第 3 章 Chapter 3

OpenMP 程序设计

OpenMP（Open Multi-Processing，开放多处理）是一种支持多平台共享存储器多处理器编程的 C/C++ 和 Fortran 语言的规范和 API。OpenMP 支持许多体系结构的处理器和操作系统（OpenMP 独立于处理器架构和操作系统，ICC、GCC 都给予了很好的支持，但是 VC 的支持比较差）。

目前支持 OpenMP 的编译器包括 GCC、ICC 和 VC 等。VC++ 从 2005 版开始支持 OpenMP 2.0 的多核编程（为了推销微软自己的并行技术，VC++ 2008 和 2010 都只支持 OpenMP 2.0，并没有支持 OpenMP 2.5 及更新的标准）。GCC 4.2 及以上版本默认支持 OpenMP，而且紧跟标准的脚步，GCC 4.9 已经开始支持最新的 OpenMP 4.0 标准。使用 GCC 编译 OpenMP 程序时，只需要在编译命令中加上 -fopenmp 选项即可。

OpenMP API 包括以下几个部分：一套编译器伪指令；一套运行时函数；一些环境变量。OpenMP 已经被大多数计算机硬件和软件厂商所接受，成为事实上的标准。

OpenMP 是一种基于隐式共享存储器的编程环境。它提供了对并行算法的高层的抽象描述，程序员通过在源代码中插入各种 pragma 伪指令（directive，指示 / 命令，软件开发人员通过这种标识告诉编译器对相应的代码段做处理）来指明自己的意图，编译器据此可以自动将程序并行化，并在必要之处加入同步互斥等通信手段。当选择告诉编译器忽略这些 pragma，或者编译器不支持 OpenMP 时，程序又可退化为串行程序，代码仍然可以正常运作，只是不能利用多线程来加速程序执行。

OpenMP 提供的这种对于并行描述的高层抽象，降低了并行编程的难度和复杂度，这样程序员可以把更多的精力投入到并行算法本身，而非其具体实现细节。对基于数据并行的多

线程程序设计而言，OpenMP 是一个很好的选择。同时，使用 OpenMP 也提供了更强的灵活性，可以较容易地适应不同的并行系统配置。线程粒度和负载均衡等是传统并行程序设计中的难题，但在 OpenMP 中，OpenMP 库从程序员手中部分接管了这两方面的工作。

OpenMP 的设计目标为：标准、简洁实用、使用方便、可移植。作为高层抽象，OpenMP 并不适合需要复杂的线程间同步、互斥及对线程做精密控制的场合。OpenMP 的另一个缺点是不能很好地在非共享内存系统（如计算机集群）上使用，在这样的系统上，MPI 更适合。

3.1 OpenMP 编程模型

OpenMP 支持数据并行，也支持任务并行，它的并行模型采用传统的 fork-join。fork 是叉子的意思，叉子只有一个柄却有很多齿，这意味着由一个主线程建立（fork）许多子线程，在一定的时候，主线程再收回（join）所有子线程。

OpenMP 的编程模型描述了其实现的 fork-join 模型的功能和限制，主要包括 OpenMP 程序如何在处理器上执行（执行模型）和 OpenMP 程序如何使用共享存储器通信（存储器模型）。

3.1.1 OpenMP 执行模型

OpenMP 支持共享存储器并行程序设计。和 MPI 支持分布式存储并行程序设计不同，共享存储器是指在这种模式下，并行执行的控制流之间共享存储器，也就是说系统只存在一个内存（既可以是物理上的，也可以是逻辑虚拟的，比如底层使用消息传递使多个分布式的存储表现出共享存储的行为）。分布式存储意味着存储器分布在不同的地方，需要显式地在各个存储器之间传输数据。软件开发人员使用 OpenMP 时，通过显式的伪指令或函数来指示编译器并行，但是 OpenMP 标准并不要求编译器检查数据和控制依赖关系、读写冲突、竞争和死锁（这对编译器来说太难了），因此软件开发人员必须自己处理这些。

OpenMP 线程的执行采用 fork-join 方式，这和 pthread 相同。不同的是在并行域的开始使用伪指令建立多个线程（fork），在并行域的结束自动回收线程（join）。

OpenMP 程序开始执行时，也是单线程执行的，这个线程称为初始线程或主线程。在 parallel 伪指令调用处，主线程产生多个子线程，此后主线程和子线程共同执行 parallel 块内代码，在块结束处，子线程会被回收，只有主线程依旧向前执行。

OpenMP 不但支持并行编程，还支持串行编程，通常这只需要处理编译器选项就可以从 OpenMP 并行程序产生串行程序，当然有些程序是不能这样做的，比如对线程索引进行了显式编码的情况。也有可能串行编译时和并行编译时产生不同的结果。

比较新的 OpenMP 标准支持嵌套（nest）线程，此时嵌套的 parallel 指令产生的外层线程在遇上内层 parallel 伪指令的地方会产生多个线程。如果编译器不支持 nest 线程，则内层

parallel 会被忽略。**为了编写可移植的代码，不建议使用嵌套的 parallel 伪指令。**

OpenMP 3.0 或更新的标准通过 task 构造更好的支持任务并行、任务依赖的概念，如果线程执行到 task 构造，会分配一个已有的线程去执行 task 构造块，因此 task 必须在 parallel 构造中。老标准中可以通过 section 伪指令达到 task 构造同样的效果。

OpenMP 提供了在 parallel 块内同步和协调线程执行的机制，另外还提供了查询和控制 OpenMP 运行环境的函数和环境变量。

在多个线程并行地读写同一个文件时，OpenMP 并不保证读写顺序。软件开发人员需要自己处理这种情况，以保存每个线程都读到正确的数据。如果每个线程读写不同的文件，则不需要顺序保证。

OpenMP 4.0 加入了 target 构造来支持异构并行计算的内容，如果主线程执行到 target 构造，便会在加速器上映射相关的数据结构，并将 target 代码块交给关联的加速器执行。

OpenMP 预定义了宏 _OPENMP 以支持条件编译。通常如果不加对应的编译器选项，编译器便会忽略 OpenMP 伪指令。宏 _OPENMP 进一步支持了一份代码既支持编译出并行程序又支持编译出串行程序的机制。

大多数 OpenMP 实现环境默认使用的线程数等于计算机的核心数量，因为对于大多数数据并行的计算密集型代码而言，这通常是最优的。不过也有一些 OpenMP 实现的默认线程数量为 1，为了能在不同的编译器之间可移植，软件开发人员还是应当显式地指明算法使用的线程数量。

3.1.2 OpenMP 存储器模型

OpenMP 支持弱存储器一致性的共享存储器模型，所有的 OpenMP 线程都可以访问同一个存储器空间。另外，每个 OpenMP 线程可以拥有自己的栈空间。弱存储器一致性是说同一时刻每个 OpenMP 线程看到的存储器内容（内存、缓存、寄存器等）可能是不一样的，这能够给编译器更多的优化可能，且简化了编译器设计。

