

COMP4007: 并行处理和体系结构

第六章: OpenMP 3.0与任务分配

授课老师: 王强、施少怀
助 教: 刘虎成、林稳翔

哈尔滨工业大学 (深圳)

内容大纲

- ▶ OpenMP 3.0 功能介绍
- ▶ 任务分配

OpenMP 3.0 之前的版本

▶ 大多数情况下很有效

✎ 作为OpenMP的“老大哥”必须明确所有事情

☞ 运行时循环长度必须已知

☞ 有限数量的并行区域

▶ 对某些常见问题效果欠佳

✎ 链表

✎ 分治问题

✎ 递归算法

OpenMP 3.0 - 主要新功能

▶ 任务分配

✎ 支持任务级别的并行化，支持复杂和动态的控制流

▶ 支持嵌套并行

✎ 更好地定义和控制嵌套并行区域，使用新的应用程序接口确定嵌套结构

▶ 增强循环调度

✎ 支持更激进的编译器优化和更好的运行时控制

▶ 循环折叠(loop collapse)

✎ 支持组合嵌套循环以实现更多并发性

新指令、应用程序接口和环境变量



▶ 两条新指令

✎ Task

✎ Taskwait

▶ 九个新的 API 例程

✎ omp_set_schedule, omp_get_schedule

✎ omp_get_ancestor_thread_num, omp_get_team_size

✎ omp_get_level, omp_get_active_level,

✎ omp_get_thread_limit

✎ omp_set_max_active_levels, omp_get_max_active_levels

▶ 四个新环境变量

✎ OMP_STACKSIZE, OMP_WAIT_POLICY

✎ OMP_THREAD_LIMIT, OMP_MAX_ACTIVE_LEVELS

▶ 支持设置线程数

✎ 通过API `omp_set_num_threads`

☞ 2.5版本仅支持在最外层通过`num_threads`设置

☞ 3.0版本支持在所有层级设置

▶ 查询嵌套层级

✎ 增加了用于查询嵌套层级、线程 ID 和团队规模的API例程

▶ 资源限制

✎ 最大线程数

☞ `OMP_THREAD_LIMIT`, `omp_get_thread_limit`

✎ 最大嵌套层级

☞ `OMP_MAX_ACTIVE_LEVELS`,

☞ `omp_set/get_max_active_levels`

▶ 静态调度(STATIC)和立即调度(NOWAIT)

✎ 2.5版本不保证NOWAIT的安全性

✎ 3.0版本明确了STATIC的含义

▶ SCHEDULE(AUTO)

✎ 允许进行灵活的调度

▶ SCHEDULE(RUNTIME)的API

✎ `omp_set_schedule(kind,modifier)`

✎ `omp_get_schedule(kind,modifier)`

`kind = {omp_sched_static, omp_sched_dynamic,
 omp_sched_guided, omp_sched_auto}
modifier = chunk_size`

循环折叠

▶ 对于完美嵌套的循环

```
#pragma omp parallel for
for (int i = 0; i < 4; i++)
{
    for (int j = 0; j < 5; j++)
    {
        printf("Thread number is %d\n",
            omp_get_thread_num());
    }
}
```

仅有 4 个线程处于活动状态。

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < 4; i++)
{
    for (int j = 0; j < 5; j++)
    {
        printf("Thread number is %d\n",
            omp_get_thread_num());
    }
}
```

20个线程处于活动状态！

▶ OMP_WAIT_POLICY

✎ 控制线程在屏障(barriers)和锁(locks)处的行为方式

✎ 取值

☞ ACTIVE - 提高应用程序性能

☞ PASSIVE - 提高系统响应速度

▶ OMP_STACKSIZE

✎ 控制 OpenMP 线程堆栈大小

✎ 取值形如

☞ size | sizeB | sizeK | sizeM | sizeG (默认为K)

- ▶ OpenMP 可执行指令(Directives)后的第一个结构化代码块或 OpenMP 结构体。结构化代码块: 代码块顶部有一条单一或复合声明形式的开始指令, 底部有一条类似的结束指令
- ▶ **parallel**结构: 并行结构会形成一组线程, 能够并行执行。

```
#pragma omp parallel [clause[ [ , ]clause] ...] new-line structured-block
```

```
clause:  if (scalar-expression)
```

```
         num_threads (integer-expression)
```

```
         default (shared | none)
```

```
         private (list)
```

```
         firstprivate (list)
```

```
         shared (list)
```

```
         copyin (list)
```

```
         reduction (operator: list)
```

OpenMP 3.0 C/C++ 语法总结 – 指令(续)



- ▶ 循环结构规定，循环的迭代将分配给线程组执行

#pragma omp for [*clause*[[, *clause*] ...] *new-line for-loops*

clause: **private** (*list*)
 firstprivate(*list*)
 lastprivate (*list*)
 reduction (*operator: list*)
 schedule (*kind*[, *chunk_size*])
 collapse(*n*)
 ordered
 nowait

最常见的*for-loop*格式如下:

```
for(var = lb; var relational-op b; var += incr)
```

OpenMP 3.0 C/C++ 语法总结 – 指令(续)



- ▶ **sections**结构包含一组结构化代码，这些模块将分配给线程组执行

```
#pragma omp sections [clause[[ , ] clause] ...] new-line
{
    [#pragma omp section new-line]
    structured-block
    [#pragma omp section new-line structured-block]
    ...
}
clause: private (list)
        firstprivate (list)
        lastprivate (list)
        reduction (operator: list)
        nowait
```

OpenMP 3.0 C/C++ 语法总结 – 指令(续)



- ▶ **single**结构指接下来的代码结构块只能由线程组中的某个线程执行（不一定是主线程），运行条件隐含任务的上下文

