OpenMP 编程指南

进入多核时代后，必须使用多线程编写程序才能让各个 CPU 核得到利用。在单核时代，通常使用操作系统提供的 API 来创建线程，然而，在多核系统中，情况发生了很大的变化， 如果仍然使用操作系统 API 来创建线程会遇到一些问题。具体来说，有以下三个问题：

1）CPU 核数扩展性问题

多核编程需要考虑程序性能随 CPU 核数的扩展性，即硬件升级到更多核后，能够不修改程序就让程序性能增长，这要求程序中创建的线程数量需要随 CPU 核数变化，不能创建固定数量的线程，否则在 CPU 核数超过线程数量上的机器上运行，将无法完全利用机器性能。虽然通过一定方法可以使用操作系统 API 创建可变化数量的线程，但是比较麻烦，不如 OpenMP 方便。

2）方便性问题

在多核编程时，要求计算均摊到各个 CPU 核上去，所有的程序都需要并行化执行，对计算的负载均衡有很高要求。这就要求在同一个函数内或同一个循环中，可能也需要将计算分摊到各个 CPU 核上，需要创建多个线程。操作系统 API 创建线程时，需要线程入口函数，很难满足这个需求，除非将一个函数内的代码手工拆成多个线程入口函数，这将大大增加程序员的工作量。使用 OpenMP 创建线程则不需要入口函数，非常方便，可以将同一函数内的代码分解成多个线程执行，也可以将一个 for 循环分解成多个线程执行。

3）可移植性问题

目前各个主流操作系统的线程 API 互不兼容，缺乏事实上的统一规范，要满足可移植性得自己写一些代码，将各种不同操作系统的 api 封装成一套统一的接口。OpenMP 是标准规范，所有支持它的编译器都是执行同一套标准，不存在可移植性问题。

OpenMP 并行程序设计（一）

OpenMP 是一个支持共享存储并行设计的库，特别适宜多核 CPU 上的并行程序设计。、

先看一个简单的使用了 OpenMP 程序

```
int main(int argc, char* argv[])
{
#pragma omp parallel for
	for (int i = 0; i < 10; i++ )
	{
		printf("i = %d\n", i);
	}
	return 0;
}
```

这个程序执行后打印出以下结果：

```
i = 0
i = 5
i = 1
i = 6
i = 2
i = 7
i = 3
i = 8
i = 4
i = 9
```

可见 for 循环语句中的内容被并行执行了。（每次运行的打印结果可能会有区别）
这里要说明一下，#pragma omp parallel for 这条语句是用来指定后面的 for 循环语句变成并行执行的，当然 for 循环里的内容必须满足可以并行执行，即每次循环互不相干，后一次循环不依赖于前面的循环。

有关#pragma omp parallel for 这条语句的具体含义及相关 OpenMP 指令和函数的介绍暂时先放一放，只要知道这条语句会将后面的 for 循环里的内容变成并行执行就行了。

将 for 循环里的语句变成并行执行后效率会不会提高呢，我想这是我们最关心的内容了。下面就写一个简单的测试程序来测试一下：

```
void test()
{
	int a = 0;
	clock_t t1 = clock();
	for (int i = 0; i < 100000000; i++)
```

```
        {
            a = i+1;
        }
        clock_t t2 = clock();
        printf("Time = %d\n", t2-t1);
}

int main(int argc, char* argv[])
{
        clock_t t1 = clock();
#pragma omp parallel for
        for ( int j = 0; j < 2; j++ ){
            test();
        }
        clock_t t2 = clock();
        printf("Total time = %d\n", t2-t1);

        test();
        return 0;
}
```

在 test()函数中，执行了 1 亿次循环，主要是用来执行一个长时间的操作。
在 main（）函数里，先在一个循环里调用 test()函数，只循环 2 次，我们还是看一下在双核 CPU 上的运行结果吧：

```
Time = 297
Time = 297
Total time = 297
Time = 297
```

可以看到在 for 循环里的两次 test()函数调用都花费了 297ms， 但是打印出的总时间却只花费了 297ms，后面那个单独执行的 test()函数花费的时间也是 297ms，可见使用并行计算后效率提高了整整一倍。

OpenMP 并行程序设计（二）

1、fork/join 并行执行模式的概念

OpenMP 是一个编译器指令和库函数的集合，主要是为共享式存储计算机上的并行程序设计使用的。
前面一篇文章中已经试用了 OpenMP 的一个 Parallel for 指令。从上篇文章中我们也可以发现 OpenMP 并行执行的程序要全部结束后才能执行后面的非并行部分的代码。这就是标准的并行模式 fork/join 式并行模式，共享存储式并行程序就是使用 fork/join 式并行的。
标准并行模式执行代码的基本思想是，程序开始时只有一个主线程，程序中的串行部分都由主线程执行，并行的部分是通过派生其他线程来执行，但是如果并行部分没有结束时是不会执行串行部分的，如上一篇文章中的以下代码：

```
int main(int argc, char* argv[])
{
	clock_t t1 = clock();
#pragma omp parallel for
	for ( int j = 0; j < 2; j++ ){
		test();
	}
	clock_t t2 = clock();
	printf("Total time = %d\n", t2-t1);

	test();
	return 0;
}
```

在没有执行完 for 循环中的代码之前，后面的 clock_t t2 = clock();这行代码是不会执行的，如果和调用线程创建函数相比，它相当于先创建线程，并等待线程执行完，所以这种并行模式中在主线程里创建的线程并没有和主线程并行运行。

2、OpenMP 指令和库函数介绍

下面来介绍 OpenMP 的基本指令和常用指令的用法，
在 C/C++中，OpenMP 指令使用的格式为

```
	# pragma omp  指令  [子句[子句]…]
```