许多 OpenMP 语句都接收两个决定其关联的块内变量的访问权限的变量：shared 和 private，同时还有一个 default 子句。default 决定了没有显式声明访问权限的变量的权限，其默认值为 shared。每个 shared 变量刚进入并行区域时，其值为进入并行区域前的值。每个线程访问 shared 限制的变量都会访问原始的变量。而线程会为 private 声明的变量建立一个副本。通常程序编写得好的话，无须使用 private，因为可通过声明线程栈内变量代替，而且可读性更好。

每次存储器访问的最小数据量是由硬件决定的（通常是一个或多个缓存线，需要注意缓存线的大小），软件开发人员需要注意这个值，因为它可能会引起伪共享问题。

OpenMP 存储器模型不对存储器操作做原子性保证，因此软件开发人员要利用 OpenMP

提供的工具编码处理。如果多个线程读写同一个变量但是没有同步，就会发生冲突，此时计算的结果没有定义。

如果主线程遇到 target data 块，便会依据其子句指定的语义，在加速器上映射相关的数据结构（包括在进入代码块时，在加速器上创建存储器和把数据从主机端复制到加速器上；退出代码块时，把数据从加速器复制回主机端，并销毁分配的数据空间）。

flush 操作是一种存储器屏障，它保证了线程的存储器内容是最新的，同时保证编译器不会将其前面的指令重排到后面，也不会将其后面的指令重排到前面。flush 是一个容易被误用的子句，其特点类似于 C 语言中的 volatile 关键字。很多开发人员误认为 flush 能够保证操作的结果被其他线程看到，这并没有错；问题在于 flush 并没有对线程的执行顺序做任何限制。比如当一个线程执行 flush 的时候，另外一个线程可能早就执行了其他语句，而修改了相关存储器空间的值。

3.2 环境变量

OpenMP 环境变量指定了某些影响 OpenMP 运行时行为的配置，能够影响到所有使用这台机器的 OpenMP 程序。OpenMP 环境变量的优先级低于函数和伪指令。由于每台机器上的配置可能都不一样，因此实际编程中使用较少，故本节不打算深入。仅简单罗列如下：

- OMP_SCHEDULE 指定了运行时负载均衡类型和每次任务分配的循环次数。
- OMP_NUM_THREADS 指定了执行并行区域时使用的默认线程数量。
- OMP_DYNAMIC 决定了是否允许调整执行并行区域的线程数量，如果设置为 true，则 OpenMP 运行时可能会调整执行并行区域的线程数量，以优化系统资源的使用。
- OMP_PROC_BIND 决定了是否允许线程迁移到其他的处理器上执行。如果为 true，则运行时不会在处理器间迁移线程。
- OMP_NESTED 决定了是否支持嵌套（nest）线程，如果为 true，则支持。
- OMP_STACKSIZE 指定了每个线程拥有的栈大小，如果出现栈溢出，应该将其改大。
- OMP_THREAD_LIMIT 指定了系统能够创建的 OpenMP 线程的最大值。

3.3 函数

OpenMP 提供了一些函数以支持显式的多线程编程。和伪指令不同的是：这些函数可能会影响后面的所有 OpenMP 语句。由于 OpenMP 函数肯定会被编译且需要链接对应的库，故相比伪指令，OpenMP 函数可能会对代码支持编译成串行程序产生影响。开发人员可以从逻辑上，也可以使用宏 _OPENMP 来保证代码也支持编译成串行程序。虽然在实际使用中 OpenMP

函数比伪指令灵活，但是笔者建议编程实践中能不用则不用。

3.3.1 普通函数

函数 omp_get_num_threads 返回执行当前并行区域内运行的总线程数量，而函数 omp_set_num_threads 设置线程数，它们的原型如下：

```
int omp_get_num_threads(void);
void omp_set_num_threads(int num);
```

omp_set_num_threads 会影响后面所有没有使用 num_threads 显式指定线程数量的 OpenMP 语句（如果后面的代码中没有使用 OpenMP 语句显式指明线程数量，那么就继承此值）。在 GCC 中，OpenMP 实现默认的线程数量就是多核处理器的核心数量，这是一个很好的特性，而 PGI 的 C 编译器 pgcc 的默认值却是 1。为了保证程序的性能和正确性，笔者建议在编译时显式指定或在运行前设定环境变量 OMP_NUM_THREADS。如果要使用 omp_set_num_threads 设置线程数目的话，omp_set_num_threads 函数必须位于对应的 #pragma 前面。

函数 omp_get_max_threads 获得当前能够使用的线程数上限，其原型如下：

```
int omp_get_max_threads(void);
```

函数 omp_get_thread_num 获得线程的索引，其原型如下：

```
int omp_get_thread_num(void);
```

其中，主线程索引为 0，如果并行构造中指定的线程数量为 N，则 omp_get_thread_num 函数为各个线程的返回值为 [0，N−1]。omp_get_thread_num 和 omp_get_num_threads 函数是 OpenMP 部分支持显式多线程编程的主要理由（所说的“部分”是指：在 OpenMP 中，要创建线程，就必须使用 parallel 构造）。

函数 omp_get_num_procs 返回程序可用的处理器数目，其原型如下：

```
int omp_get_num_procs(void);
```

函数 omp_in_parallel 判断当前代码是否在并行区域内，其原型如下：

```
int omp_in_parallel(void);
```

函数 omp_get_thread_limit(void) 返回程序能够使用的线程数目上限，其原型如下：

```
int omp_get_thread_limit(void);
```

函数 omp_set_dynamic 设置是否支持动态调整并行区域的执行线程数，而函数 omp_get_dynamic 返回是否支持动态调整并行区域的执行线程数，其原型如下：

```
void omp_set_dynamic(int);
int omp_get_dynamic(void);
```

函数 omp_set_nested 设置是否支持 nest 线程，而函数 omp_get_nested() 返回是否支持 nest 线程，其原型如下：

```
void omp_set_nested(int);
int omp_get_nested();
```

函数 omp_set_schedule 设置负载均衡策略，omp_get_schedule 获取负载均衡策略，其原型如下：

```
void omp_set_schedule(omp_sched_t kind, int modifier);
void omp_get_schedule(omp_sched_t *kind, int * modifier);
typedef enum omp_sched_t {
    omp_sched_static = 1,
    omp_sched_dynamic = 2,
    omp_sched_guided = 3,
    omp_sched_auto = 4
} omp_sched_t;
```

其中 modifier 指分发的块大小。

整体来说，在使用 OpenMP 进行多线程编程时，OpenMP 函数是完全可以不使用的，因为几乎每一个函数都有对应的编译制导语句。而锁函数是一个例外。

3.3.2 锁函数

OpenMP 提供了锁机制，基本功能和 pthread 排它锁一样，其类型为 omp_lock_t。在 OpenMP 提供的通信手段中，除了原子函数之外，锁的损耗是最小的。

