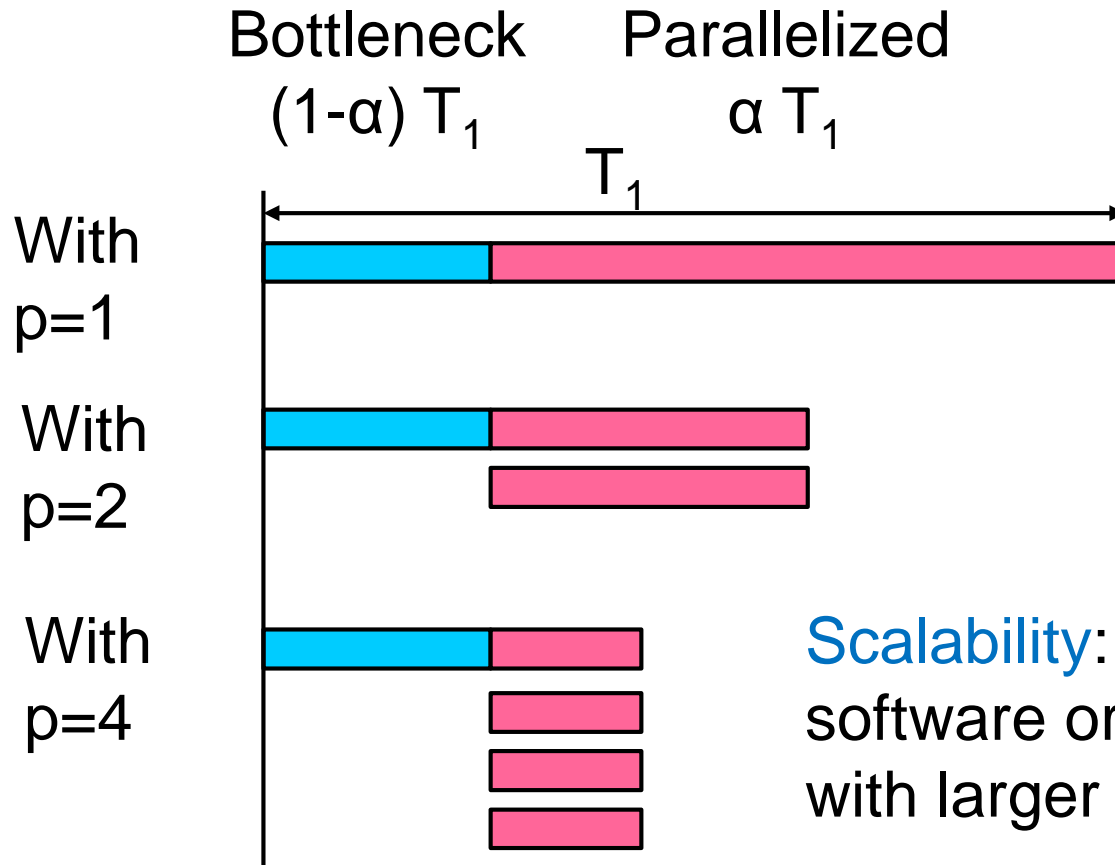
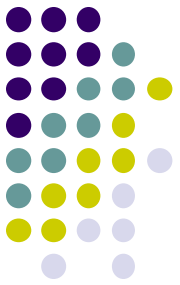
Amdahl's Law

- In an algorithm, we let
 - T_1 : execution time with 1 processor core
 - α be ratio of computation that can be parallelized
 - $1-\alpha$ be ratio that cannot be parallelized (bottleneck)
- ⇒ Estimated execution time with p processor cores is $T_p = ((1 - \alpha) + \alpha / p) T_1$
- *smaller is better*

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: Performance of software or algorithm is improved with larger resources (p)

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 Worse Than The Theory



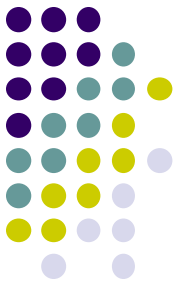
- According to Amdahl's law, T_p is monotonically decreasing
→ Larger p is not harmful?

count-good sample in [~endo-t-ac/ppcomp/17/count-omp/](#)
(TSUBAME2 node)

- $p=1$: 1 thread \times 10M times \rightarrow 0.16sec
- $p=2$: 2 threads \times 5M times \rightarrow 0.37~0.52sec
- $p=5$: 5 threads \times 2M times \rightarrow 1.2~1.7sec
- $p=10$: 10 threads \times 1M times \rightarrow 1.2~2.1sec