前面提到的 parallel for 就是一条指令，有些书中也将 OpenMP 的“指令”叫做“编译指导语句”，后面的子句是可选的。例如：

```
#pragma omp parallel private(i, j)
```

parallel 就是指令， private 是子句
为叙述方便把包含#pragma 和 OpenMP 指令的一行叫做语句，如上面那行叫 parallel 语句。

OpenMP 的指令有以下一些：

parallel，用在一个代码段之前，表示这段代码将被多个线程并行执行

for，用于 for 循环之前，将循环分配到多个线程中并行执行，必须保证每次循环之间无相关性。

parallel for， parallel 和 for 语句的结合，也是用在一个 for 循环之前，表示 for 循环的代码将被多个线程并行执行。

sections，用在可能会被并行执行的代码段之前

parallel sections，parallel 和 sections 两个语句的结合

critical，用在一段代码临界区之前

single，用在一段只被单个线程执行的代码段之前，表示后面的代码段将被单线程执行。

barrier，用于并行区内代码的线程同步，所有线程执行到 barrier 时要停止，直到所有线程都执行到 barrier 时才继续往下执行。

atomic，用于指定一块内存区域被制动更新

master，用于指定一段代码块由主线程执行

ordered， 用于指定并行区域的循环按顺序执行

threadprivate, 用于指定一个变量是线程私有的。

OpenMP 除上述指令外，还有一些库函数，下面列出几个常用的库函数：

omp_get_num_procs, 返回运行本线程的多处理机的处理器个数。

omp_get_num_threads, 返回当前并行区域中的活动线程个数。

omp_get_thread_num, 返回线程号

omp_set_num_threads, 设置并行执行代码时的线程个数

omp_init_lock, 初始化一个简单锁

omp_set_lock， 上锁操作

omp_unset_lock， 解锁操作，要和 omp_set_lock 函数配对使用。

omp_destroy_lock， omp_init_lock 函数的配对操作函数，关闭一个锁

OpenMP 的子句有以下一些

private, 指定每个线程都有它自己的变量私有副本。

firstprivate，指定每个线程都有它自己的变量私有副本，并且变量要被继承主线程中的初值。

lastprivate，主要是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。

reduce，用来指定一个或多个变量是私有的，并且在并行处理结束后这些变量要执行指定的运算。

nowait，忽略指定中暗含的等待

num_threads，指定线程的个数

schedule，指定如何调度 for 循环迭代

shared，指定一个或多个变量为多个线程间的共享变量

ordered，用来指定 for 循环的执行要按顺序执行

copyprivate，用于 single 指令中的指定变量为多个线程的共享变量

copyin，用来指定一个 threadprivate 的变量的值要用主线程的值进行初始化。

default，用来指定并行处理区域内的变量的使用方式，缺省是 shared

3、parallel 指令的用法

parallel 是用来构造一个并行块的，也可以使用其他指令如 for、sections 等和它配合使用。
在 C/C++中，parallel 的使用方法如下：

```
#pragma omp parallel [for | sections] [子句[子句]…]
{
        //代码
}
```

parallel 语句后面要跟一个大括号对将要并行执行的代码括起来。

```
void main(int argc, char *argv[]) {
#pragma omp parallel
{
                printf(“Hello, World!\n”);
}
}
```

执行以上代码将会打印出以下结果

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

可以看得出 parallel 语句中的代码被执行了四次，说明总共创建了 4 个线程去执行 parallel 语句中的代码。
也可以指定使用多少个线程来执行，需要使用 num_threads 子句：

```
void main(int argc, char *argv[]) {
#pragma omp parallel num_threads(8)
{
                printf(“Hello, World!, ThreadId=%d\n”, omp_get_thread_num() );
}
}
```

执行以上代码，将会打印出以下结果：

```
Hello, World!, ThreadId = 2
Hello, World!, ThreadId = 6
Hello, World!, ThreadId = 4
Hello, World!, ThreadId = 0
Hello, World!, ThreadId = 5
Hello, World!, ThreadId = 7
Hello, World!, ThreadId = 1
Hello, World!, ThreadId = 3
```

从 ThreadId 的不同可以看出创建了 8 个线程来执行以上代码。所以 parallel 指令是用来为一段代码创建多个线程来执行它的。parallel 块中的每行代码都被多个线程重复执行。
和传统的创建线程函数比起来，相当于为一个线程入口函数重复调用创建线程函数来创建线程并等待线程执行完。

4、for 指令的使用方法

for 指令则是用来将一个 for 循环分配到多个线程中执行。for 指令一般可以和 parallel 指令合起来形成 parallel for 指令使用，也可以单独用在 parallel 语句的并行块中。

#pragma omp [parallel] for [子句]
for 循环语句

先看看单独使用 for 语句时是什么效果：

```
int j = 0;
#pragma omp for
    for ( j = 0; j < 4; j++ ){
        printf("j = %d, ThreadId = %d\n", j, omp_get_thread_num());
    }
```

执行以上代码后打印出以下结果

```
j = 0, ThreadId = 0
j = 1, ThreadId = 0
j = 2, ThreadId = 0
j = 3, ThreadId = 0
```

从结果可以看出四次循环都在一个线程里执行，可见 for 指令要和 parallel 指令结合起来使用才有效果：

如以下代码就是 parallel 和 for 一起结合成 parallel for 的形式使用的：

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

执行后会打印出以下结果：

```
j = 0, ThreadId = 0
j = 2, ThreadId = 2
j = 1, ThreadId = 1
j = 3, ThreadId = 3
```

可见循环被分配到四个不同的线程中执行。

上面这段代码也可以改写成以下形式：

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

执行以上代码会打印出以下结果：

```
j = 1, ThreadId = 1
j = 3, ThreadId = 3
j = 2, ThreadId = 2
j = 0, ThreadId = 0
```