函数 omp_init_lock 初始化锁；omp_set_lock 获得锁；omp_unset_lock 释放锁；omp_destroyed_lock 销毁锁，它们的原型如下：

```
void omp_init_lock(omp_lock_t *);
void omp_set_lock(omp_lock_t *);
void omp_unset_lock(omp_lock_t *);
void omp_destroyed_lock(omp_lock_t *);
```

除了函数名字的差别，其功能和 pthread 中对应函数一致。

OpenMP 的编译制导语句支持临界区（critical），和锁不同的是：临界区只能锁定一段代码，即这一段代码只允许线程串行执行；而锁能够锁定多段代码，如果有线程执行了被锁定的多段代码中的一段，那其他被同一个锁锁定的代码也只能串行执行。

代码代码清单 3-1 展示了如何在 OpenMP 中使用锁。需要提醒读者注意的是：初始化锁和销毁锁的工作都要由主线程在进入并行区域前调用。

代码清单3-1　在OpenMP中使用锁

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <omp.h>

int main(int argc, char *argv[]){
int x = 3;
int y = 4;

omp_lock_t lock;
omp_init_lock(&lock);
#pragma omp parallel num_threads(3) shared(x, y)
{
     omp_set_lock(&lock);

    x += omp_get_thread_num();
    y += omp_get_thread_num();

    omp_unset_lock(&lock);
}
    printf("x = %d, y = %d\n", x, y);

    omp_destroy_lock(&lock);

    return 0;
}
```

示例代码把每个线程的索引编号加到所有线程共享的变量 x 和 y 上，如果不使用锁的话，那么有些线程的加操作结果会丢失。

3.4　OpenMP 编译制导语句

在 OpenMP 支持的 C/C++ 语言中，一条 OpenMP 编译制导语句的格式如下：

```
#pragma omp directive [clause[clause] ... ]
{
    code;
}
```

其中，中括号表示可选，OpenMP 不允许将大括号放到 #pragma 语句的后面，左大括号和右大括号必须各占一行。

对于 directive，不同的人的翻译有所不同，主要有：伪指令、构造、导语等。本章主要采用伪指令或构造及子句。

3.4.1　常用的 OpenMP 构造

许多伪指令或子句结束时都有隐含的栅栏同步，只有并行区域内所有线程都执行到此处后才允许并行区域内的线程接着执行，否则必须等待其他线程到达。在某些情况下，隐含的

栅栏同步是没有必要的，因此 OpenMP 提供了 nowait 子句以去掉它。

构造用来创建或分发并行区域，其所包含的区域由多个线程执行；而子句对构造进行补充说明。本节详细的介绍各构造的使用、语义，3.4.2 节介绍子句，同时给出许多便于理解的例子。

OpenMP 常用于循环并行化，通过循环并行化，编译指导语句使得一段代码能够在多个线程内部同时执行。循环并行化需要寻找程序代码中最耗时的循环，并将其分解到多个线程中去。在 OpenMP 中，parallel for 承担了循环并行化构造这一角色。

parallel for 构造由 parallel 构造和 for 构造共同组成，parallel 表示产生多个线程执行同一段代码，而 for 表示将循环计算任务划分到当前并行区域中的线程中。

1. Parallel

parallel 用来构造一个并行区域，表示这段代码将被多个线程并行执行，也可以使用分支指定不同的线程执行区域内不同的代码，可以和其他子句如 for、simd、sections 等配合使用，以方便地实现某些功能。需要多线程执行的代码应放到 parallel 里面，另外 parallel 结束处有默认的同步。由于 parallel 结束意味着并行区域的终结，故不支持 nowait 子句。

请大家执行代码清单 3-2 的 hello world 代码，并观看结果。其中 num_threads 指定执行并行区域的线程数。

代码清单3-2　OpenMP parallel 版Hello World

```
#include <omp.h>
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[]){
#pragma omp parallel num_threads(3)
{
    for(int i = 0; i < 3; i++)
    printf("Hello World, i am %d, iter %d\n", omp_get_thread_num(), i);
}

    return 0;
}
```

下面是在笔者的机器上某次执行的输出结果。

```
Hello World, i am 0, iter 0
Hello World, i am 0, iter 1
Hello World, i am 0, iter 2
Hello World, i am 1, iter 0
Hello World, i am 1, iter 1
Hello World, i am 1, iter 2
Hello World, i am 2, iter 0
Hello World, i am 2, iter 1
Hello World, i am 2, iter 2
```

从输出可以看出，一个有 3 个线程，每次线程执行了 3 次 for 循环，即产生了多个线程，每个线程都把 parallel 构造块里面的语句执行了一遍。

2. for

for 子句则将 for 循环分发到多个线程中执行，因此其必须在 parallel 区域内。for 循环的写法必须满足一定条件，且 OpenMP 标准没有强制编译器检查依赖关系，故软件开发人员最好自己人为保证。为了简化 for 的使用，OpenMP 提供了 parallel for，但是在有些情况下，分开使用 parallel 和 for 可能效率更高。

OpenMP 的循环并行化是指通过 parallel for 指导语句将 C/C++ 的 for 循环并行化，即将循环中的迭代分配给线程组中的各个线程分别执行，parallel for 可能是大多数 OpenMP 中使用得最多的构造。

由于某些 for 循环无法并行或编译器还不能并行，因此 OpenMP 对 for 伪指令操作的 for 循环做了如下一些限制，本节以 for(初值 ; 测试表达式 ; 增量表达式 ; 终值) 这种表示为例。

1）必须是 for 循环，且其必须具有规范的格式，能够推测出循环次数。

2）索引必须为整数类型。

3）测试表达式必须具有如下形式：索引 ; 比较运算符 ; 终值。

- 比较运算符可以是 <、<=、>=、>。
- 增量表达式的形式必须为：i++、++i、i--、--i、i+=c、i-= c、i=i+c、i=i-c。
- 初值、增量和终值都可以是任意数值表达式，但是在循环过程中值必须保持不变，以保证在循环前就能计算出循环的次数。
- 循环语句块是单出口与单入口的。因此，不能使用 break 语句，也不能用 goto、return 等语句从循环中跳出。C++ 代码不能在循环内部抛出异常，因为会导致程序从循环中退出。但是可以在循环体内使用 exit() 函数退出整个程序。当某一个线程调用此函数后，会同步其他所有线程来退出程序，不过退出时的状态是不确定的。

4）parallel for 编译指导语句可以加在任意一个循环（包括嵌套的循环）之前，则在其之后最近的循环语句被并行化。

对 for 构造的限制使用一句话总结就是：**在建立线程时要知道循环次数且不能让其他线程不知道某线程要离开循环体**。

默认情况下，for 伪指令块结束会有隐式的同步，如果不需要这个，可以使用 nowait。通常使用 nowait 能够提升性能。

请读者执行示例代码清单 3-3 的 Hello World 代码，观看结果，并与代码清单 3-2 比较。

代码清单3-3　OpenMP parallel for版Hello World

