

并行与分布式计算基础：第十三讲

杨超

chao_yang@pku.edu.cn

2019 秋



课程基本情况

- 课程名称：并行与分布式计算基础
- 授课教师：杨超（chao_yang@pku.edu.cn，理科 1 号楼 1520）
- 课程助教：尹鹏飞（pengfeiyin@pku.edu.cn）

授课内容（暂定）

- 引言
- 硬件体系架构
- 并行计算模型
- 编程与开发环境
- MPI 编程与实践
- OpenMP 编程与实践
- GPU 编程与实践
- 前沿问题选讲

上课时间（地点：二教 211）

上课时间	星期一	星期二	星期三	星期四	星期五
第 1 节 (8:00-8:50)					
第 2 节 (9:00-9:50)					
第 3 节 (10:10-11:00)				单周	
第 4 节 (11:10-12:00)				单周	
第 5 节 (13:00-13:50)		每周			
第 6 节 (14:00-14:50)		每周			
第 7 节 (15:10-16:00)					
第 8 节 (16:10-17:00)					
第 9 节 (17:10-18:00)					
第 10 节 (18:40-19:30)					
第 11 节 (19:40-20:30)					
第 12 节 (20:40-21:30)					

内容提纲

1 OpenMP 编程-4: 专家篇

- 同步构造
- 任务构造
- 线程控制

同步构造

1 OpenMP 编程-4: 专家篇

- 同步构造
- 任务构造
- 线程控制

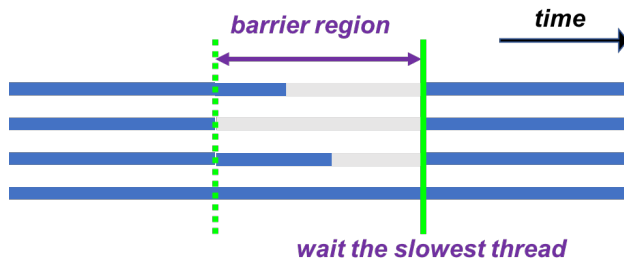
同步构造 (synchronization construct)

- barrier 构造：栅栏同步；
- single 构造：单线程执行，有同步；
- master 构造：主线程执行，无同步；
- ordered 构造：按循环顺序执行；
- critical 构造：各线程依次执行，程序片段；
- atomic 构造：各线程依次执行，单一指令。

barrier 构造

- 在并行区中特定位置显式加入栅栏同步：

```
#pragma omp barrier
```



- 例如:

```
for (i=0; i < N; i++)  
    a[i] = b[i] + c[i];
```

wait !

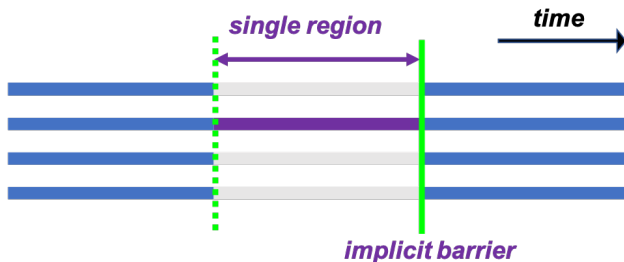
barrier

```
for (i=0; i < N; i++)  
    d[i] = a[i] + b[i];
```


single 构造

- 对并行区内的一段代码单线程执行，有同步 (可用 `nowait` 去掉):

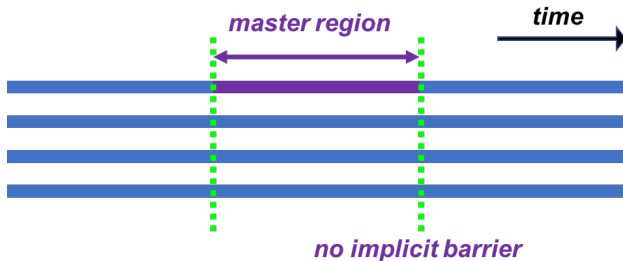
```
#pragma omp single [clause1 clause2 ...]
{
    code();
}
```



master 构造

- 对并行区内的一段代码采用主线程执行，无同步：
(可以看作是一种加上 `nowait` 的特殊版的 `single` 构造)

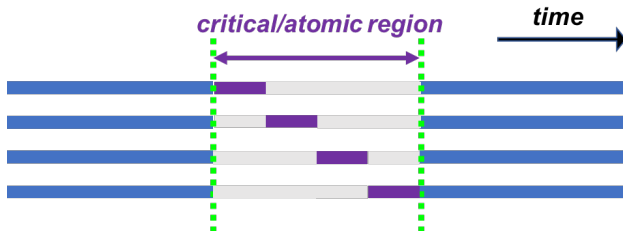
```
#pragma omp master
{
    code();
}
```



critical 构造

- 对并行区内的一段代码依次互斥 (mutual exclusion, mutex) 执行:

```
#pragma omp critical [(name) hint(...)]  
{  
    code();  
}
```