在一个 parallel 块中也可以有多个 for 语句，如：

```
int j;
#pragma omp parallel
{
#pragma omp for
        for ( j = 0; j < 100; j++ ){
                …
        }
#pragma omp for
        for (    j = 0; j < 100; j++ ){
                …
        }
…
}
```

for 循环语句中，书写是需要按照一定规范来写才可以的，即 for 循环小括号内的语句要按照一定的规范进行书写，for 语句小括号里共有三条语句

```
for( i=start; i < end; i++)
```

i=start; 是 for 循环里的第一条语句，必须写成 “变量＝初值” 的方式。如 i＝0

i < end;是 for 循环里的第二条语句，这个语句里可以写成以下 4 种形式之一：

变量 < 边界值

变量 <= 边界值

变量 > 边界值

变量 >= 边界值

如 i>10 i< 10 i>=10 i>10 等等

最后一条语句 i++可以有以下 9 种写法之一

```
i++
++i
i--
--i
i += inc
i -= inc
i = i + inc
i = inc + i
i = i –inc
```

例如 i += 2; i -= 2；i = i + 2；i = i - 2；都是符合规范的写法。

5. sections 和 section 指令的用法

section 语句是用在 sections 语句里用来将 sections 语句里的代码划分成几个不同的段，每段都并行执行。用法如下：

```
#pragma omp [parallel] sections [子句]
```

```
{
    #pragma omp section
    {
            代码块
    }
}
```

先看一下以下的例子代码：

```
void main(int argc, char *argv)
{
#pragma omp parallel sections {
#pragma omp section
    printf("section 1 ThreadId = %d\n", omp_get_thread_num());
#pragma omp section
    printf("section 2 ThreadId = %d\n", omp_get_thread_num());
#pragma omp section
    printf("section 3 ThreadId = %d\n", omp_get_thread_num());
#pragma omp section
    printf("section 4 ThreadId = %d\n", omp_get_thread_num());
}
```

执行后将打印出以下结果：

```
section 1 ThreadId = 0
section 2 ThreadId = 2
section 4 ThreadId = 3
section 3 ThreadId = 1
```

从结果中可以发现第 4 段代码执行比第 3 段代码早，说明各个 section 里的代码都是并行执行的，并且各个 section 被分配到不同的线程执行。

使用 section 语句时，需要注意的是这种方式需要保证各个 section 里的代码执行时间相差不大，否则某个 section 执行时间比其他 section 过长就达不到并行执行的效果了。

上面的代码也可以改写成以下形式：

```
void main(int argc, char *argv)
{
#pragma omp parallel {
#pragma omp sections
{
#pragma omp section
        printf("section 1 ThreadId = %d\n", omp_get_thread_num());
#pragma omp section
        printf("section 2 ThreadId = %d\n", omp_get_thread_num());
}
#pragma omp sections
```

```
{

#pragma omp section
        printf("section 3 ThreadId = %d\n", omp_get_thread_num());
#pragma omp section
        printf("section 4 ThreadId = %d\n", omp_get_thread_num());
}
}
```

执行后将打印出以下结果：

```
section 1 ThreadId = 0
section 2 ThreadId = 3
section 3 ThreadId = 3
section 4 ThreadId = 1
```

这种方式和前面那种方式的区别是，两个 sections 语句是串行执行的，即第二个 sections 语句里的代码要等第一个 sections 语句里的代码执行完后才能执行。

用 for 语句来分摊是由系统自动进行，只要每次循环间没有时间上的差距，那么分摊是很均匀的，使用 section 来划分线程是一种手工划分线程的方式，最终并行性的好坏得依赖于程序员。

本篇文章中讲的几个 OpenMP 指令 parallel, for, sections, section 实际上都是用来如何创建线程的，这种创建线程的方式比起传统调用创建线程函数创建线程要更方便，并且更高效。当然，创建线程后，线程里的变量是共享的还是其他方式，主线程中定义的变量到了并行块内后还是和传统创建线程那种方式一样的吗？创建的线程是如何调度的？等等诸如此类的问题到下一篇文章中进行讲解。

OpenMP 中的数据处理子句

1. private 子句

private 子句用于将一个或多个变量声明成线程私有的变量，变量声明成私有变量后，指定每个线程都有它自己的变量私有副本，其他线程无法访问私有副本。即使在并行区域外有同名的共享变量，共享变量在并行区域内不起任何作用，并且并行区域内不会操作到外面的共享变量。

private 子句的用法格式如下：

private(list)

下面便是一个使用 private 子句的代码例子：

```
        int k = 100;
#pragma omp parallel for private(k)
        for ( k=0; k < 10; k++)
        {
                printf("k=%d\n", k);
        }

        printf("last k=%d\n", k);
```

上面程序执行后打印的结果如下：

```
k=6
k=7
k=8
k=9
k=0
k=1
k=2
k=3
k=4
k=5
last k=100
```

从打印结果可以看出，for 循环前的变量 k 和循环区域内的变量 k 其实是两个不同的变量。用 private 子句声明的私有变量的初始值在并行区域的入口处是未定义的，它并不会继承同名共享变量的值。

出现在 reduction 子句中的参数不能出现在 private 子句中。

2. firstprivate 子句

private 声明的私有变量不能继承同名变量的值，但实际情况中有时需要继承原有共享变量的值，OpenMP 提供了 firstprivate 子句来实现这个功能。

先看一下以下的代码例子

```
        int k = 100;
#pragma omp parallel for firstprivate(k)
        for ( i=0; i < 4; i++)
        {
                k+=i;
```

```
                printf("k=%d\n",k);
        }

        printf("last k=%d\n", k);
```