```
#include <omp.h>
#include <stdlib.h>
```

```
#include <stdio.h>
int main(int argc, char *argv[]){
#pragma omp parallel for num_threads(3)
    for(int i = 0; i < 3; i++)
    printf("Hello World, i am %d, iter %d\n", omp_get_thread_num(), i);

system("pause");
    return 0;
}
```

下面是在笔者的机器上某次执行的输出结果。

```
Hello World, i am 0, iter 0
Hello World, i am 2, iter 2
Hello World, i am 1, iter 1
```

从输出可以看出，3 个线程共同执行 for 循环，即 for 循环被多个线程协作执行了。而代码清单 3-2 的代码则是每个线程都执行了 3 次循环。

parallel 和 parallel for 相比，只是指令名少一个 for 关键字，但是它们的行为则相差很大。parallel 采用了复制执行的方式，将代码在所有的线程内各执行一次；parallel for 则是采用工作分配执行方式，将循环需做的所有工作量，按一定的方式分配给各个执行线程，全部线程执行工作量的总和等于原先串行执行所完成的工作量。如果开发人员在应当使用 parallel for 的地方只使用了 parallel，那么所有的线程都会执行同样的代码，这可能会导致程序的性能反而不如串行代码。

在某些情况下，可能需要更好的控制线程的行为，此时可以使用 parallel 构造，然后依据线程索引来分配计算任务。

3. simd

simd 构造是 OpenMP 4.0 标准新增加的，它表示每个线程将使用 simd 指令来执行紧随其后的循环。由于现代的 CPU 都有 SIMD 执行单元，故应尽量优先使用。

simd 可以和 parallel、parallel for 联用，也可单独使用。

simd 有几个子句，其中 safelen(x) 表示紧随其后的循环中，每 x 次循环内都是互不相关的；aligned(list:n) 表示数组 list 对齐尺寸为 n 个字节。

代码清单 3-4 是使用 OpenMP simd 指令向量化求长向量平方和的示例。由于目前支持 OpenMP 4.0 simd 的编译器不多，故代码并未经过实际测试。

代码清单3-4　OpenMP simd 求向量平方和

```
#pragma omp parallel for simd reduction(+:ret)
for(int i = 0; i < num; i++){
   ret += a[i]*a[i];
}
```

parallel for 使用多个线程并行处理循环，simd 伪指令让每个线程使用向量化指令来加速循环。由于需要把每个线程向量的元素加和才能得到最终需要的结果，故使用了 reduction。

4. task

task 主要用于方便地实现任务分解，只在 3.0 及以上的 OpenMP 标准实现中它才得到支持，而在之前标准的实现中可以通过 sections/section 实现。

主线程遇到 task 就会分配一个线程来执行它们，因此它必须在 parallel 块中；考虑到 parallel 的特性，这又要求在 task 外面使用 single 子句。如果需要父线程要等待调用 task 的子线程结束，可以使用 taskwait 子句，详见下面第 5 点。

以递归法计算 Fibonacci 数为例展示 task 的使用，如代码清单 3-5 所示。

代码清单3-5 使用task计算Fibonacci数

```
#include <omp.h>
#include <stdio.h>

int facobi(int num){
   if((0 == num) || (1 == num)){
      return 1;
    }

    int f1, f2;
#pragma omp task shared(f1)
    f1 = facobi(num-1);
#pragma omp task shared(f2)
    f2 = facobi(num-2);
#pragma omp taskwait
    return f1+f2;
}
int main(int argc, char *argv[]){
    int r;
#pragma omp parallel shared(r)
{
#pragma omp single
    r = facobi(5);
}

   printf("%d\n", r);
  return 0;
}
```

读者可自己加入一些代码来确认代码的确被并行化了。这段代码利用了 OpenMP 3.0 标准的另一个重要特性 taskwait。

taskwait 会等待前面使用 task 启动的计算结束，如代码清单 3-5 所示，#pragma omp taskwait 会等待前面的两个 #pragma omp task（计算 f1 和 f2）完成，因为只有计算完成后才能

计算两者之和。

5. taskwait

构造 taskwait 用于 task 构造之后，用于等待前面的所有（一个或多个）task 执行完成。构造 taskwait 存在一个缺陷，那就是无法等待前面启动的多个 task 中的某几个结束，也许以后的版本中有这个特性。

6. taskyield

线程执行到 taskyield 构造时，会释放占用的 CPU 及某些资源并暂停执行，以便其他线程有机会占用 CPU。

taskyield 不会对 OpenMP 代码的功能产生影响，但是它会影响某些应用的性能。如在应用要等待某些条件发生才能接着执行的时候，线程就应当调用 #pragma omp taskyield 放弃 CPU 和其他资源，以便其他线程有机会占用 CPU 执行。

7. sections/section

sections/section 构造用于将 sections 里的代码划分成几个不同的段，每段由一个线程执行，各线程工作量的总和等于原来的工作量。在 OpenMP 标准 3.0 以前的实现中，它们被用来实现任务并行（OpenMP 3.0 中的构造 task）。section 结束时隐含有栅栏同步，可以使用 nowait 子句除去。

代码清单 3-6 使用 parallel sections 和 section 实现了两个任务同时执行。

代码清单3-6　OpenMP sections/section示例

```
#include <omp.h>
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[]){
#pragma omp parallel sections
{
#pragma omp section
{
    printf("I am thread %d, execute sections one\n", omp_get_thread_num());
}
#pragma omp section
{
    printf("I am thread %d, execute sections two\n", omp_get_thread_num());
}
}
    return 0;
}
```

使用任务并行时，负载均衡非常容易成为问题，通常这可由启动大量粒度细一些的并行任务来解决。

8. single

构造 single 指定相关的并行区域只由一个线程执行，至于到底是哪个线程执行，由运行时的具体情况决定。通常使用 single 构造表示需要对某些代码做特殊处理，可能是基于这种考虑，single 结束处有隐式的栅栏同步。代码清单 3-7 通过打印语句展示了 single 构造的行为。

代码清单3-7 OpenMP single示例

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main(int argc, char *argv[]){
#pragma omp parallel num_threads(3)
{
    printf("i am %d in line %d\n", omp_get_thread_num(), __LINE__);
#pragma omp single
{
    printf("i am %d in line %d\n", omp_get_thread_num(), __LINE__);
}
    printf("i am %d in line %d\n", omp_get_thread_num(), __LINE__);
}
    return 0;
}
```

查看结果会发现，single 构造包含的行只被打印了一次，而且多次运行发现，执行该行的线程并不固定。

9. master

构造 master 指定相关的并行区域由主线程（即 0 号线程）执行，有这种要求的情况非常少，通常 single 是更好的选择，故笔者不建议使用它。当然在基于主从模式的应用中，可能用它来做某些负载均衡等处理。

和 single 不同的是，master 结束处没有隐式栅栏。

代码清单 3-8 展示了 master 构造的行为，读者多次运行将会发现：

- 每次运行 #pragma omp master 语句的线程都是 0 号线程（即主线程）执行。
- master 构造包含的行只被打印了一次。

代码清单3-8 OpenMP master示例

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main(int argc, char *argv[]){
#pragma omp parallel num_threads(3)
```