```
#pragma omp single [clause[[ ,] clause] ...] new-line  
    structured-block  
clause:  private (list)  
         firstprivate (list)  
         copyprivate (list)  
         nowait
```

OpenMP 3.0 C/C++ 语法总结 – 指令(续)

- ▶ **task**定义了一个显式任务。任务的数据环境会根据任务构造块的数据共享属性子句和默认值创建。

```
#pragma omp task [clause [ [ , ] clause ] ... ] new-line  
                structured-block  
clause:  if (scalar-expression)  
          untied  
          default (shared | none)  
          private (list)  
          firstprivate (list)  
          shared (list)
```

OpenMP 3.0 C/C++ 语法总结 – 指令(续)



- ▶ **master**指定了一个由线程组中主线程执行的结构块。在进入或退出**master**结构时没有隐含的障碍

#pragma omp master *new-line*
structured-block

- ▶ **critical**将相关结构化代码块的执行限制为一次只能由一个线程执行

#pragma omp critical [(*name*)] *new-line*
structured-block

- ▶ **barrier**在该结构出现的位置指定了一个显式的同步

#pragma omp barrier *new-line*

- ▶ **taskwait** 指定在当前任务开始之前先等待已生成的子任务运行结束

#pragma omp taskwait *new-line*

OpenMP 3.0 C/C++ 语法总结 – 指令(续)

- ▶ **atomic** 确保以原子方式更新特定存储位置，而不是将其同时暴露给多个写入线程

#pragma omp atomic *new-line*

expression-stmt

expression-stmt: 可取如下形式之一

x binop = expr

x++

++x

x--

--x

- ▶ **flush** 使线程的内存临时视图与全局内存一致，并强制执行变量写内存操作顺序。

#pragma omp flush [*(list)*] *new-line*

- ▶ **ordered**指定了循环区域中的结构块，该结构块将按照循环迭代的顺序执行。这就对有序区域内的代码进行了排序，同时允许区域外的代码并行运行。

```
#pragma omp ordered new-line  
                structured-block
```

- ▶ **threadprivate**指定对变量进行复制方式，且运行过程中每个线程都有自己的副本