上面代码执行后打印结果如下：

```
k=100
k=101
k=103
k=102
last k=100
```

从打印结果可以看出，并行区域内的私有变量 k 继承了外面共享变量 k 的值 100 作为初始值，并且在退出并行区域后，共享变量 k 的值保持为 100 未变。

3. lastprivate 子句

有时在并行区域内的私有变量的值经过计算后，在退出并行区域时，需要将它的值赋给同名的共享变量，前面的 private 和 firstprivate 子句在退出并行区域时都没有将私有变量的最后取值赋给对应的共享变量，lastprivate 子句就是用来实现在退出并行区域时将私有变量的值赋给共享变量。

举个例子如下：

```
        int k = 100;
#pragma omp parallel for firstprivate(k),lastprivate(k)
        for ( i=0; i < 4; i++)
        {
                k+=i;
                printf("k=%d\n",k);
        }
        printf("last k=%d\n", k);
```

上面代码执行后的打印结果如下：

```
k=100
k=101
k=103
k=102
last k=103
```

从打印结果可以看出，退出 for 循环的并行区域后，共享变量 k 的值变成了 103，而不是保持原来的 100 不变。

由于在并行区域内是多个线程并行执行的，最后到底是将那个线程的最终计算结果赋给了对应的共享变量呢？OpenMP 规范中指出，如果是循环迭代，那么是将最后一次循环迭代中的值赋给对应的共享变量；如果是 section 构造，那么是最后一个 section 语句中的值赋给对应的共享变量。注意这里说的最后一个 section 是指程序语法上的最后一个，而不是实际运行时的最后一个运行完的。

如果是类（class）类型的变量使用在 lastprivate 参数中，那么使用时有些限制，需要一个可

访问的，明确的缺省构造函数，除非变量也被使用作为 firstprivate 子句的参数；还需要一个拷贝赋值操作符，并且这个拷贝赋值操作符对于不同对象的操作顺序是未指定的，依赖于编译器的定义。

4. threadprivate 子句

threadprivate 子句用来指定全局的对象被各个线程各自复制了一个私有的拷贝，即各个线程具有各自私有的全局对象。
用法如下：
#pragma omp threadprivate(list) new-line
下面用 threadprivate 命令来实现一个各个线程私有的计数器，各个线程使用同一个函数来实现自己的计数。计数器代码如下：

```
int counter = 0;
#pragma omp threadprivate(counter)
int increment_counter()
{
        counter++;
        return(counter);
}
```

如果对于静态变量也同样可以使用 threadprivate 声明成线程私有的，上面的 counter 变量如改成用 static 类型来实现时，代码如下：

```
int increment_counter2()
{
static int counter = 0;
#pragma omp threadprivate(counter)
        counter++;
        return(counter);
}
```

threadprivate 和 private 的区别在于 threadprivate 声明的变量通常是全局范围内有效的，而 private 声明的变量只在它所属的并行构造中有效。
threadprivate 的对应只能用于 copyin，copyprivate，schedule，num_threads 和 if 子句中，不能用于任何其他子句中。
用作 threadprivate 的变量的地址不能是常数。
对于 C++的类（class）类型变量，用作 threadprivate 的参数时有些限制，当定义时带有外部初始化时，必须具有明确的拷贝构造函数。
对于 windows 系统，threadprivate 不能用于动态装载（使用 LoadLibrary 装载）的 DLL 中，可以用于静态装载的 DLL 中，关于 windows 系统中的更多限制，请参阅 MSDN 中有关 threadprivate 子句的帮助材料。
有关 threadprivate 命令的更多限制方面的信息，详情请参阅 OpenMP2.5 规范。

5.shared 子句

shared 子句用来声明一个或多个变量是共享变量。
用法如下：
shared(list)
需要注意的是，在并行区域内使用共享变量时，如果存在写操作，必须对共享变量加以保护，否则不要轻易使用共享变量，尽量将共享变量的访问转化为私有变量的访问。
循环迭代变量在循环构造区域里是私有的。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 子句用来允许用户控制并行区域中变量的共享属性。
用法如下：
default(shared | none)
使用 shared 时，缺省情况下，传入并行区域内的同名变量被当作共享变量来处理，不会产生线程私有副本，除非使用 private 等子句来指定某些变量为私有的才会产生副本。
如果使用 none 作为参数，那么线程中用到的变量必须显示指定是共享的还是私有的，除了那些由明确定义的除外。

7.reduction 子句

reduction 子句主要用来对一个或多个参数条目指定一个操作符，每个线程将创建参数条目的一个私有拷贝，在区域的结束处，将用私有拷贝的值通过指定的运行符运算，原始的参数条目被运算结果的值更新。

reduction 子句用法如下:
reduction(operator:list)
下表列出了可以用于 reduction 子句的一些操作符以及对应私有拷贝变量缺省的初始值，私有拷贝变量的实际初始值依赖于 redtucion 变量的数据类型。

reduction 操作中各种操作符号对应拷贝变量的缺省初始值

Operator	Initialization value
+	0
*	1
-	0
&	~0
|	0
^	0
&&	1
||	0

例如一个整数求和的程序如下：

```
int i, sum = 100;

#pragma omp parallel for reduction(+: sum)
for ( i = 0; i < 1000; i++ )
```

```
{
  sum += i;
}

printf( "sum = %ld\n", sum);
```

注意，如果在并行区域内不加锁保护就直接对共享变量进行写操作，存在数据竞争问题，会导致不可预测的异常结果。共享数据作为 private、firstprivate、lastprivate、threadprivate、reduction 子句的参数进入并行区域后，就变成线程私有了，不需要加锁保护了。