↓ Slower

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



Reducing Bottlenecks

- Methods for reducing bottlenecks depend on algorithms!
 - We need to puzzle over
 - There are algorithms that 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





Case of “count-omp” Sample

- “count-good” version has too frequent access to a shared variables
- count-fast version introduces private variables

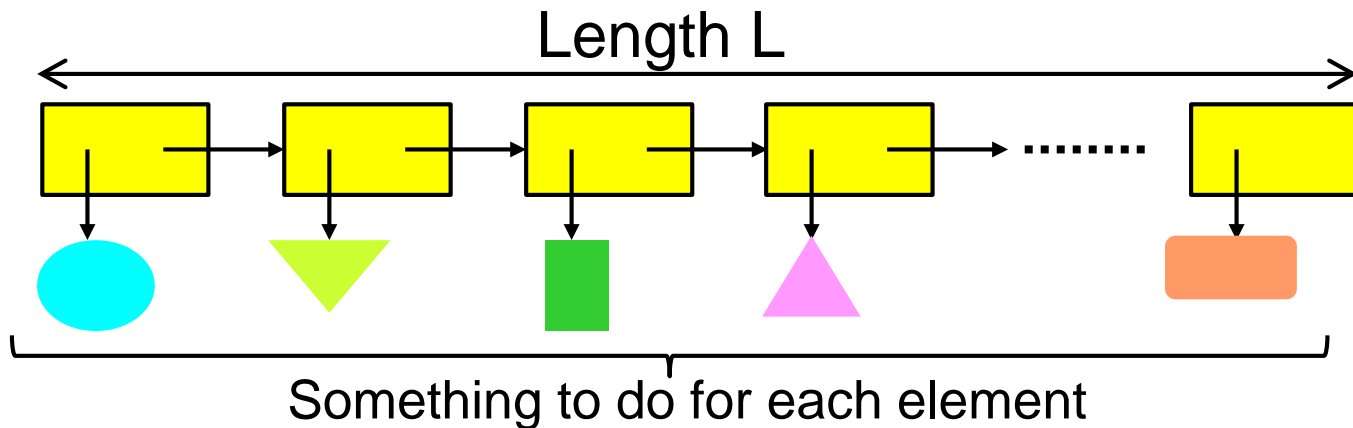
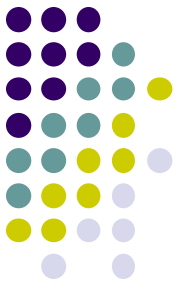
Step 1: Each thread accumulates values into **private** “local_s”

Step 2: Then each thread does “s += local_s” in a critical section **once per thread**

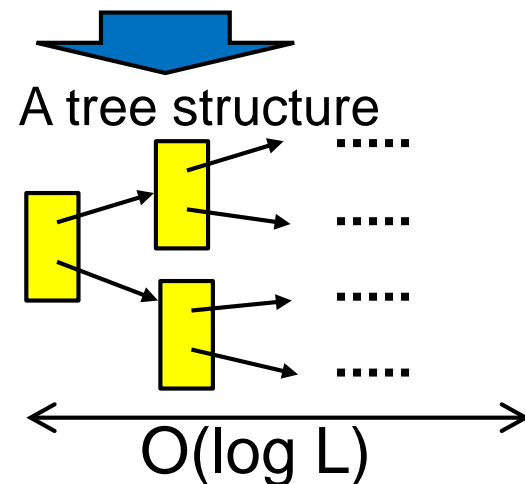
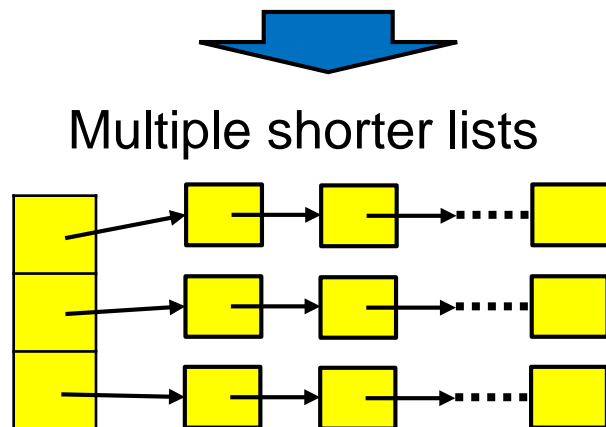
- With this version, 10threads × 1M times add → 4msec

