

# 亮亮的园子

一个具有学术气质的IT技术博客

## OpenMP共享内存并行编程详解

实验平台：win7，VS2010

### 1. 介绍

并行计算机可以简单分为共享内存和分布式内存，共享内存就是多个核心共享一个内存，目前的PC就是这类（不管是只有一个多核CPU还是可以插多个CPU，它们都有多个核心和一个内存），一般的大型计算机结合分布式内存和共享内存结构，即每个计算节点内是共享内存，节点间是分布式内存。想要在这些并行计算机上获得较好的性能，进行并行编程是必要条件。目前流行的并行程序设计方法是，分布式内存结构上使用MPI，共享内存结构上使用Pthreads或OpenMP。我们这里关注的是共享内存并行计算机，因为编辑这篇文章的机器就属于此类型（普通的台式机）。和Pthreads相比OpenMP更简单，对于关注算法、只要求对线程之间关系进行最基本控制（同步，互斥等）的我们来说，OpenMP再适合不过了。

本文对windows上Visual Studio开发环境下的OpenMP并行编程进行简单的探讨。本文参考了wikipedia关于OpenMP条目、OpenMP.org（有OpenMP Specification）、MSDM上关于OpenMP条目以及教材《MPI与OpenMP并行程序设计（C语言版）》：

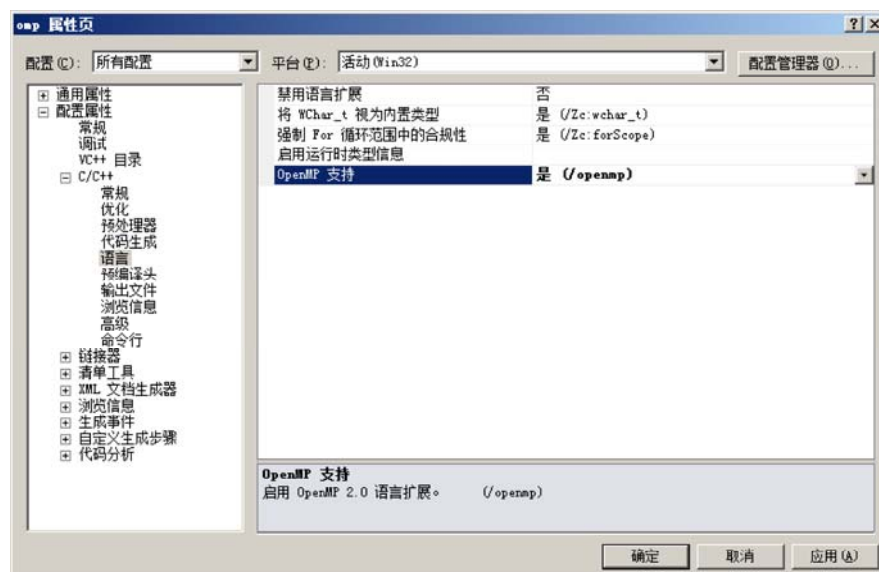
1. <http://zh.wikipedia.org/wiki/OpenMP>
2. <http://openmp.org/>
3. [http://msdn.microsoft.com/en-us/library/tt15eb9t\(v=vs.100\).aspx](http://msdn.microsoft.com/en-us/library/tt15eb9t(v=vs.100).aspx)
4. 《MPI与OpenMP并行程序设计（C语言版）》第17章，Michael J. Quinn著，陈文光等译，清华大学出版社，2004

注意，OpenMP目前最新版本为4.0.0，而VS2010仅支持OpenMP2.0（2002年版本），所以本文所讲的也是OpenMP2.0，本文注重使用OpenMP获得接近核心数的加速比，所以OpenMP2.0也足够了。