```
{
    printf("i am %d in line %d\n", omp_get_thread_num(), __LINE__);
#pragma omp master
{
    printf("i am %d in line %d\n", omp_get_thread_num(), __LINE__);
}
    printf("i am %d in line %d\n", omp_get_thread_num(), __LINE__);
}
    return 0;
}
```

在实际项目中，应当避免使用 #pragma omp master，因为通常 #pragma omp single 会是更好的选择，当然调试程序时例外。

10. barrier

构造 barrier 的作用是显式调用栅栏同步。在其调用处，除非并行局域内的所有的线程已经到达，否则到达此处的线程必须等待并行区域内的其他线程到达，只有并行区域内的所有线程都到达调用处，线程才可以向下执行。

barrier 构造同时保证了可见性，即在所有线程都执行到 barrier 但是还没有往前执行的时刻，所有线程看到的共享存储器内容是一致的。

使用 barrier 要注意可能由此引起的死锁，主要设计缺陷为软件开发人员可以在某个线程内部代码使用了 barrier，此时无论如何也是等不到其他线程到达的。

如果代码中多个线程需要交换数据，那么就必须适时等待数据准备好。代码清单 3-9 是一个简单的多线程通过 barrier 交换数据的例子。

代码清单3-9　栅栏示例

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>

int main(int argc, char *argv[]){
int a[6];

#pragma omp parallel num_threads(3) shared(a)
{
   int id = omp_get_thread_num();
   a[id] = id;
   a[3+id] = 3+id;
#pragma omp barrier
   int swap = a[id];
   a[id] = a[5-id];
   a[5-id] = swap;
}
```

```
for(int i = 0; i < 6; i++)
printf("%d ", a[i]);

    return 0;
}
```

程序正确地输出了结果 5 4 3 2 1 0，这说明 barrier 构造同步了所有线程，否则结果和线程的执行顺序有关。

11. critical

为了保证在多线程执行的程序中，出现数据竞争时能够得到正确结果，OpenMP 提供了 3 种不同类型的线程同步机制：排它锁、原子操作和临界区。

构造 critical 声明一个临界区，临界区一次只允许一个线程执行，因此它提供了一种串行的机制。构造 critical 结束处没有隐式栅栏。

代码代码清单 3-10 展示了如何利用临界区来解决读写冲突问题。读者应当仔细理解，因为临界区是非常有用的同步机制。

代码清单3-10　临界区示例

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main(int argc, char *argv[]){
int x = 3;
int y = 4;
#pragma omp parallel num_threads(3) shared(x, y)
{
#pragma omp critical
{
    x += omp_get_thread_num();
    y += omp_get_thread_num();
}
}
    printf("x = %d, y = %d\n", x, y);

    return 0;
}
```

OpenMP 中的 critical 通常使用锁实现，关于 OpenMP 中锁和临界区的区别请参考 3.3.2 节。critical 使得并行区域内的线程串行地执行某个任务，故为性能考虑，应当尽量减少 critical 内部代码的数量。

12. atomic

构造 atomic 实现了原子操作，原子操作可以简单理解为除非一个线程操作完成，否则绝不允许其他线程操作。这类似于临界区和锁。由于现代处理器的限制，原子操作实现的功能

非常有限，但是其速度相比其他机制要快得多，因此要优先使用。

atomic 构造只能作用在 +、*、-、/、&、^、|、>> 和 << 运算符上，并且运算符不能是被 C++ 重载的函数。

atomic 构造具有子句 read、write、update 和 capture。read 表示原子读，只可用于读取操作；而 write 表示原子写，只可用于写操作；update 表示既有写又有读，默认即是这种情况；capture 也表示有读有写，和 update 不同的是，其具有返回值，如 y=x++、y=++x 等。

代码清单 3-11 展示了如何使用原子函数实现和临界区同样的功能。

代码清单3-11　原子函数示例

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main(int argc, char *argv[]){
int x = 3;
int y = 4;
#pragma omp parallel num_threads(3) shared(x, y)
{

#pragma omp atomic
    x += omp_get_thread_num();
#pragma omp atomic
    y += omp_get_thread_num();
}
    printf("x = %d, y = %d\n", x, y);

    return 0;
}
```

现代处理器对原子操作的种类做了限制，只有一些相当简单的操作得到支持，因此其适应面并没有临界区和锁广泛。由于原子操作的粒度非常细，其优势在于通常其性能比锁和临界要好得多，故能够使用原子函数解决的多线程同步问题就不应当使用锁或临界区。

13. proc_bind

proc_bind 表示线程如何映射到处理器核心上，OpenMP 目前支持 spread、close、master。spread 表示线程会尽量均匀地分布在各个核心上；close 表示线程会尽量分布在相邻的处理器上；master 表示所有线程都和主线程分布在同一个处理器上。

14. nowait

nowait 构造用于去掉某些构造结束处隐含的同步语句，由于减少了同步消耗，nowait 构造可以潜在地提升程序性能。

nowait 通常和 for 构造共同使用，如代码清单 3-12 所示。

代码清单3-12 OpenMP nowait示例

```
#pragma omp parallel
{
#pragma omp for nowait
for(int i = 0; i < num; i++)
   A[i] = …;//compute result
}
#pragma omp for nowait  schedule(dynamic)
for(int i = 0; i < n; i++)
   B[i] = …;//compute result
}
}
```

由于两个循环之间并没有相互依赖，因此一个线程执行完了就没有必要等待其他线程，直接执行下一个循环即可。

15. flush

OpenMP 中的 flush 是一种存储器屏障，它会保证调用者线程完成操作后，其所有写共享存储器操作都已经完成，并保证其结果对其他线程可见。

在绝大多数情况下，flush 对程序的正确性没有影响，绝大多数并行程序开发人员不应当使用它。

3.4.2 常用的 OpenMP 子句

子句是构造的修饰和补充。不同的构造支持不同的子句组合。

1. collapse 子句

collapse(n) 表示紧随其后的 n 层循环会被合并然后并行化。在一些情况下，collapse 能够解决线程间负载均衡或线程负载太小的问题。一个常见的场景如代码清单 3-13 的伪代码所示。假设有一个双层循环，外层循环次数都比较少，内层循环的计算量也不大。单独使用 OpenMP 线程化内层循环都会存在负载不够的问题，即每个线程的计算量比较小，导致线程的计算时间相比线程的建立、销毁时间不够长；单独使用 OpenMP 线程化外层循环则会存在负载均衡问题。

代码清单3-13 OpenMP collapse示例

```
for(int i = 0; i < 3; i++){
   for(int j = 0; j < num; j++){
          y[i][j] = …;//computing
   }
}
```

对代码清单 3-13 代码的一个常见更好的线程化方法是：同时线程化两层循环。OpenMP 的 collapse 子句提供了直接的支持，使用 OpenMP collapse 子句线程化两层循环的伪代码如代