竞争条件 (race condition) 和线程安全 (thread safe)

- 竞争条件 (race condition) 指的是并程序的执行结果具有随机性，依赖于某些事件的发生顺序，在 OpenMP 中竞争条件的产生往往是由于多个线程同时更新同一片内存地址空间 (如共享变量)；
- 称程序为线程安全 (thread safe)，一般指竞争条件可以完全避免，如 I/O 操作、OS 操作、通用库函数等均有可能不是线程安全的，需要使用单线程调用；
- 在 OpenMP 中，避免竞争条件发生的主要手段有：
 - ▶ 使用 `critical` 构造；
 - ▶ 使用 `atomic` 构造；
 - ▶ 使用 `reduction` 从句等。

- 例如，如下操作不是线程安全的：

```
1  #pragma omp parallel for shared(sum,a)
2    for (int i=0; i<n; i++) {
3        a[i] = ...
4        sum += a[i];
5    }
```

- 对此，可做下述修改，避免竞争条件发生：

```
1  #pragma omp parallel for shared(sum,a)
2    for (int i=0; i<n; i++) {
3        a[i] = ...
4    #pragma omp critical
5        sum += a[i];
6    }
```

atomic 构造

- 可认为是一种特殊的 `critical` 构造，对单个特定格式的语句或语句组中某个变量进行原子操作，用法 (如对 `x` 原子操作):

```
#pragma omp atomic read | capture
    something = x;
#pragma omp atomic write | capture
    x = something;
#pragma omp atomic [ update | capture ]
    x = x binop something;
#pragma omp atomic capture
    { x = x binop something; anotherthing = x; }
```

- 详细用法可参见 OpenMP 相关技术手册。

示例：同步构造 (1)

omp_sync.c

```
1  #include <omp.h>
2  #include <stdio.h>
3
4  int main(int argc, char *argv[]){
5      int  nt, tid;
6      int  i, a[10], sum = 0;
7
8      #pragma omp parallel private(i, tid)
9      {
10         nt = omp_get_num_threads();
11         tid = omp_get_thread_num();
12         for (i = tid; i < 10; i += nt) {
13             a[i] = i;
14         }
15     #pragma omp barrier
16     #pragma omp single
17         for (i = 0; i < 10; i++) {
18             printf(" tid = %d/%d, a[%d] = %d\n", tid, nt, i, a[i]);
```

示例：同步构造 (2)

```
19     }
20     #pragma omp for
21     for (i = 0; i < 10; i++) {
22     #pragma omp critical (summation)
23         sum += a[i];
24     }
25     #pragma omp master
26     printf(" sum = %d\n", sum);
27     }
28     return 0;
29 }
```


- 运行结果:

```
tid = 3/4, a[0] = 0
tid = 3/4, a[1] = 1
tid = 3/4, a[2] = 2
tid = 3/4, a[3] = 3
tid = 3/4, a[4] = 4
tid = 3/4, a[5] = 5
tid = 3/4, a[6] = 6
tid = 3/4, a[7] = 7
tid = 3/4, a[8] = 8
tid = 3/4, a[9] = 9
sum = 45
```

- 练习: 尝试修改代码中的各种同步构造, 测试结果。

程序的遗孤 (orphaning)

- 并行区的作用范围
 - ▶ 静态范围 (static extent): 并行区直接影响的代码段;
 - ▶ 动态范围 (dynamic extent): 并行区间间接影响的代码, 例如在并行区内被调用的函数.
- 遗孤 (orphaning): 工作共享和同步构造被放在并行区静态范围外
 - ▶ 如果在动态范围之内, 等同于非遗孤情况;
 - ▶ 否则, 制导语句不起作用.

```
(void) dowork(); !- Sequential FOR  
  
#pragma omp parallel  
{  
    (void) dowork(); !- Parallel FOR  
}
```

```
void dowork()  
{  
    #pragma omp for  
    for (i=0;....)  
    {  
        :  
    }  
}
```

- sections 构造不支持遗孤.