### 2. 第一个OpenMP程序

step 1：新建控制台程序

step 2：项目属性，所有配置下“配置属性>>C/C++>>语言>>OpenMP支持”修改为是（/openmp），如下图：



step 3：添加如下代码：

按 Ctrl+C 复制代码

```
#include<omp.h>
#include<iostream>
int main()
{
    std::cout << "parallel begin:\n";
    #pragma omp parallel
    {
        std::cout << omp_get_thread_num();
    }
    std::cout << "\n parallel end.\n";
    std::cin.get();
    return 0;
}
```

按 Ctrl+C 复制代码

step 4：运行结果如下图：

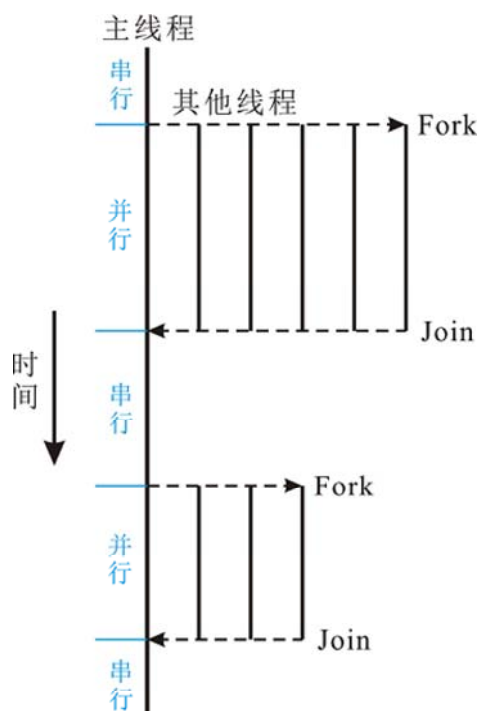
```
parallel begin:
03251647
parallel end.
```

可以看到，我的计算机是8核的（严格说是8线程的），这是我们实验室的小型工作站（至多支持24核）。

### 3. “第一个OpenMP程序”幕后，并行原理

OpenMP由Compiler Directives（编译指导语句）、Run-time Library Functions（库函数）组成，另外还有一些和OpenMP有关的Environment Variables（环境变量）、Data Types（数据类型）以及\_OPENMP宏定义。之所以说OpenMP非常简单，是因为，所有这些总共只有50个左右，OpenMP2.0 Specification仅有100余页。第2节的“第一个OpenMP程序”的第6行“#pragma omp parallel”即Compiler Directive，“#pragma omp parallel”下面的语句将被多个线程并行执行（也即被执行不止一遍），第8行的omp\_get\_thread\_num()即Run-time Library Function，omp\_get\_thread\_num()返回当前执行代码所在线程编号。

共享内存计算机上并行程序的基本思路就是使用多线程，从而将可并行负载分配到多个物理计算核心，从而缩短执行时间（同时提高CPU利用率）。在共享内存的并行程序中，标准的并行模式为fork/join式并行，这个基本模型如下图所示：



其中，主线程执行算法的顺序部分，当遇到需要进行并行计算式，主线程派生出（创建或者唤醒）一些附加线程。在并行区域内，主线程和这些派生线程协同工作，在并行代码结束时，派生的线程退出或者挂起，同时控制流回到单独的主线程中，称为汇合。对应第2节的“第一个OpenMP程序”，第4行对应程序开始，4-5行对应串行部分，6-9行对应第一个并行块（8个线程），10-13行对应串行部分，13行对应程序结束。

简单来说，OpenMP程序就是在一般程序代码中加入Compiler Directives，这些Compiler Directives指示编译器其后的代码应该如何处理（是多线程执行还是同步什么的）。所以说OpenMP需要编译器的支持。上一小节的step 2即打开编译器的OpenMP支持。和Pthreads不同，OpenMP下程序员只需要设计高层并行结构，创建及调度线程均由编译器自动生成代码完成。

## 4. Compiler Directives

### 4.1 一般格式

Compiler Directive的基本格式如下：

```
#pragma omp directive-name [clause[ [ ] clause]...]
```