```
#pragma omp threadprivate(list) new-line
```

OpenMP 3.0 C/C++ 语法总结 – 子句

- ▶ 并非所有子句(Clauses)都对所有指令有效。对特定指令有效的子句详见该指令的描述。大多数子句接受以逗号分隔的列表项, 这些列表项必须是可见的。
- ▶ 数据共享属性子句: 该子句只适用于该子句所在结构体中可见的变量。
 - ✎ **default(shared|none):** 更改**parallel**或**task**构造的变量的默认数据共享属性。
 - ✎ **shared(list):** 声明由**parallel**或**task**构造生成的任务共享一个或多个列表项。
 - ✎ **private(list):** 声明一个或多个列表项是任务私有的
 - ✎ **firstprivate(list):** 声明一个或多个列表项是任务私有的, 并用遇到该结构体时相应原始项的值初始化每个列表项。
 - ✎ **lastprivate(list):** 指定一个或多个列表项为隐式任务的私有项, 并在结构运行结束后更新相应的原始项。
 - ✎ **reduction(operator:list):** 使用指定的关联运算符对列表项中声明的项进行归约操作。每个列表项都会累加为一个私有副本, 然后与原始项合并。

OpenMP 3.0 C/C++ 语法总结 – 子句

- ▶ 数据复制子句: 支持将数据值从一个隐式任务或线程上的私有变量或线程私有变量复制到团队中其他隐式任务或线程上的相应变量。
 - ✎ **copyin**(*list*): 将主线程的线程私有变量值复制到执行**parallel**区域的团队其他成员的线程私有变量中
 - ✎ **copyprivate**(*list*): 将一个隐式任务的数据环境中的值广播到属于该**parallel**区域的其他隐式任务的数据环境中。

- ▶ 运行时库函数: 执行环境函数能够监控线程、处理器和并行环境。锁函数支持与OpenMP 锁同步。计时函数支持可移植的挂钟计时。运行时库函数的原型定义在文件 "omp.h "中。
- ▶ 执行环境函数
 - ✎ **void omp_set_num_threads(int num_threads):**影响未指定 num threads 子句的后续**parallel**结构块使用的线程数
 - ✎ **int omp_get_num_threads(void):** 返回当前线程组中的线程数
 - ✎ **int omp_get_max_threads(void):** 返回未指定num_threads子句的**parallel**结构块能够使用的最多的线程数
 - ✎ **int omp_get_thread_num(void):** 返回当前运行任务的线程的ID, 注意序号从0开始
 - ✎ **int omp_get_num_procs(void):** 返回程序可用的处理器数量
 - ✎ **int omp_in_parallel(void):** 当线程正在运行**parallel**结构块中的并行语句时返回`true`, 其他情况返回`false`
 - ✎ **void omp_set_dynamic(int dynamic_threads):** 开启/关闭线程数自动调节
 - ✎ **int omp_get_dynamic(void):**返回内部控制变量 (ICV) *dyn-var*的值, 该值决定线程数的动态调整是启用还是禁用



▶ 执行环境函数

- ✎ **int omp_get_thread_limit(void):** 返回程序可用的OpenMP线程数的上限
- ✎ **void omp_set_max_active_levels(int max_levels):** 通过设置内部控制变量`max-active-levels-var`控制同时运行的嵌套parallel结构的最大深度
- ✎ **int omp_get_max_active_levels(void):** 返回内部控制变量`max-active-levels-var`的值
- ✎ **int omp_get_level(void):** 返回正在运行的嵌套并行区域的层级(包含调用者)
- ✎ **int omp_get_ancestor_thread_num(int level):** 返回当前线程给定嵌套层的祖先线程或当前线程的线程编号
- ✎ **int omp_get_team_size(int level):** 对于当前线程的嵌套层, 返回其祖先线程或当前线程所属线程组的大小
- ✎ **int omp_get_active_level(void):** 计算包含调用任务的嵌套活动parallel结构的数量

▶ 计时函数

- ✎ **double omp_get_wtime(void):** 以秒为单位返回挂钟走过的时间
- ✎ **double omp_get_wtick(void):** 返回omp_get_wtime所用计时器的精度

- ▶ 环境变量名称为大写字母，分配给它们的值不区分大小写，可以有前导空格和尾部空格。
- ▶ **OMP_SCHEDULE** *type[,chunk]*: 设置运行时的调度类型(*run-sched-var*)和分块大小。OpenMP支持的调度类型为**static**、**dynamic**、**guided**或**auto**，Chunk 为正整数
- ▶ **OMP_NUM_THREADS** *num* : 设置控制**parallel**结构可使用的线程数内部变量 *nthreads-var*
- ▶ **OMP_DYNAMIC** *dynamic*: 设置控制**parallel**结构运行时是否可自动调节线程数的变量*dyn-var*. 有效值取值为*true*或*false*
- ▶ **OMP_NESTED** *nested*: 设置控制是否允许嵌套并行的变量*nest-var*, 有效的取值为*true*或*false*

- ▶ **OMP_STACKSIZE** *size*: 设置控制OpenMP线程堆栈大小的内部变量*stacksize-var*, *size*取值必须为正整数且形如*size*, *sizeB*, *sizeK*, *sizeM*, *sizeG*. 如果大小单位**B**, **K**, **M**或**G**被省略, 那么默认单位为千字节(**K**)
- ▶ **OMP_WAIT_POLICY** *policy*: 设置控制等待线程预期行为的内部变量*wait-policy-var*. 有效取值为**active**(允许等待线程占用CPU时钟周期)和**passive**
- ▶ **OMP_MAX_ACTIVE_LEVEL** *levels*: 设置控制活动的嵌套**parallel**结构层级的内部变量*max-active-levels-var*.
- ▶ **OMP_THREAD_LIMIT** *limit*: 设置控制程序可使用的OpenMP线程数上限的内部变量*thread-limit-var*.

OpenMP 3.0 C/C++ 语法总结 – 补充

▶ 归约操作合法的操作符

操作符	初始化值
+	0
*	1
-	0
&	~0
	0
^	0
&&	1
	0

▶ 循环结构的调度类型

- ▶ **static** 迭代被划分为大小为 `chunk_size` 的迭代块，这些迭代块按照线程数的顺序，以循环方式分配给线程组中的线程
- ▶ **dynamic** 每个线程执行一个迭代块，完成后请求另一个迭代块，直到完成所有迭代块
- ▶ **guided** 每个线程执行一个迭代块，完成后请求另一个迭代块，直到完成所有迭代块。分块大小从大开始，慢慢缩小到指定的 `chunk_size` 大小
- ▶ **auto** 调度的决定权下放给编译器和/或运行时系统
- ▶ **runtime** 调度类型和分块大小取自运行计划变量 `run-sched-var`

<https://www.openmp.org/wp-content/uploads/OpenMP3.0-SummarySpec.pdf>

<https://www.openmp.org/wp-content/uploads/spec30.pdf>

OpenMP 3.0 任务执行模型视图



▶ 任务由一批独立的工作组成

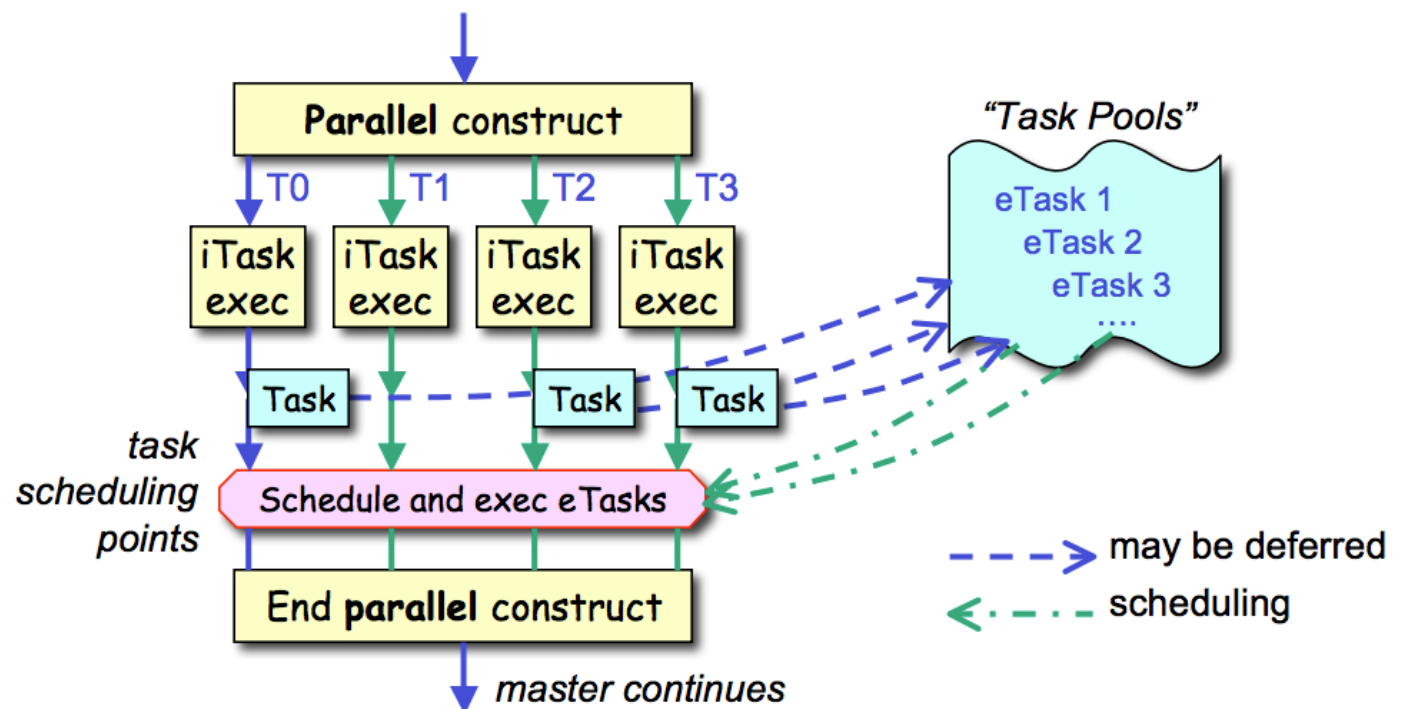
✎ 可同时执行不同任务

▶ 运行时系统决定任务的调度

✎ 在某些点上（隐式的和显式的），运行时保证完成任务

✎ **parallel**指令生成的隐式任务

✎ **task**指令生成的显式任务



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



OpenMP 3.0 任务(Task)



