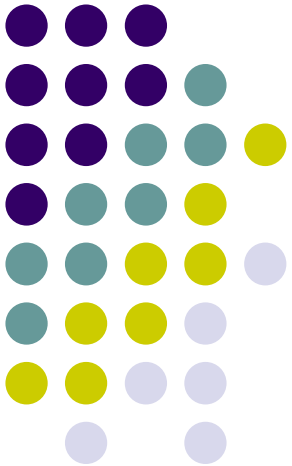
2018 Practical Parallel Computing (実践的並列コンピューティング) No. 5

Shared Memory Parallel Programming with OpenMP (3)

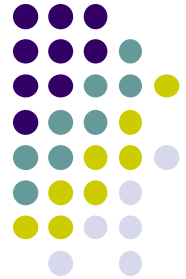


Toshio Endo

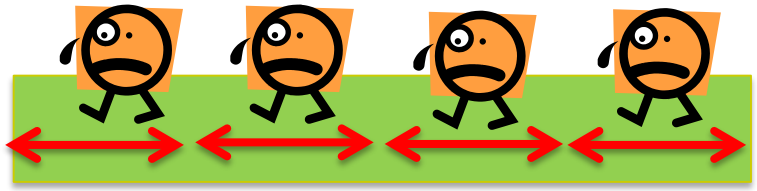
School of Computing & GSIC

endo@is.titech.ac.jp

Today's Topic：Task Parallelism ~Comparison with Data Parallelism~

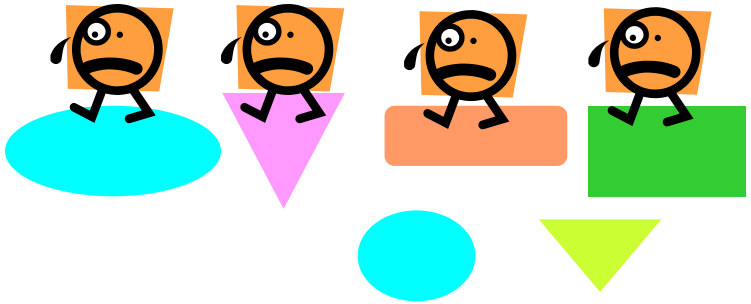


- Data Parallelism：
  - Every thread does uniform/similar tasks for different part of large data



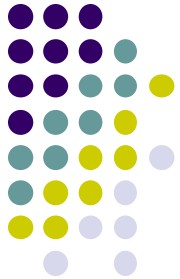
cf) mm, diffusion samples

- Task Parallelism：
  - Each thread does different tasks
    - Sometimes the number of tasks is unknown beforehand
    - Sometimes tasks are generated recursively



cf) fib, sort samples today

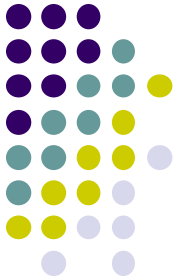
Data Parallelism/Task Parallelism in OpenMP



- #pragma omp for
  - Used for data parallelism (basically)
  - Number of tasks is known before starting for-loop
    - for (i = 0; i < n; i++) … → *n* tasks are divided among threads
- #pragma omp task
  - Used for task parallelism (basically)
  - Number of tasks may change during execution

※ You may write data parallel algorithm with “omp task” if you want, or vice versa

task/taskwait syntaxes



```
#pragma omp parallel
#pragma omp single
{
#pragma omp task
  {
    A;
  }

#pragma omp task
   B;

#pragma omp taskwait
}
```

"task" syntax generates a task (called child task) that executes the following block/sentence

- A task is executed by one of threads who is idle (has nothing to do)
- Child tasks and parent task may be executed in parallel
- Recursive task generation is ok

"taskwait" syntax waits end of all child tasks

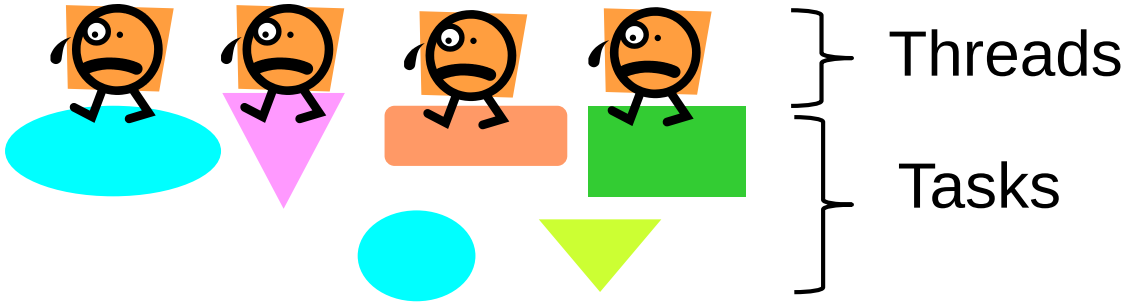
Differences between "Tasks" and "Threads"