码清单 3-14 所示。

代码清单3-14 OpenMP collapse示例

```
#pragma omp parallel for collapse(2)
for(int i = 0; i < 3; i++){
     for(int j = 0; j < num; j++){
          y[i][j] = …;//computing
   }
}
```

collapse(2) 表示同时线程化接下来的两层循环，这样循环次数就变成了 3×num，这既改善了负载均衡性，又增加了每个线程的计算量。

2. private 子句

private 将一个或多个变量声明成线程私有，每个线程都会拥有该变量的一个副本，且不允许其他线程染指。private 子句声明的变量的初始值在并行区域的入口处是未定义的，即使并行区域外已经给予了值也不会初始化为在并行区域前同名的共享变量值。一般而言，private 声明的变量都可以使用线程私有栈上的变量替代。另外，出现在 reduction 子句中的参数不能出现在 private 子句中。

代码清单 3-15 展示了 private 子句的功能。

代码清单3-15 OpenMP private示例

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[]){
    int x = 5;
    printf("x = %d, address of x = %p\n", x, &x);
#pragma omp parallel num_threads(3) private(x)
{
    printf("i am %d, x = %d, address of x = %p\n", omp_get_thread_num(), x, &x);
}
    printf("x = %d, address of x = %p\n", x, &x);

    return 0;
}
```

从输出可以看出，并行区域内 x 的值和并行区域外毫无关系，而并行区域前后两者又是一样的。

3. firstprivate 子句

private 变量不能继承并行区域前同名变量的值，但实践中有时需要初始化为原有共享变量的值，OpenMP 提供了 firstprivate 子句来实现这个功能。firstprivate 指定的变量每个线程都有它自己的私有副本，并且继承主线程中的初值（并行区域前的值）。firstprivate 子句并不会

更改原共享变量的值。

代码清单 3-16 展示了 firstprivate 的功能。

代码清单3-16 firstprivate的功能

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[]){
    int x = 5;
#pragma omp parallel num_threads(3) firstprivate(x)
{
printf("i am %d, x = %d\n", omp_get_thread_num(), x);
x++;
}
    printf("x = %d\n", x);

    return 0;
}
```

从结果可以看出，并行区域内的变量 x 值为 5，退出并行区域后，变量 x 的值仍为 5。

4. lastprivate 子句

有时需要在退出并行区域时，将它的值赋给同名的并行区域后变量，lastprivate 子句实现了这一功能。lastprivate 用来指定将线程中的私有变量的值在并行处理结束后复制回主线程中的对应变量。括号内变量值在离开并行域后会保留下来。

代码清单 3-17 展示了 lastprivate 的功能。

代码清单3-17 lastprivate的功能

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[]){
    int x = 5;
#pragma omp parallel for num_threads(3) lastprivate(x)
for(int i = 0; i < 3; i++){
    x = 36;
    printf("i am %d, %d\n", omp_get_thread_num(), x+=i);
}

    printf("x = %d\n", x);

    return 0;
}
```

从结果可以看出，退出 for 循环的并行区域后，共享变量 x 的值发生了改变，而不是保持原来的值不变。

由于并行区域内有多个线程并行执行，因此最后到底是将哪个线程的最终计算结果赋给了对应的变量是一个问题。如果是 for 循环，那么是最后一次循环计算得到的值；如果是 section 构造，那么是最后一个 section 语句中计算得到的值。

5. shared 子句

shared 子句用来声明一个或多个变量是共享变量。需要注意的是，由于在并行区域内的多个线程都可以访问共享变量，因此必须对共享变量的写操作加以保护。为了提高性能，应尽量使用私有变量而不是共享变量。

循环迭代变量在 for 循环并行区域内是私有的。声明在循环构造区域内的自动变量都是私有的。

6. default 子句

default 子句用来指定并行处理区域内没有显式指定访问权限的变量的默认访问权限，其取值有 shared 和 none 两个。指定为 shared 表示在没有显式指定访问权限时，传入并行区域内的变量访问权限为 shared；指定为 none 意味着必须显式地为这些变量指定访问权限。

在某些情况下，OpenMP 默认变量访问权限会导致一些问题，如需要 private 访问权限的数组被默认成 shared 了。故建议显式地使用 default(none) 来去掉变量的默认访问权限。

7. reduction 子句

reduction 子句用来在运算结束时对并行区域内的一个或多个参数执行一个操作。每个线程将创建参数的一个副本，在运算结束时，将各线程的副本进行指定的操作，操作的结果赋值给原始的参数。

reduction 支持的操作符是有限的，支持 + – * / += –= *= /= |&^，且不能是 C++ 重载后的运算符，具体可参见 OpenMP 规范。

代码清单 3-18 是使用 reduction 子句求 π 的 OpenMP 代码示例。

代码清单3-18　使用reduction 子句求π

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

double computePI(int numThread, int num){
   double PI = 0.0;
   const double delta = 1.0/(numThread*num);
#pragma omp parallel for num_threads(numThread) reduction(+:PI)
   for(int i = 0; i < num*numThread; i++){
      double x = (0.5+i)*delta;
      PI += 1.0/(1+x*x);
```

```
    }
    return PI*delta*4;
}

int main(int argc, char *argv[]){
    int numThread, num;
    if(argc != 3){
      printf("usage: ./a.out numThread num\n");
      return 1;
   }else{
      numThread = atoi(argv[1]);
      num = atoi(argv[2]);
      printf("use thread %d, every thread compute times %d\n", numThread, num);
      if(0 == numThread || 0 == num){
        printf("input data format error\n");
        return 1;
      }
    }
clock_t s = clock();
   double PI = computePI(numThread, num);
clock_t e = clock();
   printf("PI is %1.15f, use clock %ld\n", PI, e-s);
   return 0;
}
```

8. schedule 子句

schedule 子句指定采取的负载均衡策略及每次分发的数据大小。其使用方式如下：

```
schedule(type[,size])
```

其中，size 参数是可选的；type 参数表示负载均衡策略，主要有 4 种：dynamic、guided、static 和 runtime。

实际上只有 static、dynamic、guided 三种方式，runtime 指的是根据环境变量的设置来选择前 3 种中的某一种。

size 参数表示每次分发的循环迭代次数，必须是整数。3 种策略都可用可不用 size 参数。当 type 为 runtime 时，需要忽略 size 参数。

1）静态负载均衡策略（static）：当编译指导语句没有指定 schedule 子句时，大部分系统中默认采用 static，static 方式使用的算法非常简单。假设有从 0 开始的 N 次循环迭代，K 个线程，如果没有指定 size 参数，那么给第一个线程分配的迭代为 [0，N/K]。因为 N/K 不一定是整数，因此存在着轻微的负载均衡问题；如果指定 size 参数的话，那么分配给第一个线程的是 [0，size−1][size*K，size*K+size−1]…，其他线程类推，直到分配完循环迭代。