练习

- 修改 `omp_sync.c`, 求和部分的代码用如下函数:

`omp_sync.c`

```
1  int compute_sum(int *pt, int n) {  
2      int i, sum = 0;  
3      #pragma omp for  
4      for (i = 0; i < n; i++) {  
5          sum += pt[i];  
6      }  
7      return sum;  
8  }
```

任务构造

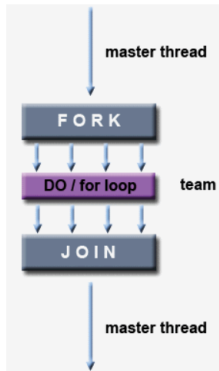
1 OpenMP 编程-4: 专家篇

- 同步构造
- 任务构造
- 线程控制

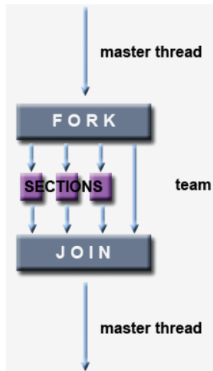
回忆：工作共享构造 (work-sharing construct)

用于将代码分配采用某种机制给不同的线程执行：循环、分块、单独
(注：在入口没有同步，但是在出口包含了一个隐含的栅栏同步)。

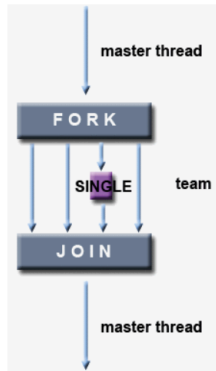
DO / for - shares iterations of a loop across the team. Represents a type of "data parallelism".



SECTIONS - breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional parallelism".



SINGLE - serializes a section of code



OpenMP 工作共享构造的缺陷

- 任务必须可数，比如，下面的任务 (如链表、递归等) 无法支持：

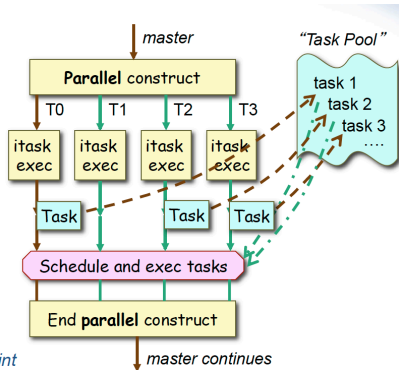
```
#pragma omp parallel
{
    ...
    while (my_pointer != NULL) {
        do_independent_work(my_pointer);
        my_pointer = my_pointer->next;
    } // End of while loop
    ...
}
```

- ▶ 只支持任务可数的情况 (for 循环或者 section 区块);
- ▶ 如果不能转换为可数任务，缺乏灵活的任务处理机制;
- ▶ OpenMP3.0 开始支持的任务并行是一个有益补充。

OpenMP 的任务并行 (task parallelism)

- 显式定义一系列可执行的任务及其相互依赖关系，通过任务调度的方式多线程动态执行，支持任务的延迟执行 (deferred execution)。

- Starts with the *master* thread
- Encounters a **parallel** construct
 - Creates a team of threads, *id 0* for the *master* thread
 - Generates implicit tasks, one per thread
 - Threads in the team executes implicit tasks
- Encounters a worksharing construct
 - Distributes work among threads (or implicit tasks)
- Encounters a **task** construct
 - Generates an explicit task
 - Execution of the task could be deferred
- Execution of explicit tasks
 - Threads execute tasks at a *task scheduling point* (such as **task**, **taskwait**, **barrier**)
 - Thread may switch from one task to another task
- At the end of a **parallel** construct
 - All tasks complete their execution
 - Only the *master* thread continues afterwards



---➔ may be deferred

←... scheduling

- ♦ *implicit tasks cannot be deferred*
- ♦ *explicit tasks could be deferred*

OpenMP 任务构造 (1)

- 定义任务:

```
#pragma omp task [clause1 clause2]  
...
```

- 支持的从句:

```
if (scalar expression)  
final (scalar expression)  
untied  
default (shared | none)  
mergeable  
private (list)  
firstprivate (list)  
shared (list)
```


OpenMP 任务构造 (2)

- 完成任务 (含子任务)

- ▶ 自动完成：在程序的显式或者隐式同步点；
- ▶ 手动完成：

```
#pragma omp taskwait
```

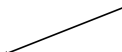
```
...
```

- 变量的数据域

- ▶ 并行区中的共享变量：在 task 区中默认也为共享；
- ▶ 并行区中的私有变量：在 task 区中默认为 firstprivate；
- ▶ task 区中的其他变量：默认为私有。

```
#pragma omp parallel shared(A) private(B)
{
    ...
    #pragma omp task
    {
        int C;
        compute(A, B, C);
    }
}
```

A is shared
B is firstprivate
C is private



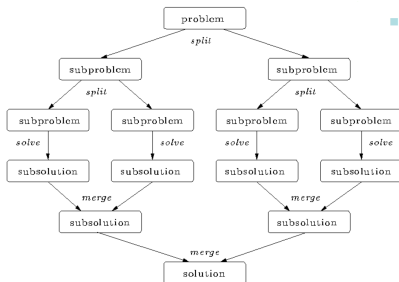
例子：计算斐波那契数

- 斐波那契数 (Fibonacci Number):

$$F(n) = \begin{cases} n, & \text{if } n = 0, 1, \\ F(n-1) + F(n-2), & \text{otherwise.} \end{cases}$$