Task A and task B are executed in parallel

Thread A and thread B are executed in parallel

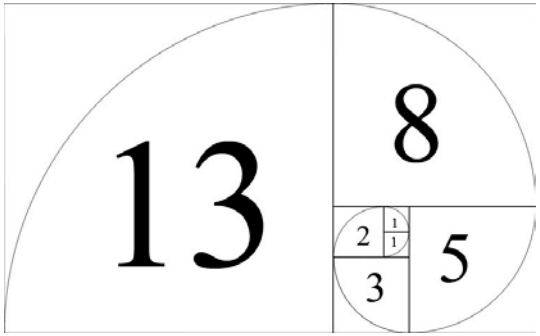
→ So, what is the difference?

- Number of threads is (basically) constant during a parallel region
  - OMP_NUM_THREADS, usually no more than number of processor cores
- Number of tasks may be changed frequently
  - may be >>number of processor cores
- When a thread becomes idle, it takes one of tasks and executes it

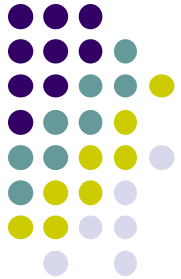


“fib” Sample Program

- Available at ~endo-t-ac/ppcomp/18/fib/
- Calculates the Fibonacci number
  - fib(n) = fib(n-1) + fib(n-2)
  - 1, 1, 2, 3, 5, 8, 13…
- Execution: ./fib [n]
  - ./fib 40 → outputs 40th Fibonacci number
- Recursive function call is used
  - It uses an inefficient algorithm as a sample
- Computational complexity: O(fib(n))
  - (We do not know it before the calculation)



OpenMP Version of fib (version 1)



```
long fib_r(int n)
{
  long f1, f2;
  if (n <= 1) return n;

#pragma omp task shared(f1)
  f1 = fib_r(n-1);

#pragma omp task shared(f2)
  f2 = fib_r(n-2);

#pragma omp taskwait

  return f1+f2;
}
```

Available at

~endo-t-ac/ppcomp/18/fib-slow-omp/

- In this version,

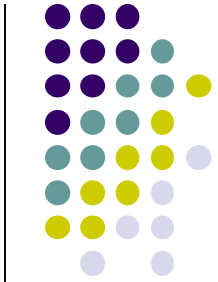
  a task = recursive call

Tasks are generated

We wait for completion of the above 2 tasks

[Q] What if we omit “omp taskwait”?

Note on Using “task” Syntax



- In OpenMP, tasks are taken and executed by idle threads

→ We need to prepare idle threads before creating tasks

```
long fib(int n)
{
  long ans;
#pragma omp parallel
#pragma omp single
  {
    ans = fib_r(n);
  }
  return ans;
}
```

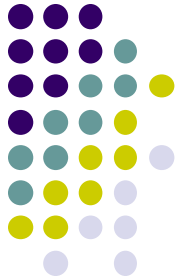
←Multiple threads start

←Only a single thread executes followings (other threads become idle)

←Parallel region finishes

[Q] What if we omit “omp single”?
→ Every thread execute “fib_r(n)” redundantly
→ No speed up!

Rules about Variables



In default, *copies* of variables are created for each child task

- The value of "*n*" is brought from parent to child task
  → OK ☺
- But a child has a only copy → update to "f1" or "f2" is not visible to parent. NG! ☹

"shared(var)" option makes the variable "var" be shared between parent and child

- Using it, update to "f1" or "f2" is visible to parent

The First Version is Too Slow

Execution time of ./fib 40

- TSUBAME3.0 node, compiled with “gcc –O –fopenmp”

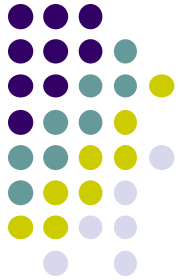
fib	1	
	threads	
	0.60	seconds

fib-slow -omp	1	2	4	8	
	threads				
	33	~300	~360	~480	seconds

- OpenMP version is much slower than original fib
  - With 1 thread, 40x slower
- Also it is much slower with multi-threads

→ How can we improve?

Pitfall in “task” Syntax



- While OpenMP allows to generate many tasks, task generation cost is not negligible

Rough comparison:

Function call cost << Task generation cost

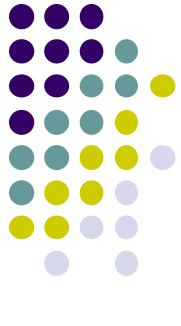
<< Thread generation cost

- In version 1, “./fib n” generates O(fib(n)) tasks

→ Too much!