其中“[]”表示可选，每个Compiler Directive作用于其后的语句（C++中“{}”括起来部分是一个复合语句）。

directive-name可以为：parallel, for, sections, single, atomic, barrier, critical, flush, master, ordered, threadprivate（共11个，只有前4个有可选的clause）。

clause（子句）相当于Directive的修饰，定义一些Directive的参数什么的。clause可以为：copyin(variable-list), copyprivate(variable-list), default(shared | none), firstprivate(variable-list), if(expression), lastprivate(variable-list), nowait, num\_threads(num), ordered, private(variable-list), reduction(operation: variable-list), schedule(type[,size]), shared(variable-list)（共13个）。

例如“#pragma omp parallel”表示其后语句将被多个线程并行执行，线程个数由系统预设（一般等于逻辑处理器个数，例如i5 4核8线程CPU有8个逻辑处理器），可以在该directive中加入可选的clauses，如“#pragma omp parallel num\_threads(4)”仍旧表示其后语句将被多个线程并行执行，但是线程个数为4。

### 4.2 详细解释

本节的叙述顺序同我的另一篇博文：[OpenMP编程总结表](#)，读者可以对照阅读，也可以快速预览OpenMP所有语法。

如果没有特殊说明，程序均在Debug下编译运行。

#### parallel

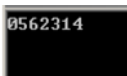
parallel表示其后语句将被多个线程并行执行，这已经知道了。“#pragma omp parallel”后面的语句（或者，语句块）被称为parallel region。

可以用if clause条件地进行并行化，用num\_threads clause覆盖默认线程数：

```
1 int a = 0;
2 #pragma omp parallel if(a) num_threads(6)
3 {
4     std::cout << omp_get_thread_num();
5 }
```



```
int a = 7;
#pragma omp parallel if(a) num_threads(6)
{
    std::cout << omp_get_thread_num();
}
```



可以看到多个线程的执行顺序是不能保证的。

private, firstprivate, shared, default, reduction, copyin clauses留到threadprivate directive时说。

#### for

第2节的“第一个OpenMP程序”其实不符合我们对并行程序的预期——我们一般并不是要对相同代码在多个线程并行执行，而是，对一个计算量庞大的任务，对其进行划分，让多个线程分别执行计算任务的每一部分，从而达到缩短计算时间的目的。这里的关键是，每个线程执行的计算互不相同（操作的数据不同或者计算任务本身不同），多个线程协作完成所有计算。OpenMP for指示将C++ for循环的多次迭代划分给多个线程（划分指，每个线程执行的迭代互不重复，所有线程的迭代加起来正好是C++ for循环的所有迭代），这里C++ for循环需要一些限制从而能在执行C++ for之前确定循环次数，例如C++ for中不应含有break等。OpenMP for作用于其后的第一层C++ for循环。下面是一个例子：

```
1 const int size = 1000;
2 int data[size];
3 #pragma omp parallel
4 {
5     #pragma omp for
6     for(int i=0; i<size; ++i)
7         data[i] = 123;
8 }
```

默认情况下，上面的代码中，程序执行到“#pragma omp parallel”处会派生出7个线程，加上主线程共8个线程（在我的机器上），C++ for的1000次迭代会被分成连续的8段——0-124次迭代由0号线程计算，125-249次迭代由1号线程计算，以此类推。可能你已经猜到了，具体C++ for的各次迭代在线程间如何分配可以由clause指示，它就是schedule(type[,size])，后面会具体说。

如果parallel region中只包含一个for directive作用的语句，上面代码就是这种情况，此时可以将parallel和for“缩写”为**parallel for**，上面代码等价于这样：

```
1 const int size = 1000;
2 int data[size];
3 #pragma omp parallel for
4 for(int i=0; i<size; ++i)
5     data[i] = 123;
```

正确使用for directive有两个条件，第1是C++ for符合特定限制，否则编译器将报告错误，第2是C++ for的各次迭代的执行顺序不影响结果正确性，这是一个逻辑条件。例子如下：

```
1 #pragma omp parallel num_threads(6)
2 {
3     #pragma omp for
4     for(int i=0; i<1000000; ++i)
5         if(i>999)
6             break;
7 }
```