```
#pragma omp task [clause...]  
{“this is my task”}
```

▶ 数据环境与任务相关联

- ✎ 隐式任务默认数据属性为**shared**
- ✎ 大多数情况下显式任务默认数据属性为**firstprivate**

▶ 任务同步

- ✎ **taskwait** 可同步生成的子任务
- ✎ 隐式或显示的**barriers**可以同步所有的显式任务

▶ 任务独占锁

- ✎ 由某一任务设置锁只能由同一任务取消

▶ clause定义如下

▶ clause定义如下

- ✎ if (scalar-expr)
- ✎ final (scalar-expr)
- ✎ untied
- ✎ default(shared/none)
- ✎ private(list)
- ✎ firstprivate(list)
- ✎ shared(list)



在OpenMP中, `barrier` 和 `taskwait` 都是用于控制并发执行的指令, 但它们有不同的作用和使用场景。

1. `#pragma omp barrier`:

- `barrier` 指令用于确保所有线程在并行区域中的执行都达到了同一点, 然后等待其他线程也到达这一点。它会阻塞所有线程, 直到所有线程都执行到 `barrier` 处才会继续执行后面的代码。
- 主要用于同步线程的执行, 确保在执行后面的代码之前, 所有线程都完成了在并行区域中的任务。

```
c Copy code

#pragma omp parallel
{
    // 并行区域
    // ...

    #pragma omp barrier // 所有线程在此等待, 直到所有线程都执行到这里

    // 后续代码
    // ...
}
```

2. `#pragma omp taskwait`:

- `taskwait` 指令用于等待所有与当前线程关联的任务完成执行, 然后才能继续执行后面的代码。它主要用于同步任务的执行。

#pragma omp barrier:

- `barrier` 指令用于确保所有线程在并行区域中的执行都达到了同一点, 然后等待其他线程也到达这一点。它会阻塞所有线程, 直到所有线程都执行到 `barrier` 处才会继续执行后面的代码。
- 主要用于同步线程的执行, 确保在执行后面的代码之前, 所有线程都完成了在并行区域中的任务。



```
c Copy code

#pragma omp parallel
{
    // 并行区域
    // ...

    #pragma omp single
    {
        // 创建任务
        #pragma omp task
        {
            // 任务的执行代码
        }

        // 创建另一个任务
        #pragma omp task
        {
            // 另一个任务的执行代码
        }

        #pragma omp taskwait // 等待所有与当前线程关联的任务完成

        // 后续代码
        // ...
    }
}
```

- 与 `barrier` 不同，`taskwait` 只等待与当前线程关联的任务完成，而不是等待所有线程。如果在嵌套任务中，`taskwait` 只等待当前任务的子任务完成。
- 总的来说，`barrier` 用于等待所有线程达到某一点，而 `taskwait` 用于等待与当前线程关联的任务完成。在并行编程中，根据具体的需求和程序结构选择使用合适的同步机制。

▶ 并行区域的一些适用规则

- ✎ 静态变量和全局变量的数据属性为**shared**
- ✎ 自动存储(本地)变量的数据属性为**private**

▶ 如果默认情况下没有派生**shared**区域

- ✎ 孤立任务变量默认属性为**firstprivate**!
- ✎ 非孤立任务变量继承**shared**属性!
- ✎ 除非在变量在外层上下文中是共享, 否则变量的属性为**firstprivate**


任务数据域: 示例