- How can we reduce the number of tasks?

OpenMP Version of fib (version 2)



```
long fib_r(int n)
{
  long f1, f2;
  if (n <= 1) return n;

  if (n <= 30) {
    f1 = fib_r(n-1);
    f2 = fib_r(n-2);
  }
  else {
#pragma omp task shared(f1)
    f1 = fib_r(n-1);
#pragma omp task shared(f2)
    f2 = fib_r(n-2);
#pragma omp taskwait
  }
  return f1+f2;
}
```

if *n* is “sufficiently” small, we do not generate tasks

Available at

~endo-t-ac/ppcomp/18/fib-omp/

- To avoid generating too many tasks, we check n
  - Changing threshold (=30) would affect performance
- If n is large, we generate tasks
- If n is small, we do not generate

Performance of Version 2

Execution time of ./fib 40

fib

1	threads
0.6	seconds

fib-slow -omp

1	2	4	8	threads
33	~300	~360	~480	seconds

fib-omp

1	2	4	8	threads
0.75	0.46	0.29	0.21	seconds

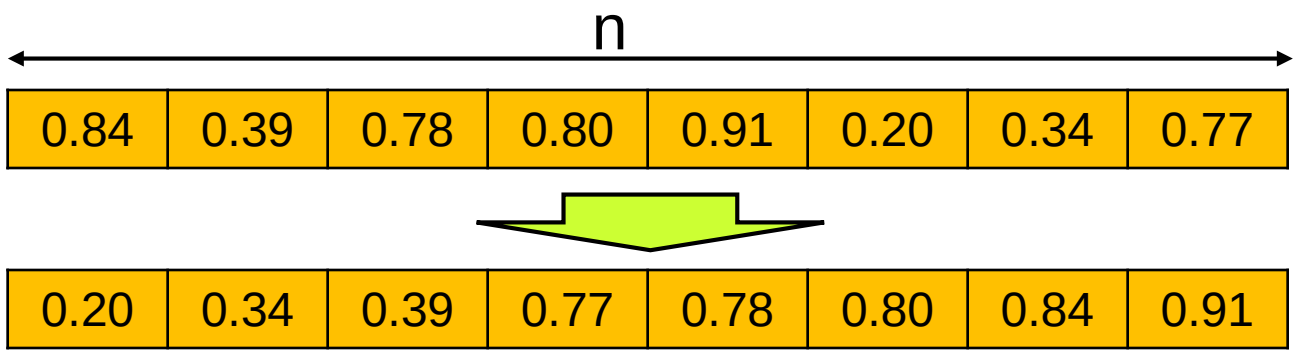
- Performance of Version 2 is largely improved and more stable
  - With 1 thread, still 25% slower than sequential fib

→ Restricting task generation is important for speed

“sort” Sample Program Related to Assignment [O2]