⌘ “omp for **reduction(...)**” is internally compiled to use a similar method

Case of Tracing a List



- “Critical path” has L length
 - If L is large and each task is small, tracing the list itself will become a bottleneck (ex. calling “omp task” for L times)

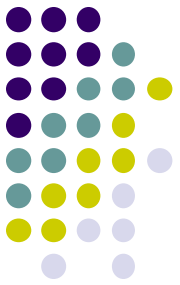


What We Have Learned in OpenMP Part

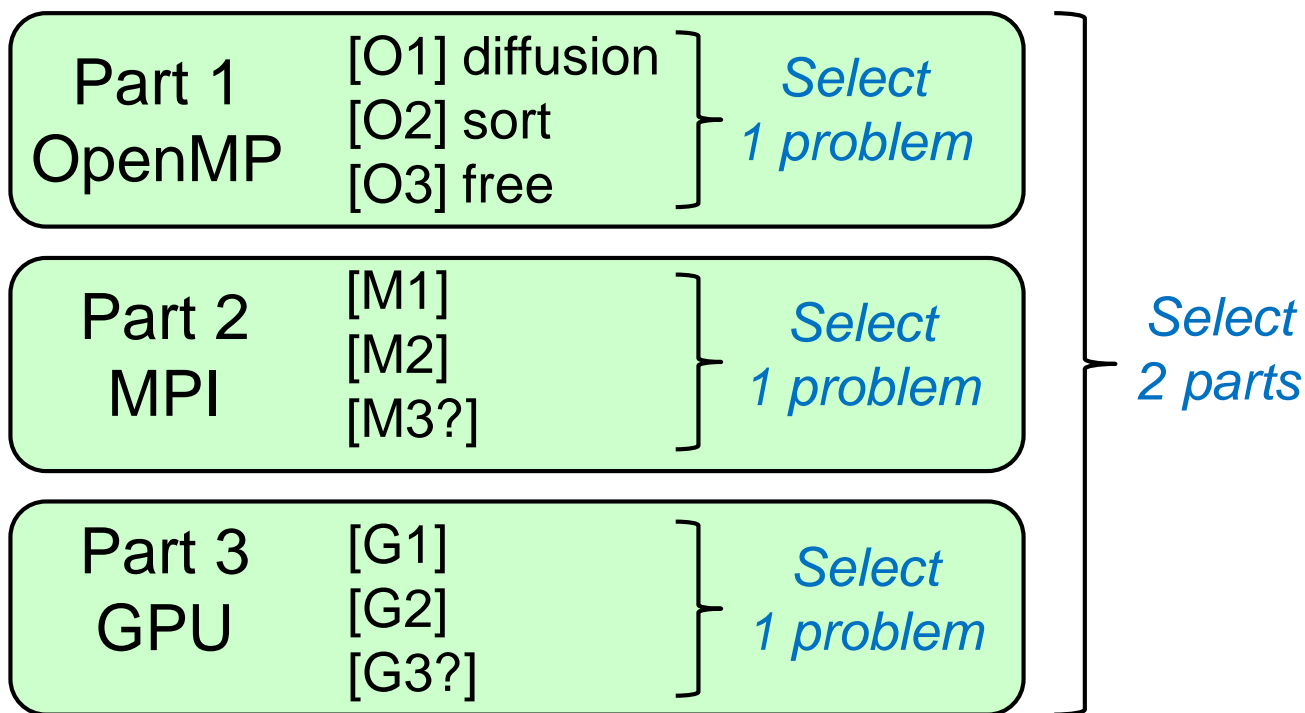


- OpenMP: A programming tool for parallel computation by using multiple processor cores
 - Shared memory parallel model
 - `#pragma omp parallel` → Parallel region
 - `#pragma omp for` → Parallelize for-loops
 - `#pragma omp task` → Task parallelism
- Processor cores we can use are limited a single node
- In MPI part, we will go over the wall of a node

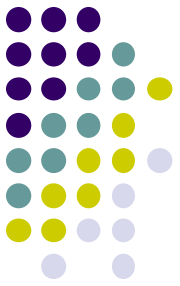
Assignments in this Course



- There is homework for each part. Submissions of reports for **2 parts** are required
- Also attendances will be considered



Assignments in OpenMP Part (Abstract)



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

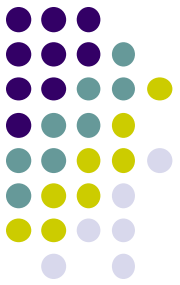
Due date: May 8 (Monday)

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

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

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

For more detail, please see Apr 13 slides or OCW-i.



Next Class:

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