8. copyin 子句

copyin 子句用来将主线程中 threadprivate 变量的值拷贝到执行并行区域的各个线程的 threadprivate 变量中，便于线程可以访问主线程中的变量值，
用法如下：
copyin(list)

copyin 中的参数必须被声明成 threadprivate 的，对于类类型的变量，必须带有明确的拷贝赋值操作符。
对于前面 threadprivate 中讲过的计数器函数，如果多个线程使用时，各个线程都需要对全局变量 counter 的副本进行初始化，可以使用 copyin 子句来实现，示例代码如下：

```
int main(int argc, char* argv[])
{
        int iterator;
#pragma omp parallel sections copyin(counter)
        {
#pragma omp section
                {
                        int count1;
                        for ( iterator = 0; iterator < 100; iterator++ )
                        {
                                count1 = increment_counter();
                        }
                        printf("count1 = %ld\n", count1);
                }
#pragma omp section
                {
                        int count2;
                        for ( iterator = 0; iterator < 200; iterator++ )
                        {
                                count2 = increment_counter();
                        }
                        printf("count2 = %ld\n", count2);
                }
```

```
        }
        printf("counter = %ld\n", counter);
}
```

打印结果如下：

```
count1 = 100
count2 = 200
counter = 0
```

从打印结果可以看出，两个线程都正确实现了各自的计数。

9. copyprivate 子句

copyprivate 子句提供了一种机制用一个私有变量将一个值从一个线程广播到执行同一并行区域的其他线程。

用法如下：

```
copyprivate(list)
```

copyprivate 子句可以关联 single 构造，在 single 构造的 barrier 到达之前就完成了广播工作。copyprivate 可以对 private 和 threadprivate 子句中的变量进行操作，但是当使用 single 构造时，copyprivate 的变量不能用于 private 和 firstprivate 子句中。

下面便是一个使用 copyprivate 的代码例子：

```
int counter = 0;
#pragma omp threadprivate(counter)
int increment_counter()
{
        counter++;
        return(counter);
}
#pragma omp parallel
        {
                int     count;
#pragma omp single copyprivate(counter)
                {
                        counter = 50;
                }
                count = increment_counter();
                printf("ThreadId: %ld, count = %ld\n", omp_get_thread_num(), count);
}
```

打印结果为：

```
ThreadId: 2, count = 51
ThreadId: 0, count = 51
ThreadId: 3, count = 51
ThreadId: 1, count = 51
```

如果没有使用 copyprivate 子句，那么打印结果为：

ThreadId: 2, count = 1

ThreadId: 1, count = 1

ThreadId: 0, count = 51

ThreadId: 3, count = 1

从打印结果可以看出，使用 copyprivate 子句后，single 构造内给 counter 赋的值被广播到了其他线程里，但没有使用 copyprivate 子句时，只有一个线程获得了 single 构造内的赋值，其他线程没有获取 single 构造内的赋值。

OpenMP 中的任务调度

OpenMP 中，任务调度主要用于并行的 for 循环中，当循环中每次迭代的计算量不相等时，如果简单地给各个线程分配相同次数的迭代的话，会造成各个线程计算负载不均衡，这会使得有些线程先执行完，有些后执行完，造成某些 CPU 核空闲，影响程序性能。例如以下代码：

```
int i, j;
int a[100][100] = {0};
for ( i =0; i < 100; i++)
{
for( j = i; j < 100; j++ )
{
    a[i][j] = i*j;
}
}
```

如果将最外层循环并行化的话，比如使用 4 个线程，如果给每个线程平均分配 25 次循环迭代计算的话，显然 i＝0 和 i＝99 的计算量相差了 100 倍，那么各个线程间可能出现较大的负载不平衡情况。为了解决这些问题，OpenMP 中提供了几种对 for 循环并行化的任务调度方案。

在 OpenMP 中，对 for 循环并行化的任务调度使用 schedule 子句来实现，下面介绍 schedule 字句的用法。

1. Schedule 子句用法

schedule 子句的使用格式为：

schedule(type[,size])

schedule 有两个参数：type 和 size，size 参数是可选的。

1. type 参数

表示调度类型，有四种调度类型如下：

- dynamic
- guided
- runtime
- static

这四种调度类型实际上只有 static、dynamic、guided 三种调度方式，runtime 实际上是根据环境变量来选择前三种中的某中类型。

run-sched-var

2. size 参数 (可选)

size 参数表示循环迭代次数，size 参数必须是整数。static、dynamic、guided 三种调度方式都可以使用 size 参数，也可以不使用 size 参数。当 type 参数类型为 runtime 时，size 参数是非法的（不需要使用，如果使用的话编译器会报错）。

2.静态调度(static)

当 parallel for 编译指导语句没有带 schedule 子句时，大部分系统中默认采用 static 调度方式，这种调度方式非常简单。假设有 n 次循环迭代，t 个线程，那么给每个线程静态分配大约 n/t

次迭代计算。这里为什么说大约分配 n/t 次呢？因为 n/t 不一定是整数，因此实际分配的迭代次数可能存在差 1 的情况，如果指定了 size 参数的话，那么可能相差一个 size。

静态调度时可以不使用 size 参数，也可以使用 size 参数。

不使用 size 参数

不使用 size 参数时，分配给每个线程的是 n/t 次连续的迭代，不使用 size 参数的用法如下：

schedule(static)

例如以下代码:

```
#pragma omp parallel for schedule(static)
for(i = 0; i < 10; i++ )
{
    printf("i=%d, thread_id=%d\n", i, omp_get_thread_num());
}
```

上面代码执行时打印的结果如下：

```
i=0, thread_id=0
i=1, thread_id=0
i=2, thread_id=0
i=3, thread_id=0
i=4, thread_id=0
i=5, thread_id=1
i=6, thread_id=1
i=7, thread_id=1
i=8, thread_id=1
i=9, thread_id=1
```

可以看出线程 0 得到了 0～4 次连续迭代，线程 1 得到 5～9 次连续迭代。注意由于多线程执行时序的随机性，每次执行时打印的结果顺序可能存在差别，后面的例子也一样。