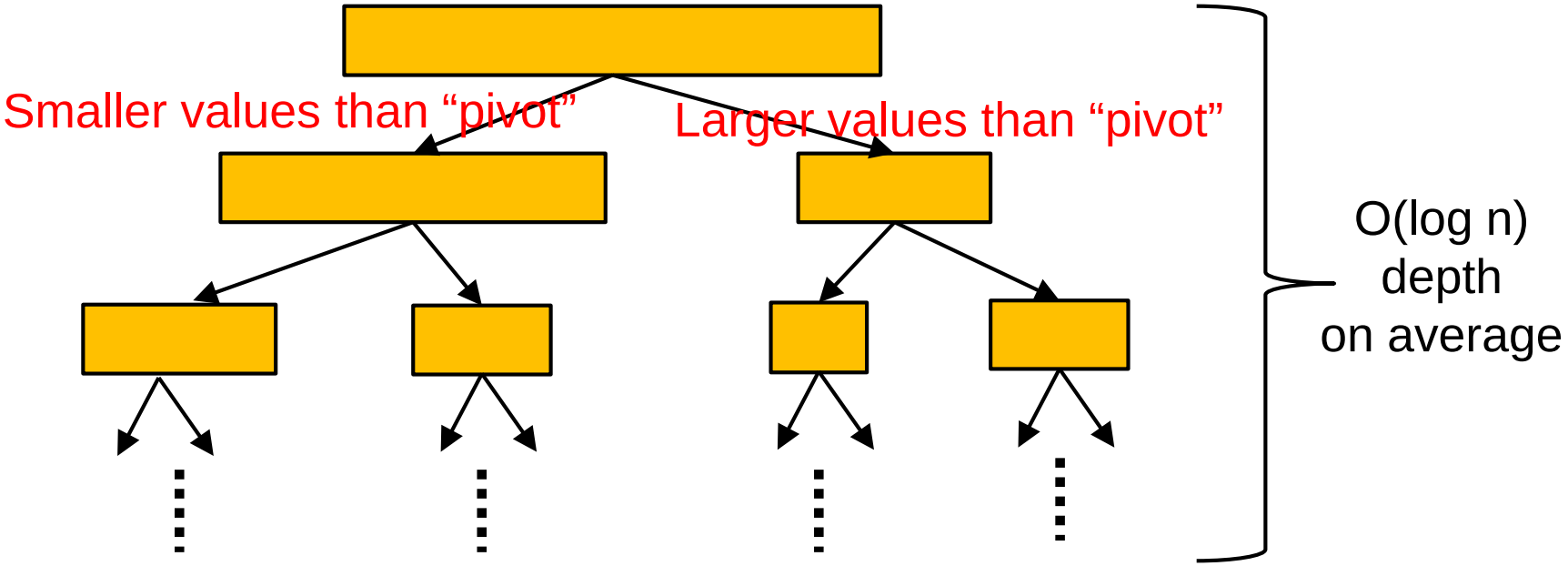
Available at ~endo-t-ac/ppcomp/18/sort/

- Execution: ./sort [n]
- It sorts an array of length *n* by the quick sort algorithm
  - Array elements have double type
- Compute Complexity: O(n log n) on average
  - More efficient than $O(n^2)$ algorithm such as bubble sort



Quick Sort

- A recursive algorithm
  - Take a value, called “pivot” from the array
  - Partition array into two parts, “small” and “large”
  - “small” part and “large” part are sorted recursively



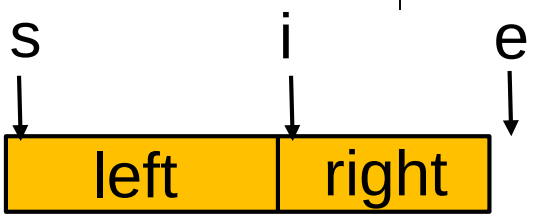
Structure of Sort Sample

```
int sort(double *data, int s, int e)
{
  int i, j;
  double pivot;
  if (n <= 1)  return 0;

  /* pivot selection */
       :

  /* partition data[] into 2 parts */
       :
  /* Here “i” is boundary of 2 parts */

  sort(data, s, i);   /* Sort left part recursively*/
  sort(data, i, e);  /* Sort right part recursively */
}
```

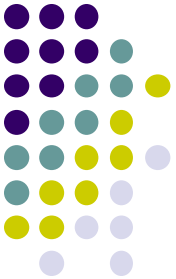


Harder to parallelize (not impossible)

Generating 2 tasks would be a good idea

[Q] Should we restrict too much task generation? And how?

[Revisited] When We Can Use “omp for”



- Loops with some (complex) forms cannot be supported, unfortunately ☹
- The target loop must be in the following form

```
#pragma omp for
  for (i = value; i op value; incr-part)
    body
```

“*op*” : <, >, <=, >=, etc.

“*incr-part*” : i++, i--, i+=c, i-=c, etc.

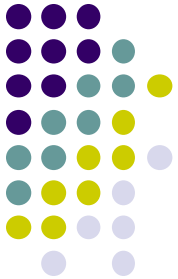
OK ☺: for (x = n; x >= 0; x-=4)

NG ☹: for (i = 0; test(i); i++)

NG ☹: for (p = head; p != NULL; p = p->next)

⇨ *Instead, we can parallelize it with “task” syntax*

Parallelize Irregular Loops with "task" Syntax



- In list search, number of iterations cannot be known before execution → we can use "task"

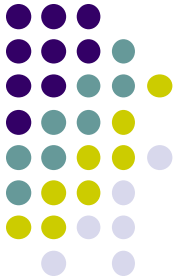
```
#pragma omp parallel
#pragma omp single
  {
     for (p = head; p != NULL;
          p = p->next) {
#pragma omp task
        [Do something with p]
     }
#pragma omp taskwait
  }
```

- A task for one list node = one OpenMP task

Note:
- The number of generated tasks = List length.

→ Task generation costs may be large

Assignments in OpenMP Part (Abstract)



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

Due date: May 7 (Monday)

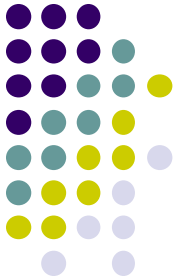
[O1] Parallelize “diffusion” sample program by OpenMP.
(~endo-t-ac/ppcomp/18/diffusion/ on TSUBAME)

[O2] Parallelize “sort” sample program by OpenMP.
(~endo-t-ac/ppcomp/18/sort/ on TSUBAME)

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

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

Next Class:



- OpenMP(4)
  - Mutual execution for correct programs
  - Bottlenecks in parallel programs