- 分而治之 (Divide and Conquer):

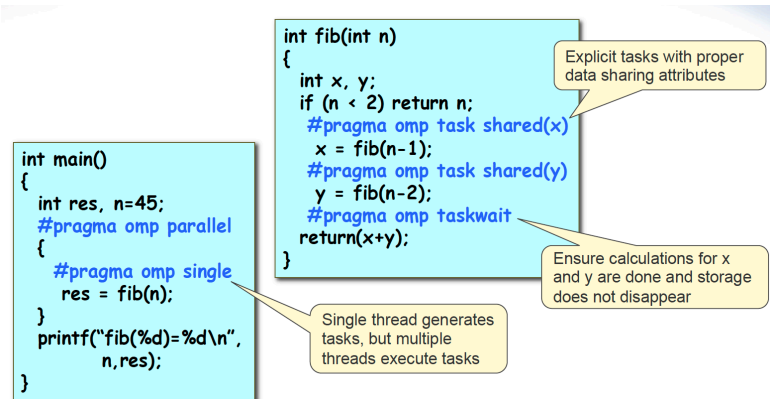
Split the problem into smaller sub-problems; continue until the sub-problems can be solve directly



3 Options:

- ☐ Do work as you split into sub-problems
- ☐ Do work only at the leaves
- ☐ Do work as you recombine

- 递归实现：创建一个二叉任务树，从叶子结点开始进行并行执行。



- 思考：为什么性能不好？

- 任务并行的额外开销 (overhead): 任务调度 (task scheduling)。
- 任务粒度 (granularity): 每个任务的 “大小”
 - ▶ 粒度太大: 负载不平衡;
 - ▶ 粒度太小: 调度开销过大。
- 思考 1: 如何判断调度开销是否过大?
- 答案: 试一下串行运行, 比较时间.
- 思考 2: 如何减少上述程序的调度开销?
- 答案: 增大任务粒度, 在 n 过小时不再生产任务.
- 调整了任务粒度后, 感觉还是太慢?

- 修改一下要求：求第 0 至 n 个数。

omp_fib2.c

```
1 long fib(int n) {
2     long x, y, res;
3     if      (n < 2)  res = n;
4     else if (n < 30) res = fib(n-1) + fib(n-2);
5     else {
6         #pragma omp task shared(x)
7         x = fib(n-1);
8         #pragma omp task shared(y)
9         y = fib(n-2);
10        #pragma omp taskwait
11        res = x + y;
12    }
13    a[n] = res;
14    return a[n];
15 }
```

- 思考：更慢了，到底哪里慢了呢？

线程控制

1 OpenMP 编程-4: 专家篇

- 同步构造
- 任务构造
- 线程控制

动态线程

- 动态线程：系统动态选择并行区的线程数 (默认：一般为关闭)。
- 打开/关闭动态线程

- ▶ 库函数：

```
void omp_set_dynamic(int flag)
```

- ▶ 环境变量：

```
export OMP_DYNAMIC=true
```

- 检查动态线程是否打开

- ▶ 库函数：

```
int omp_get_dynamic (void)
```

- 一个小例子:

- ▶ flag 为 0: 并行区开启 10 个线程;
- ▶ flag 非 0: 并行区开启 1-10 个线程 (系统决定)。

```
1  ...
2  omp_set_dynamic(flag);
3  #pragma omp parallel num_threads(10)
4  {
5      /* do work here */
6  }
7  ...
```


嵌套并行 (nested parallelism)

- 嵌套并行：指在并行区之内开启并行区 (默认：一般为开启)。
- 打开/关闭嵌套并行

- ▶ 库函数：

```
void omp_set_nested(int flag)
```

- ▶ 环境变量：

```
export OMP_NESTED=true  
export OMP_NUM_THREADS=n1,n2,n3,...
```

- 检查嵌套并行是否打开

- ▶ 库函数：

```
int omp_get_nested (void)
```

- 思考：下面的例子运行结果是什么？

omp_nested.c

```
1  ...
2  omp_set_dynamic(0);
3  #pragma omp parallel num_threads(2)
4  {
5
6      omp_set_nested(1);
7  #pragma omp parallel num_threads(3)
8  {
9      #pragma omp single
10     printf ("Inner: num_thds=%d\n", omp_get_num_threads());
11 }
12
13 #pragma omp barrier
14     omp_set_nested(0);
15 #pragma omp parallel num_threads(3)
16 {
17     #pragma omp single
```

```
18     printf ("Inner: num_thds=%d\n", omp_get_num_threads());
19 }
20
21 #pragma omp barrier
22 #pragma omp single
23     printf ("Outer: num_thds=%d\n", omp_get_num_threads());
24
25 }
26 ...
```