2）动态负载均衡策略（dynamic）：动态负载均衡策略动态地将迭代分发到各个线程。当线程计算完时，就会取得下一次任务。在静态负载均衡策略失效时，使用动态负载均衡策略

通常可以得到性能提升。

动态负载均衡的机制类似于工作队列，线程在并行执行的时候，不断从这个队列中取出相应的工作完成，直到队列为空为止。由于每一个线程在执行的过程中的线程标识号是不同的，可以根据这个线程标识号来分配不同的任务。OpenMP 中动态负载均衡时，每次分发 size 个循环计算，而队列中的数据就是所有的循环变量值。

动态负载均衡策略可用可不用 size 参数，不使用 size 参数时等价于 size 为 1。使用 size 参数时，每次分配给线程的迭代次数为指定的 size 次。

选择 size 参数时，要注意伪共享问题。通常要使得每个线程从内存中加载的数据大小可以占满一个缓存线。

3）指导负载均衡策略（guided）：guided 负载均衡策略采用启发式自调度方法。开始时每个线程会分配较大的循环次数，之后分配的循环次数会逐渐减小。通常每次分配的循环次数会指数级下降到指定的 size 大小，如果没有指定 size 参数，那么最后会降到 1。

4）运行时负载均衡策略（rumtime）：runtime 是指在运行时根据环境变量 OMP_SCHEDULE 的值来确定负载均衡策略，最终使用的负载均衡策略是上述 3 种中的一种。

例子代码清单 3-19 展示了不同的负载均衡策略的行为，读者可使用不同的负载均衡策略仔细观察结果，以加深对 OpenMP 负载均衡策略的理解。

代码清单3-19　不同负载均衡策略示例

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[]){
#pragma omp parallel for num_threads(3) schedule(dynamic, 2)
for(int i = 0; i < 300; i++){
    printf("i am %d, i = %d\n", omp_get_thread_num(), i);
}

    return 0;
}
```

代码代码清单 3-20 展示了因为负载均衡策略不当而导致的伪共享问题。

代码清单3-20　伪共享示例

```
#include <time.h>
#include <omp.h>
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[]){
    const int n = 500000;
    float a[n], b[n], c[n];

    for(int i = 0; i < n; i++){
```

```
        a[i] = 1.0f*rand()/n;
        b[i] = 1.0f*rand()/n;
    }
clock_t s = clock();
#pragma omp parallel for num_threads(4) schedule(dynamic, 1)
   for(int i = 0; i < n; i++)
      c[i] = a[i]*6/b[i] + 5;
clock_t e = clock();
printf("use time = %ld clocks, %f\n", e-s, c[250]);
    return 0;
}
```

简单分析如下：以第一次计算为例，线程 0 读写 c[0]，线程 1 读写 c[1]，线程 2 读写 c[2]，线程 3 读写 c[3]，这 4 个线程读写的 c 数组缓存在同一缓存线上，因此存在伪共享问题。解决这个性能问题的方法很简单，去掉负载均衡语句，或改成静态负载均衡，或将每次分配的块大小改为 16（这个在将来的机器上可能不是最优的）。

实际上，上述例子的性能可能不会受影响，因为现代 X86 CPU 具有流处理指令，对于只读不写或只写不读的数据，如果访问步长为 1，就可能会被流处理。

9. if 子句

子句 if 提供了一种由运行时决定是否并行的机制，如果 if 条件为真，并行区域会被多个线程执行，否则只由一个线程执行。if 语句示例如代码清单 3-21 所示。

代码清单3-21　if语句示例

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[]){
    int x = 5;

#pragma omp parallel for num_threads(3) if(x <= 5)
for(int i = 0; i < 3; i++){
    printf("i am %d, address of x = %p\n", omp_get_thread_num(), &x);
}

    return 0;
}
```

由于 x 等于 5，因此满足 if 条件，所以并行区域 parallel for 循环会被多个线程执行；如果将 if 条件改为 x ＞ 5；那么并行区域将会只有一个线程执行。

3.5　OpenMP 异构并行计算

OpenMP 4.0 加入了异构计算的支持，包括构造和子句。这些伪指令和 OpenACC 功能类

似，但是语法做了一些适应 OpenMP 的修改，并支持更多、更广泛的加速器硬件。在笔者写作此书时，还没有编译器支持本节的构造和子句，因此笔者只简要介绍，不给出示例。

1. 构造

由于异构计算涉及多种处理器和存储器，故需要语句来指明代码在哪个处理器上执行，数据如何在主机存储器和加速器存储器间映射。

映射包括两层含义：

- 如果数据在同一个物理存储器上，那么可直接使用原来的存储器空间。
- 如果数据在不同的物理存储器上，那么需要在加速器上分配存储器空间，并且在加速器存储器空间和主机存储器空间之间复制数据。

1）target：target 表示块内代码在加速器上执行，同时进入代码块之前需要将相关的数据映射到加速器上 (map)，而计算完成后需要将数据从加速器映射回主机上。

2）target data：target data 块内的多个 target 代码块共享同一份加速器上的数据存储。通常使用 target data 是为了减小数据的传输开销。

3）target update：target update 用于 target data 内部，用来显式地在加速器和主机间传输数据，其子句有 to 和 from，分别表示从主机到加速器和从加速器到主机的传输。

4）teams：teams 表示紧随其后的循环会在多个线程组之间分割，一个线程组计算一部分。

2. 子句

1）device：device 子句表示紧随其后的代码块将会在其指定的加速器上执行。

2）num_teams：num_teams 指定线程组的数目。

3）map：map 指定数据映射的方向，如果加速器和主机使用不同的物理存储器，那么就意味着数据复制。map 具有 to 和 from 两个方向。to 表示从主机到加速器，而 from 则相反。

4）distribute：distribute 表示在线程组内分割循环计算，但这不意味着循环计算会在线程组内线程间分割（只有线程组内的主线程会运算）。

5）dist_schedule：dist_schedule 含义和 schedule 一样，不同的是其作用对象是线程组。比如 dist_schedule(static，1024) 指采用静态调度策略，每次为每个线程组分发 1024 个循环。

3.6 OpenMP 程序优化

通常影响 OpenMP 程序性能的因素有：代码的质量、编译器、库和操作系统的效率。

3.6.1 OpenMP 程序优化准则

本节主要关注于 OpenMP 代码的质量。提高 OpenMP 代码质量的基本方法主要有以下

几种：

- 尽量一次进入一个并行段，然后在内部并行，这样可以减少多次建立线程的开销。
- 尽量使用和处理器核数相同的线程数，这个准则不是绝对的，有时（尤其是在支持超线程的机器上）使用二倍处理器核数的线程可能获得更好的性能。
- 加大并行块的粒度，尽量并行外层循环，如果必须并行内层循环，可考虑将内层循环转化为外层循环再并行。在外层循环上用 parallel，内层用 for，并同时注意同步。
- 减小 critical 块大小。在保证正确性的前提下，尽量将操作移到 critical 外面，并且尽量使用原子操作代替。
- 如果默认情况下负载非常不平衡，使用动态负载均衡策略；如果循环层数很多，但是每层的循环次数都比较小的话，可以考虑合并循环或 collapse。
- 少用 barrier，并尽可能地使用 nowait，尤其是在并行域结束的地方。
- 不要共享不必要共享的数据，也就是尽量 default(none)。
- 当 shared 和 private 都适用时，使用 shared，因为 private 会为每个线程复制一份存储。