使用 size 参数

使用 size 参数时，分配给每个线程的 size 次连续的迭代计算，用法如下：

schedule(static, size)

例如以下代码:

```
#pragma omp parallel for schedule(static, 2)
for(i = 0; i < 10; i++ )
{
    printf("i=%d, thread_id=%d\n", i, omp_get_thread_num());
}
```

执行时会打印以下结果：

```
i=0, thread_id=0
i=1, thread_id=0
i=4, thread_id=0
i=5, thread_id=0
i=8, thread_id=0
i=9, thread_id=0
```

i=2, thread_id=1
i=3, thread_id=1
i=6, thread_id=1
i=7, thread_id=1
从打印结果可以看出，0、1 次迭代分配给线程 0，2、3 次迭代分配给线程 1，4、5 次迭代分配给线程 0，6、7 次迭代分配给线程 1，…。每个线程依次分配到 2 次连续的迭代计算。

3.动态调度(dynamic)

动态调度是动态地将迭代分配到各个线程，动态调度可以使用 size 参数也可以不使用 size 参数，不使用 size 参数时是将迭代逐个地分配到各个线程，使用 size 参数时，每次分配给线程的迭代次数为指定的 size 次。
下面为使用动态调度不带 size 参数的例子：

```
#pragma omp parallel for schedule(dynamic)
for(i = 0; i < 10; i++ )
{
        printf("i=%d, thread_id=%d\n", i, omp_get_thread_num());
}
```

打印结果如下：
i=0, thread_id=0
i=1, thread_id=1
i=2, thread_id=0
i=3, thread_id=1
i=5, thread_id=1
i=6, thread_id=1
i=7, thread_id=1
i=8, thread_id=1
i=4, thread_id=0
i=9, thread_id=1

下面为动态调度使用 size 参数的例子：

```
#pragma omp parallel for schedule(dynamic, 2)
for(i = 0; i < 10; i++ )
{
        printf("i=%d, thread_id=%d\n", i, omp_get_thread_num());
}
```

打印结果如下：
i=0, thread_id=0
i=1, thread_id=0
i=4, thread_id=0
i=2, thread_id=1
i=5, thread_id=0
i=3, thread_id=1
i=6, thread_id=0

i=8, thread_id=1
i=7, thread_id=0
i=9, thread_id=1
从打印结果可以看出第 0、1，4、5，6、7 次迭代被分配给了线程 0，第 2、3，8、9 次迭代则分配给了线程 1，每次分配的迭代次数为 2。

4.guided 调度（guided）

guided 调度是一种采用指导性的启发式自调度方法。开始时每个线程会分配到较大的迭代块，之后分配到的迭代块会逐渐递减。迭代块的大小会按指数级下降到指定的 size 大小，如果没有指定 size 参数，那么迭代块大小最小会降到 1。
例如以下代码：

```
    #pragma omp parallel for schedule(guided,2)
    for(i = 0; i < 10; i++ )
    {
        printf("i=%d, thread_id=%d\n", i, omp_get_thread_num());
}
```

打印结果如下：
i=0, thread_id=0
i=1, thread_id=0
i=2, thread_id=0
i=3, thread_id=0
i=4, thread_id=0
i=8, thread_id=0
i=9, thread_id=0
i=5, thread_id=1
i=6, thread_id=1
i=7, thread_id=1
第 0、1、2、3、4 次迭代被分配给线程 0，第 5、6、7 次迭代被分配给线程 1，第 8、9 次迭代被分配给线程 0，分配的迭代次数呈递减趋势，最后一次递减到 2 次。

5.runtime 调度（rumtime）

runtime 调度并不是和前面三种调度方式似的真实调度方式，它是在运行时根据环境变量 OMP_SCHEDULE 来确定调度类型，最终使用的调度类型仍然是上述三种调度方式中的某种。
例如在 unix 系统中，可以使用 setenv 命令来设置 OMP_SCHEDULE 环境变量：
setenv OMP_SCHEDULE “dynamic, 2”
上述命令设置调度类型为动态调度，动态调度的迭代次数为 2。
在 windows 环境中，可以在”系统属性|高级|环境变量”对话框中进行设置环境变量。

OpenMP 创建线程中的锁及原子操作性能比较

在多核 CPU 中锁竞争到底会造成性能怎样的下降呢？相信这是许多人想了解的，因此特地写了一个测试程序来测试原子操作，windows CriticalSection， OpenMP 的锁操作函数在多核 CPU 中的性能。

原子操作选用 InterlockedIncrement 来进行测试，
对每种锁和原子操作，都测试在单任务执行和多任务执行 2000000 次加锁解锁操作所消耗的时间。
测试的详细代码见后面。

测试机器环境： Intel 2.66G 双核 CPU 机器一台

测试运行结果如下：
SingleThread, InterlockedIncrement 2,000,000: a = 2000000, time = 78
MultiThread, InterlockedIncrement 2,000,000: a = 2000000, time = 156
SingleThread, Critical_Section 2,000,000:a = 2000000, time = 172
MultiThread, Critical_Section, 2,000,000:a = 2000000, time = 3156
SingleThread,omp_lock 2,000,000:a = 2000000, time = 250
MultiThread,omp_lock 2,000,000:a = 2000000, time = 1063

在单任务运行情况下，所消耗的时间如下：
原子操作 78ms
Windows CriticalSection 172ms
OpenMP 的 lock 操作 250ms

因此从单任务情况来看，原子操作最快，Windows CriticalSection 次之，OpenMP 库带的锁最慢，但这几种操作的时间差距不是很大，用锁操作比原子操作慢了 2～3 倍左右。