```
int a = 1;
void foo()
{
    int b = 2, c = 3;
    #pragma omp parallel private(b)
    {
        int d = 4;
        #pragma omp task
        {
            int e = 5;
        }
    }
}
```


// Scope of a: shared
// Scope of b: firstprivate
// Scope of c: shared
// Scope of d: firstprivate
// Scope of e: private

▶ 隐式或显式的**barrier**

 `#pragma omp barrier`


 等待当前并行区域内生成的所有显式任务完成

▶ 利用**taskwait**

 `#pragma omp taskwait`

 确保完成在此之前生成的所有子任务

▶ 在任务中间同步两项任务

 使用锁或“flush”指令

任务同步 – 示例



```
#pragma omp parallel num_threads(np)
{
    #pragma omp task
    function_A();
    #pragma omp barrier
    #pragma omp single
    {
        #pragma omp task
        function_B();
    }
}
```


- ▶ 默认值：任务与首先执行它们的线程绑定→不一定是创建者. 调度存在以下约束
 - ✎ 只有与任务绑定的线程才能执行该任务
 - ✎ 任务只能在任务调度点暂停
 - ☞ Task creation, task finish, taskwait, barrier, taskyield
 - ✎ 如果任务未在barrier中暂停，执行线程只能切换到与该线程绑定的该任务的直系后代任务
- ▶ 使用未绑定子句(untied)创建的任务永远不会绑定
 - ✎ 在任务调度点恢复时可能由不同线程执行
 - ✎ 实施自由度更大，例如负载平衡

▸ 好的方面

- ✎ 数据与任务相关联。它迫使程序员需认真考虑数据的位置，从而编写出更好的代码
- ✎ 工作窃取式的调度（可能）可提高性能
- ✎ 应用范围更广

▸ 收获

- ✎ 程序员必须谨慎的对待变量的作用域，确保它们不会在任务执行结束前消失或超出作用域
- ✎ 拥有**threadprivate**的数据可能无法保存

任务切换 – 难点



▶ 定义

✎ 线程将其执行的指令从一个任务切换到另一个任务的行为

▶ 动作

✎ 暂停或完成当前任务

✎ 恢复或开始另一项任务（有限制条件）

✎ 只能在任务调度点发生

▶ 任务调度点

✎ 紧接着任务构建

✎ 任务区域结束时

✎ **taskwait**位置，隐式或显式的**barrier**位置

✎ 在指定的执行位置（仅适用于不绑定的任务）



任务切换 – 糟糕的方面

▶ 定义

✎ 一个线程暂后却由另一个线程恢复的任务

▶ 绑定(Tied) vs 非绑定(Untied)的任务

✎ 绑定任务不允许切换执行线程（即任务绑定在同一线程上）

☞ 创建任务时没有使用“untied”子句，隐式任务

✎ 非绑定任务允许切换执行线程

☞ 创建任务时使用“untied”子句

▶ 影响

✎ 拥有threadprivate的数据可能无法保留

✎ 锁只能由任务而非线程拥有



任务切换 – 不优雅方面

▶ **if(expr)** 值为 **false** 的任务

✎ 任务（无论是否绑定）会立即执行

▶ 对于非绑定任务

✎ 未规定如何调度

✎ 依靠更智能的编译器/实现来提高性能

▶ 不处于**barrier**位置的绑定任务

✎ 受任务调度制约

☞ 由与线程绑定的一组任务区域定义

☞ 任务可被调度的情况

▢ 任务集合为空，或

▢ 该任务是任务集中所有任务的后代

- ▶ 任务区域(Task region)
 - ✎ 任务执行所需的代码组成
- ▶ 后裔任务(Descendant task)
 - ✎ 任务的子任务或其子代任务
- ▶ 祖先线程(Ancestor thread)
 - ✎ 父线程或其父线程的一个祖先线程
- ▶ 活跃的并行区域(Active parallel region)
 - ✎ 团队中有一个以上线程的并行区域
- ▶ 非活跃并行区域(Inactive parallel region)
 - ✎ 团队中只有一个线程的并行区域

示例 1

- 编写一个程序，打印“A race car”或“A car race”，并最大限度地提高并行性

```
#include<stdlib.h>
#include<stdio.h>
int main(int argc, char *argv[])
{
    printf("A ");
    printf("race ");
    printf("car ");
    printf("\n");
    return 0;
}
```

输出:

“A race car”

```
#include<stdlib.h>
#include<stdio.h>
int main(int argc, char *argv[])
{
    #pragma omp parallel
    {
        printf("A ");
        printf("race ");
        printf("car ");
    }
    printf("\n");
    return 0;
}
```

输出:

“A A race race car car” or

“A race A car race car” or

...

示例 1(续)



```
#include<stdlib.h>
#include<stdio.h>
int main(int argc, char *argv[])
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            printf("race ");
            printf("car ");
        }
    }
    printf("\n");
    return 0;
}
```

输出: (只有1个线程)
“A race car”

示例 1(续)



▶ 任务可按任意顺序执行

```
#include<stdlib.h>
#include<stdio.h>
int main(int argc, char *argv[])
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            #pragma omp task
            {printf("race "); }
            #pragma omp task
            {printf("car "); }
        }
    }
    printf("\n");
    return 0;
}
```

输出:
“A race car” or
“A car race”

示例 1 扩展



- ▶ 以 “is fun to watch” 结束句子

```
#include<stdlib.h>
#include<stdio.h>
int main(int argc, char *argv[])
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            #pragma omp task
            {printf("race "); }
            #pragma omp task
            {printf("car "); }
            printf("is fun to watch ");
        }
    }
    printf("\n");
    return 0;
}
```

任务在任务执行点执行！

输出:

“A is fun to watch race car” or
“A is fun to watch car race”

示例 1 扩展(续)