3.6.2 OpenMP 并行优化实例

利用积分法计算 π 值是一个非常适合并行处理的例子，本节就以此例子说明 OpenMP 同步机制的选择。

1. 串行代码

在 reduction 节，本文说明了一种计算 π 的方式，下面的代码是其对应的串行代码。本节以代码清单 3-22 代码为出发点，展示一些并行代码优化的策略。

代码清单3-22 串行代码

```
float cpuPI(int num){
    float sum = 0.0f;
    float temp;

    for(int i = 0; i < num; i++){
        temp= (i+0.5f)/num;
        sum+=4/(1+temp*temp);
    }

    return sum/num;
}
```

从代码可以看出：除了一个除法和一个赋值外，计算几乎完全消耗在 for 循环。如果能够将其并行化，那么获得的加速将会相当可观。

2. 临界区实现

可以看出，并行的障碍在对 sum 的更新上，因此最简单的办法就是使用临界区保护对应代码。修改后实现如代码清单 3-23 所示。

代码清单3-23 临界区实现

```
float cpuPIOMPCriticalNaive(int num){
    float sum=0.0f;
#pragma omp parallel for default(none) shared(num, sum)
    for(int i = 0; i < num; i++){
        float temp = (i+0.5f)/num;
    #pragma omp critical
    {
        sum += 4/(1+temp*temp);
    }
    }

    return sum/num;
}
```

由于使用临界区导致多线程串行执行的时间长，这段代码的性能通常会比串行还慢。代码清单 3-24 是一个优化版本，主要要点是尽量减少临界区的代码量。

代码清单3-24 临界区优化

```
float cpuPIOMPCritical(int num){
    float sum = 0.0f;
#pragma omp parallel for default(none) shared(num, sum)
   for(int i = 0; i < num; i++){
      float temp = (i+0.5f)/num;
      temp = 4/(1+temp*temp);
    #pragma omp critical
    {
      sum += temp;
    }
    }
    return sum/num;
}
```

代码清单 3-23 中可以看出，需要使用临界区保护的只是对 sum 变量的更新，而计算时不用放到临界区里面的，故将其转移到临界区外会更好，以减少总的串行执行的代码。

实际上循环内只对变量 sum 进行了加法运算，而原子操作支持加法，故可以使用 OpenMP 支持的原子操作进一步提升性能。

3. 原子操作实现

对变量 sum 的更新是简单的浮点加法运算，OpenMP 的原子操作性能比临界区要好。本节改用原子操作来实现，如代码清单 3-25 所示。

代码清单3-25　原子操作

```
float cpuPIOMPAtomic(int num){
   float sum = 0.0f;
#pragma omp parallel for default(none) shared(num, sum)
   for(int i = 0; i < num; i++){
      float temp = (i+0.5f)/num;
      temp = 4/(1+temp*temp);
   #pragma omp atomic
      sum += temp;
    }

    return sum/num;
}
```

原子操作是 OpenMP 中性能更好的互斥访问方式，实际上从代码来看，多个线程对 sum 的更新满足归约模式，故使用 OpenMP 中的归约语句会是更好的选择。

4. 归约实现

无论是临界区，还是原子操作，都会导致多线程之间的等待，从而损害程序性能。下面使用的归约操作无须线程之间相互等待，因此应该能够提升其性能。其代码如代码清单 3-26 所示。

代码清单3-26　归约

```
float cpuPIOMPReduction(int num){
   float sum = 0.0f;
#pragma omp parallel for default(none) shared(num) reduction(+:sum)
   for(int i = 0; i < num; i++){
      float temp = (i+0.5f)/num;
      temp = 4/(1+temp*temp);
      sum += temp;
    }
    return sum/num;
}
```

从互斥访问来看，归约完全去掉了对共享变量 sum 的互斥访问，故性能应当比原子操作版本更优。

每个线程的互斥访问去掉了，而现在的处理器都是向量处理器，因此可以使用向量化进一步提升性能。

5. 向量化

OpenMP 4.0 引入了 simd 子句，以支持向量化。使用向量化进一步提升性能的代码如代码清单 3-27 所示。

代码清单3-27　OpenMP向量化

```
float cpuPIOMPReduction(int num){
    float sum = 0.0f;
```

```
#pragma omp parallel simd for default(none) shared(num) reduction(+:sum)
   for(int i = 0; i < num; i++){
      float temp = (i+0.5f)/num;
      temp = 4/(1+temp*temp);
      sum += temp;
    }

    return sum/num;
}
```

在笔者写作本书时，GCC 编译器还没有支持 simd，相信读者看到本书的时候，GCC 已经支持 simd 了。

6. 主函数

本节给出了程序的主函数，如代码清单 3-28 所示。希望读者能够在自己的机器上实际运行，并观察各个不同版本的性能结果，以感受 OpenMP 支持的不同互斥方式的性能。

代码清单3-28　主函数

```
int main(int argc, char *argv[]){
   const int num = 1000000;
   long s, e;
   float pi;

   s = getTimeOfMSecond();
   pi = cpuPI(num);
   e = getTimeOfMSecond();
   printf("serial time = %d clocks, pi = %f\n", e-s, pi);

   s = getTimeOfMSecond();
   pi = cpuPIOMPCriticalNaive(num);
   e = getTimeOfMSecond();
   printf("naive time = %d clocks, pi = %f\n", e-s, pi);

   s = getTimeOfMSecond();
   pi = cpuPIOMPCritical(num);
   e = getTimeOfMSecond();
   printf("critical time = %d clocks, pi = %f\n", e-s, pi);

   s = getTimeOfMSecond();
   pi = cpuPIOMPAtomic(num);
   e = getTimeOfMSecond();
   printf("atomic time = %d clocks, pi = %f\n", e-s, pi);

   s = getTimeOfMSecond();
   pi = cpuPIOMPReduction(num);
   e = getTimeOfMSecond();
   printf("reduction time = %d clocks, pi = %f\n", e-s, pi);
```

```
    return 0;
}
```

3.7　本章小结

本章首先介绍 OpenMP 编程模型，即 OpenMP 的执行模型和存储器模型，然后简单介绍了 OpenMP 的环境变量和函数，最后详细介绍并用代码示例了 OpenMP 中最常见的部分——编译制导语句。OpenMP 4.0 为了支持异构并行计算，引入了一些新的编译制导语句，目前只有很少的编译器支持这些语句，因此没有给出实例。

本章介绍了 OpenMP 程序优化准则，然后以使用 OpenMP 优化计算圆周率为率，详细展示了如何在 OpenMP 中使用临界区、原子操作、归约和向量化。