在多个任务运行的情况下，所消耗的时间如下：

原子操作 156ms
Windows CriticalSection 3156ms
OpenMP 的 lock 操作 1063ms

在多任务运行情况下，情况发生了意想不到的变化，原子操作时间比单任务操作时慢了一倍，在两个 CPU 上运行比在单个 CPU 上运行还慢一倍，真是难以想象，估计是任务切换开销造成的。
Windows CriticalSection 则更离谱了，居然花了 3156ms，是单任务运行时的 18 倍多的时间，慢得简直无法想象。
OpenMP 的 lock 操作比 Windows CriticalSection 稍微好一些，但也花了 1063ms，是单任务时的 7 倍左右。

由此可以知道，在多核 CPU 的多任务环境中，原子操作是最快的，而 OpenMP 次之，Windows CriticalSection 则最慢。

同时从这些锁在单任务和多任务下的性能差距可以看出，，多核 CPU 上的编程和以往的单核多任务编程会有很大的区别。
需要说明的是，本测试是一种极端情况下的测试，锁住的操作只是一个简单的加 1 操作，并且锁竞争次数达 200 万次之多，在实际情况中，一由于任务中还有很多不需要加锁的代码在运行，实际情况中的性能会比本测试的性能好很多。

测试代码如下：

```
// TestLock.cpp : OpenMP 任务中的原子操作和锁性能测试程序。
//

#include <windows.h>
#include <time.h>
#include <process.h>
#include <omp.h>
#include <stdio.h>

void TestAtomic()
{
	clock_t t1,t2;
	int		i = 0;
	volatile LONG		a = 0;

	t1 = clock();

	for( i = 0; i < 2000000; i++ )
	{
		InterlockedIncrement( &a);
	}

	t2 = clock();
	printf("SingleThread, InterlockedIncrement 2,000,000: a = %ld, time = %ld\n", a, t2-t1);

	t1 = clock();

#pragma omp parallel for
	for( i = 0; i < 2000000; i++ )
	{
		InterlockedIncrement( &a);
	}

	t2 = clock();
	printf("MultiThread, InterlockedIncrement 2,000,000: a = %ld, time = %ld\n", a, t2-t1);
```

```
}

void TestOmpLock()
{
	clock_t t1,t2;
	int i;
	int a = 0;
	omp_lock_t	mylock;

	omp_init_lock(&mylock);

	t1 = clock();

	for( i = 0; i < 2000000; i++ )
	{
		omp_set_lock(&mylock);
		a+=1;
		omp_unset_lock(&mylock);
	}
	t2 = clock();

	printf("SingleThread,omp_lock 2,000,000:a = %ld, time = %ld\n", a, t2-t1);

	t1 = clock();

#pragma omp parallel for
	for( i = 0; i < 2000000; i++ )
	{
		omp_set_lock(&mylock);
		a+=1;
		omp_unset_lock(&mylock);
	}
	t2 = clock();

	printf("MultiThread,omp_lock 2,000,000:a = %ld, time = %ld\n", a, t2-t1);

	omp_destroy_lock(&mylock);
}



void TestCriticalSection()
{
	clock_t t1,t2;
```

```
	int i;
	int a = 0;
	CRITICAL_SECTION	cs;

	InitializeCriticalSection(&cs);

	t1 = clock();

	for( i = 0; i < 2000000; i++ )
	{
		EnterCriticalSection(&cs);
		a+=1;
		LeaveCriticalSection(&cs);
	}
	t2 = clock();

	printf("SingleThread, Critical_Section 2,000,000:a = %ld, time = %ld\n", a, t2-t1);

	t1 = clock();

#pragma omp parallel for
	for( i = 0; i < 2000000; i++ )
	{
		EnterCriticalSection(&cs);
		a+=1;
		LeaveCriticalSection(&cs);
	}
	t2 = clock();

	printf("MultiThread, Critical_Section, 2,000,000:a = %ld, time = %ld\n", a, t2-t1);

	DeleteCriticalSection(&cs);

}

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

	TestAtomic();
	TestCriticalSection();
	TestOmpLock();

	return 0;
}
```

OpenMP 程序设计的两个小技巧

1、 动态设置并行循环的线程数量

在实际情况中，程序可能运行在不同的机器环境里，有些机器是双核，有些机器是 4 核甚至更多核。并且未来硬件存在升级的可能，CPU 核数会变得越来越多。如何根据机器硬件的不同来自动设置合适的线程数量就显得很重要了，否则硬件升级后程序就得进行修改，那将是一件很麻烦的事情。

比如刚开始在双核系统中开发的软件，线程数量缺省都设成 2，那么当机器升级到 4 核或 8 核以后，线程数量就不能满足要求了，除非修改程序。

线程数量的设置除了要满足机器硬件升级的可扩展性外，还需要考虑程序的可扩展性，当程序运算量增加或减少后，设置的线程数量仍然能够满足要求。显然这也不能通过设置静态的线程数量来解决。

在具体计算需要使用多少线程时，主要需要考虑以下两点：

1)当循环次数比较少时，如果分成过多数量的线程来执行，可能会使得总运行时间高于较少线程或一个线程执行的情况。并且会增加能耗。

2)如果设置的线程数量远大于 CPU 核数的话，那么存在着大量的任务切换和调度等开销，也会降低整体效率。

那么如何根据循环的次数和 CPU 核数来动态地设置线程的数量呢？下面以一个例子来说明动态设置线程数量的算法，假设一个需要动态设置线程数的需求为：

1.以多个线程运行时的每个线程运行的循环次数不低于 4 次

2.总的运行线程数最大不超过 2 倍 CPU 核数

下面代码便是一个实现上述需求的动态设置线程数量的例子

```
    const int MIN_ITERATOR_NUM = 4;
    int ncore = omp_get_num_procs(); //获取执行核的数量
    int max_tn = n / MIN_ITERATOR_NUM;
    int tn = max_tn > 2*ncore ? 2*ncore : max_tn; //tn 表示要设置的线程数量
#pragma omp parallel for if( tn > 1) num_threads(tn)
      for ( i = 0; i < n; i++ )
      {
            printf("Thread Id = %Id\n", omp_get_thread_num());
            //Do some work here
      }
```