```
#include<stdlib.h>
#include<stdio.h>
int main(int argc, char *argv[])
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            #pragma omp task
            {printf("race "); }
            #pragma omp task
            {printf("car "); }
            #pragma omp taskwait
            printf("is fun to watch ");
        }
    }
    printf("\n");
    return 0;
}
```

输出:

“A car race is fun to watch” or
“A race car is fun to watch”

示例 2 : Fibonacci



```
#include<stdlib.h>
#include<stdio.h>
int fibo(int n) {
    if (n < 2) return n;
    return fibo(n-1)+fibo(n-2);
}
int main(int argc, char *argv[])
{
    fibo(input);
}
```

示例 2 : Fibonacci(续)



▶ 带任务分配功能的第一个版本 (omp-v1)

```
int main(int argc, char *argv[])
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            fibo(input);
        }
    }
}
```

```
int fibo(int n) {
    if (n < 2) return n;
    int x, y;
    #pragma omp task shared(x)
    {
        x = fibo(n-1);
    }
    #pragma omp task shared(y)
    {
        y = fibo(n-2);
    }
    #pragma omp taskwait
    return x+y;
}
```

✎ 只有一个线程从 main() 进入 fibo(), 负责创建两个初始工作任务

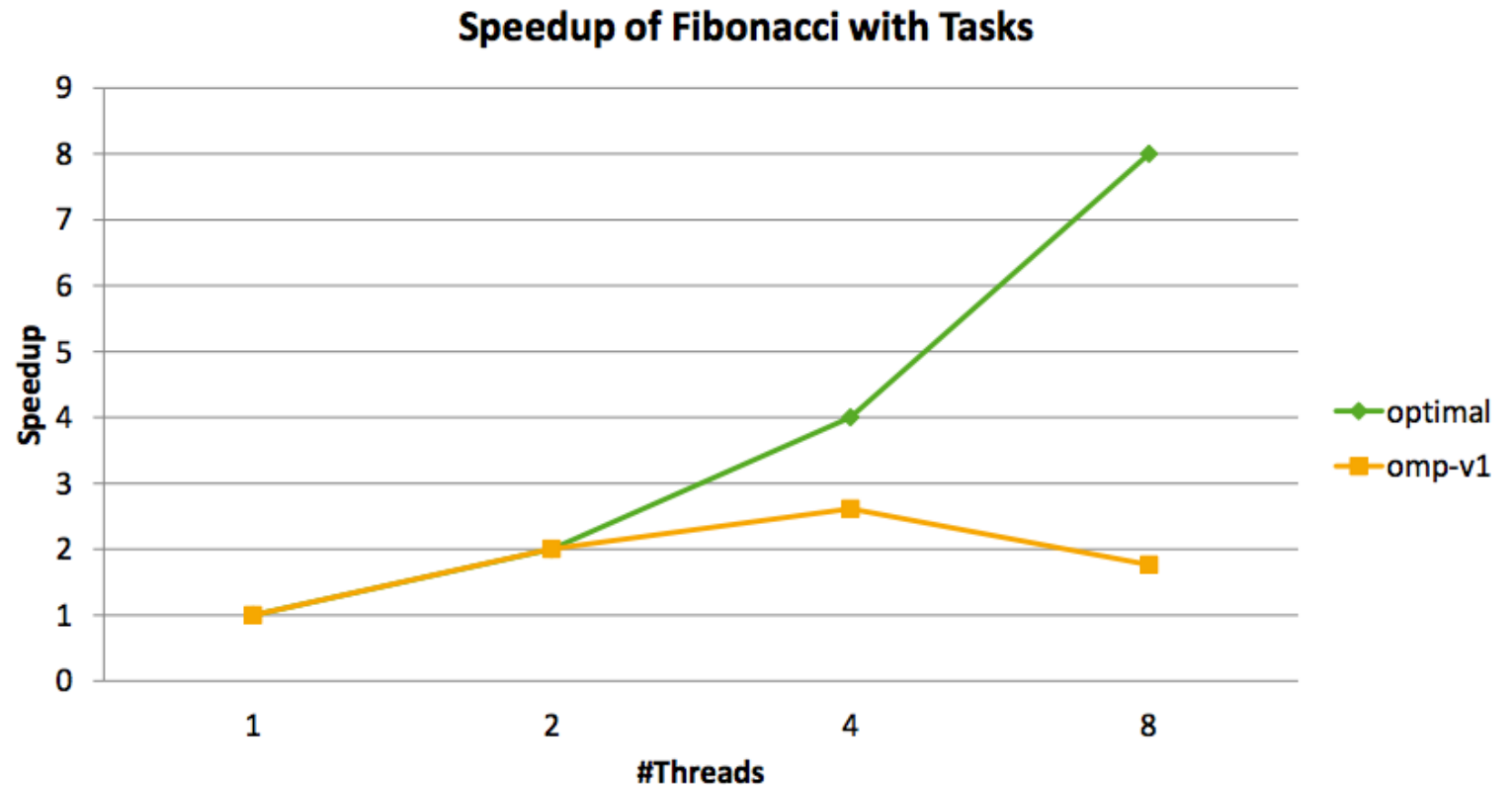
✎ 需要Taskwait, 否则 x 和 y 将丢失

示例 2 : Fibonacci(续)

- ▶ 任务创建的开销阻碍了更好的可扩展性！

```
int fibo(int n) {  
    if (n < 2) return n;  
    int x, y;  
    #pragma omp task shared(x)  
    {  
        x = fibo(n-1);  
    }  
    #pragma omp task shared(y)  
    {  
        y = fibo(n-2);  
    }  
    #pragma omp taskwait  
    return x+y;  
}
```

omp-v1



示例 2 : Fibonacci(续)

- ▶ 带任务分配功能的第一个版本 (omp-v2)

```
int main(int argc, char *argv[])
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            fibo(input);
        }
    }
}
```

```
int fibo(int n) {
    if (n < 2) return n;
    int x, y;
    #pragma omp task shared(x) \
        if (n > 30)
    {
        x = fibo(n-1);
    }
    #pragma omp task shared(y) \
        if (n > 30)
    {
        y = fibo(n-2);
    }
    #pragma omp taskwait
    return x+y;
}
```

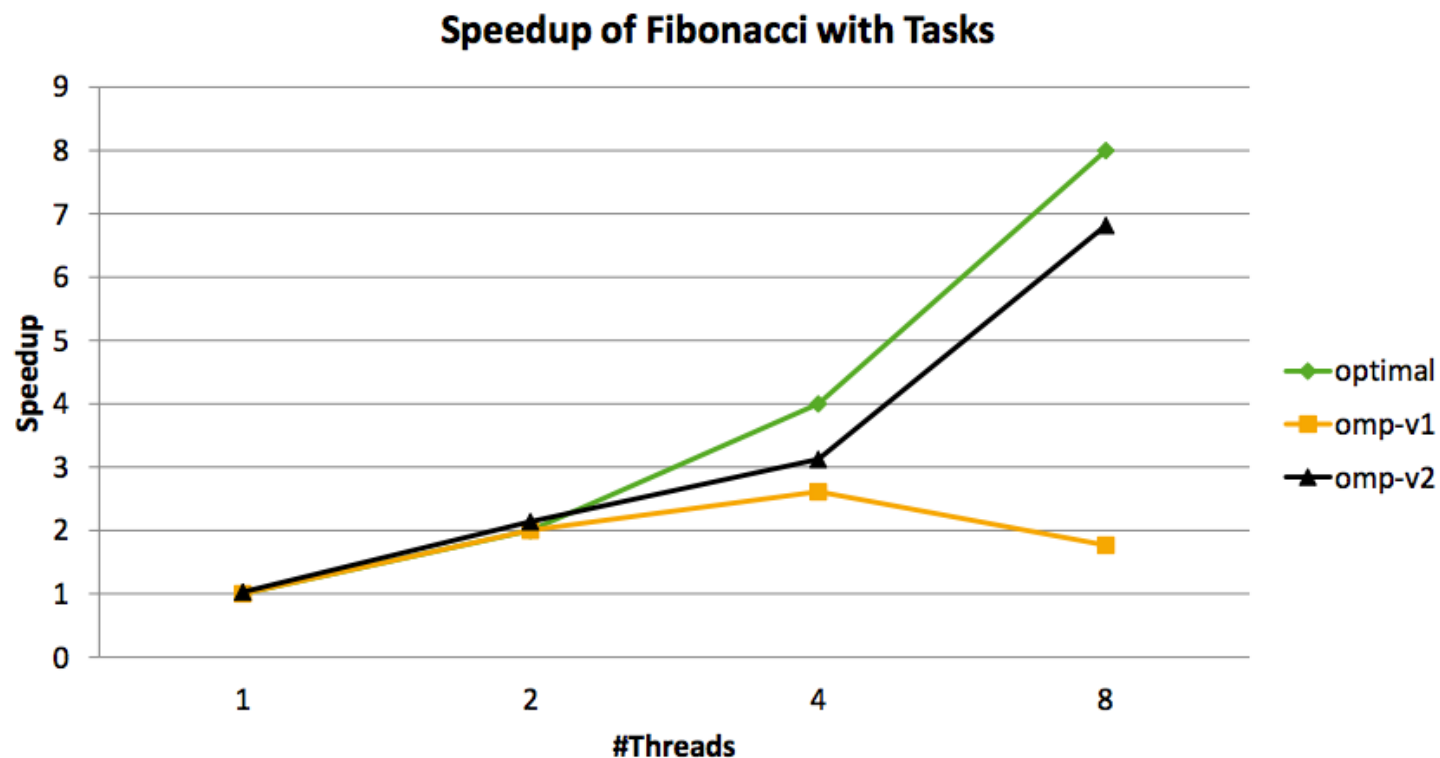
示例 2 : Fibonacci(续)



- 速度提升尚可，但使用 4 或 8 个线程运行时仍有一些开销

```
int fibo(int n) {  
    if (n < 2) return n;  
    int x, y;  
    #pragma omp task shared(x) \  
        if (n > 30)  
    {  
        x = fibo(n-1);  
    }  
    #pragma omp task shared(y) \  
        if (n > 30)  
    {  
        y = fibo(n-2);  
    }  
    #pragma omp taskwait  
    return x+y;  
}
```

omp-v2



示例 2 : Fibonacci(续)

- ▶ 帶任务分配功能的第一个版本（omp-v3）

```
int main(int argc, char *argv[])
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            fibo(input);
        }
    }
}
```