在上面代码中，根据每个线程运行的循环次数不低于 4 次，先计算出最大可能的线程数 max_tn，然后计算需要的线程数量 tn，tn 的值等于 max_tn 和 2 倍 CPU 核数中的较小值。然后在 parallel for 构造中使用 if 子句来判断 tn 是否大于 1，大于 1 时使用单个线程，否则使用 tn 个线程，，这样就使得设置的线程数量满足了需求中的条件。

比如在一个双核 CPU 上，n=64，最终会以 2 倍 CPU 核数（4 个）线程运行，而不会以 max_tn = 64/4＝16 个线程运行。

在实际情况中，当然不能每个循环都象上面一样写几行代码来计算一遍，可以将其写成一个独立的功能函数如下：

```
const int g_ncore = omp_get_num_procs(); //获取执行核的数量

/** 计算循环迭代需要的线程数量
      根据循环迭代次数和 CPU 核数及一个线程最少需要的循环迭代次数
```

来计算出需要的线程数量，计算出的最大线程数量不超过 CPU 核数

```
    @param     int n -  循环迭代次数
    @param     int min_n -  单个线程需要的最少迭代次数
    @return int -  线程数量
*/
int dtn(int n, int min_n)
{
    int max_tn = n / min_n;
    int tn = max_tn > g_ncore ? g_ncore : max_tn; //tn 表示要设置的线程数量
    if ( tn < 1 )
    {
        tn = 1;
    }
    return tn;
}
```

这样每次并行化循环时就可以直接使用函数 dtn()来获取合适的线程数量，前面的代码可以简写成如下形式：

```
#pragma omp parallel for num_threads(dtn(n, MIN_ITERATOR_NUM))
    for ( i = 0; i < n; i++ )
    {
        printf("Thread Id = %ld\n", omp_get_thread_num());
        //Do some work here
    }
```

当然具体设置多少线程要视情况而定的，一般情况下线程数量刚好等于 CPU 核数可以取得比较好的性能，因为线程数等于 CPU 核数时，每个核执行一个任务，没有任务切换开销。

2、嵌套循环的并行化

在嵌套循环中，如果外层循环迭代次数较少时，如果将来 CPU 核数增加到一定程度时，创建的线程数将可能小于 CPU 核数。另外如果内层循环存在负载平衡的情况下，很难调度外层循环使之达到负载平衡。

下面以矩阵乘法作为例子来讲述如何将嵌套循环并行化，以满足上述扩展性和负载平衡需求。

一个串行的矩阵乘法的函数代码如下：

```
/**  矩阵串行乘法函数
    @param     int *a -  指向要相乘的第个矩阵的指针
    @param     int row_a -  矩阵 a 的行数
    @param     int col_a -  矩阵 a 的列数
    @param     int *b -  指向要相乘的第个矩阵的指针
    @param     int row_b -  矩阵 b 的行数
    @param     int col_b -  矩阵 b 的列数
    @param     int *c -  计算结果的矩阵的指针
```

```
    @param      int c_size - 矩阵 c 的空间大小（总元素个数）
    @return     void - 无
*/
void Matrix_Multiply(int *a, int row_a, int col_a,
                                int *b, int row_b,int col_b,
                                int *c, int c_size)
{
    if ( col_a != row_b || c_size < row_a * col_b )
    {
        return;
    }

    int i, j, k;
//#pragma omp for private(i, j, k)
    for ( i = 0; i < row_a; i++ )
    {
        int row_i = i * col_a;
        int row_c = i * col_b;
        for ( j = 0; j < col_b; j++ )
        {
            c[row_c + j] = 0;
             for ( k = 0; k < row_b; k++ )
             {
                 c[row_c + j] += a[row_i + k] * b[k * col_b + j];
             }
        }
    }
}
```

如果在外层循环前加上 OpenMP 的 for 语句时，它就变成了一个并行的矩阵乘法函数，但是这样简单地将其并行化显然无法满足前面所述的扩展性需求。

其实可以采用一个简单的方法将最外层循环和第 2 层循环合并成一个循环，下面便是采用合并循环后的并行实现。

```
void Parallel_Matrix_Multiply(int *a, int row_a, int col_a,
                                int *b, int row_b,int col_b,
                                int *c, int c_size )
{
    if ( col_a != row_b )
    {
        return;
    }

    int i, j, k;
    int index;
```

```
    int border = row_a * col_b;

    i = 0;
    j = 0;
#pragma omp parallel private(i,j,k) num_threads(dtn(border, 1))
    for ( index = 0; index < border; index++ )
    {
        i = index / col_b;
        j = index % col_b;

        int row_i = i * col_a;
        int row_c = i * col_b;

        c[row_c+j] = 0;
        for ( k = 0; k < row_b; k++ )
        {
            c[row_c + j] += a[row_i+k] * b[k*col_b+j];
        }
    }
}
```

从上面代码可以看出，合并后的循环边界 border = row_a * col_b;即等于原来两个循环边界之积，然后在循环中计算出原来的外层循环和第 2 层循环的迭代变量 i 和 j，采用除法和取余来求出 i 和 j 的值。

需要注意的是，上面求 i 和 j 的值必须要保证循环迭代的独立性，即不能有循环迭代间的依赖关系。不能将求 i 和 j 值的过程优化成如下的形式：

```
if ( j == col_b )
{
    j = 0;
    i++;
}
// …… 此处代表实际的矩阵乘法代码
j++;
```

上面这种优化，省去了除法，效率高，但是只能在串行代码中使用，因为它存在循环迭代间的依赖关系，无法将其正确地并行化。