```
int fibo(int n) {
    if (n < 2) return n;
    if (n <= 30) return serial_fibo(n);
    int x, y;
    #pragma omp task shared(x) \
        if (n > 30)
    {
        x = fibo(n-1);
    }
    #pragma omp task shared(y) \
        if (n > 30)
    {
        y = fibo(n-2);
    }
    #pragma omp taskwait
    return x+y;
}
```

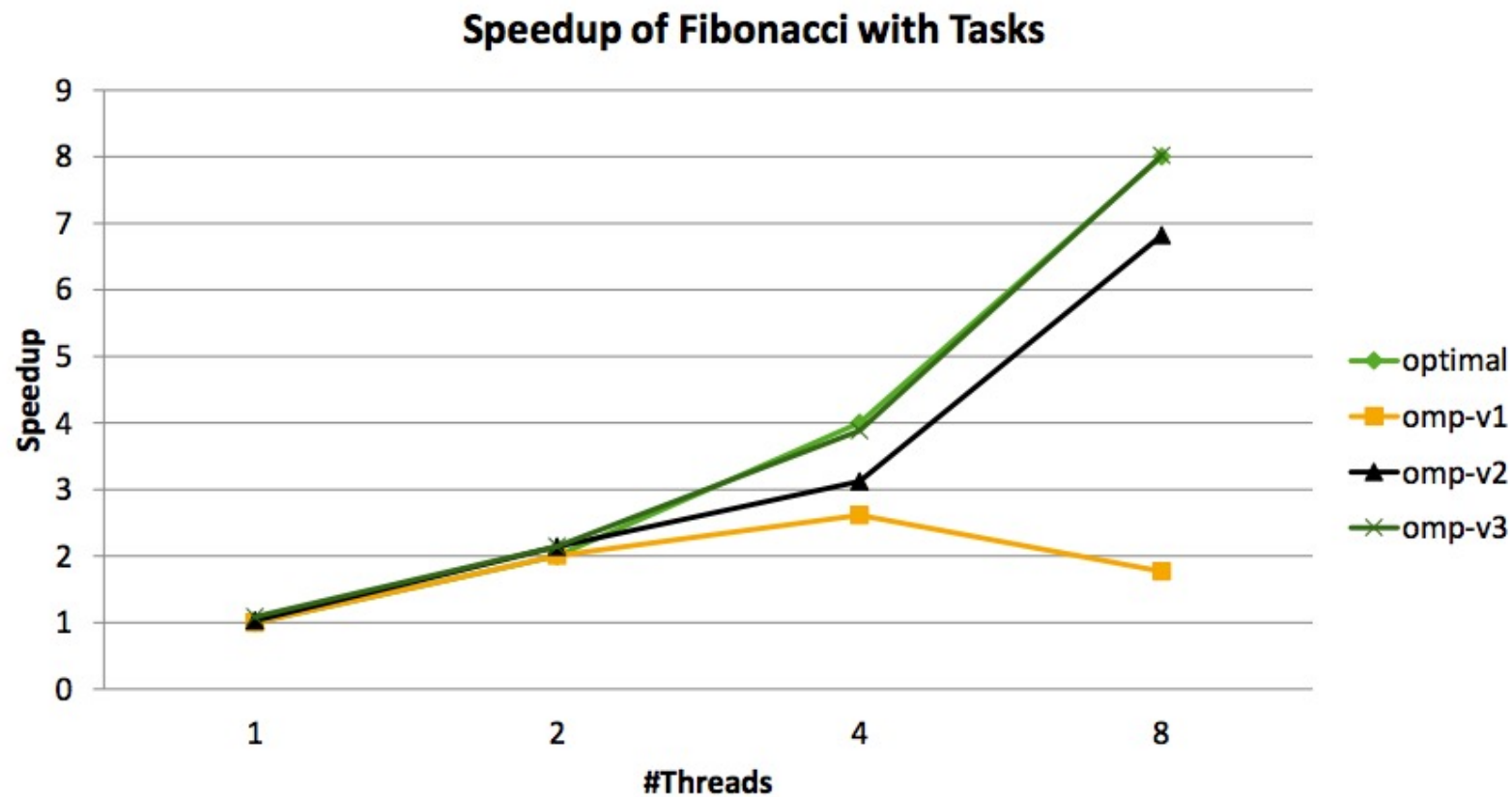
示例 2 : Fibonacci(续)



▶ 看起来很完美！

```
int fibo(int n) {  
    if (n < 2) return n;  
    if (n <= 30) return serial_fibo(n);  
    int x, y;  
    #pragma omp task shared(x) \  
        if (n > 30)  
    {  
        x = fibo(n-1);  
    }  
    #pragma omp task shared(y) \  
        if (n > 30)  
    {  
        y = fibo(n-2);  
    }  
    #pragma omp taskwait  
    return x+y;  
}
```

omp-v3



示例3：链表



▶ 支持任务分配功能前难以完成

✎ 首先计算迭代次数，然后将 while 循环转换为 for 循环

▶ 使用任务实现简单

✎ 使用单一结构：一个线程生成任务

✎ 所有其他线程在任务可用时执行任务

```
...  
my_pointer = listhead;  
while(my_pointer) {  
    do_independent_work(my_pointer);  
    my_pointer = my_pointer->next;  
} // End of while loop  
...
```

示例3：链表(续)



```
...  
my_pointer = listhead;  
#pragma omp parallel  
{  
    #pragma omp single  
    {  
        while(my_pointer) {  
            #pragma omp task firstprivate(my_pointer)  
            {do_independent_work(my_pointer); }  
            my_pointer = my_pointer->next;  
        } // End of while loop  
    }  
}  
...
```

▶ 支持嵌套并行

✎更好地定义和控制嵌套并行区域，使用新的应用程序接口确定嵌套结构

▶ 增强循环调度

✎支持更激进的编译器优化和更好的运行时控制

▶ 循环折叠(loop collapse)

✎支持组合嵌套循环以实现更多并发性

▶ 任务分配

✎支持任务级别的并行化，支持复杂和动态的控制流