编译器报错如下：

error C3010: “break”：不允许跳出 OpenMP 结构化

块 **schedule**(type[,size])设置C++ for的多次迭代如何在多个线程间划分：

1. schedule(static, size)将所有迭代按每连续size个为一组，然后将这些组轮转分给各个线程。例如有4个线程，100次迭代，schedule(static, 5)将迭代：0-4, 5-9, 10-14, 15-19, 20-24...依次分给0, 1, 2, 3, 0...号线程。schedule(static)同schedule(static, size\_av)，其中size\_av等于迭代次数除以线程数，即将迭代分成连续的和线程数相同的等分（或近似等分）。
2. schedule(dynamic, size)同样分组，然后依次将每组分给目前空闲的线程（故叫动态）。
3. schedule(guided, size)把迭代分组，分配给目前空闲的线程，最初组大小为迭代数除以线程数，然后逐渐按指数方式（依次除以2）下降到size。
4. schedule(runtime)的划分方式由环境变量OMP\_SCHEDULE定义。

下面是几个例子，可以先忽略critical directive：

```
1 #pragma omp parallel num_threads(3)
2 {
3     #pragma omp for
4     for(int i=0; i<9; ++i){
5         #pragma omp critical
6         std::cout << omp_get_thread_num() << i << " ";
7     }
8 }
```

```
00 01 02 26 27 28 13 14 15
```

上面输出说明0号线程执行0-2迭代，1号执行3-5,2号执行6-9，相当于schedule(static, 3)。

```
1 #pragma omp parallel num_threads(3)
2 {
3     #pragma omp for schedule(static, 1)
4     for(int i=0; i<9; ++i){
5         #pragma omp critical
6         std::cout << omp_get_thread_num() << i << " ";
7     }
8 }
```

```
00 03 06 11 14 17 22 25 28
```

```
1 #pragma omp parallel num_threads(3)
2 {
3     #pragma omp for schedule(dynamic, 2)
4     for(int i=0; i<9; ++i){
5         #pragma omp critical
6         std::cout << omp_get_thread_num() << i << " ";
7     }
8 }
```

```
00 01 04 05 06 12 13 28 07
```

ordered clause配合ordered directive使用，请见ordered directive，nowait留到barrier directive时说，private, firstprivate, lastprivate, reduction留到threadprivate directive时说。

### sections

如果说for directive用作数据并行，那么sections directive用于任务并行，它指示后面的代码块包含将被多个线程并行执行的section块。下面是一个例子：

```
1 #pragma omp parallel
2 {
3     #pragma omp sections
4     {
5         #pragma omp section
6         std::cout << omp_get_thread_num();
7         #pragma omp section
8         std::cout << omp_get_thread_num();
9     }
10 }
```

```

9     }
10  }

```



```
04_
```

上面代码中2个section块将被2个线程并行执行，多个个section块的第1个 “#pragma omp section” 可以省略。这里有些问题，执行这段代码是总共会有多少个线程呢，“#pragma omp parallel” 没有clause，默认是8个线程（又说的在我的机器上），2个section是被哪2个线程执行是不确定的，当section块多于8个时，会有一个线程执行不止1个section块。

同样，上面代码可以“缩写”为**parallel sections**：



```

1 #pragma omp parallel sections
2 {
3     #pragma omp section
4     std::cout << omp_get_thread_num();
5     #pragma omp section
6     std::cout << omp_get_thread_num();
7 }

```



nowait clause留到barrier directive时说，private, firstprivate, lastprivate, reduction clauses留到threadprivate directive时说。

### single

指示代码将仅被一个线程执行，具体是哪个线程不确定，例子如下：



```

1 #pragma omp parallel num_threads(4)
2 {
3     #pragma omp single
4     std::cout << omp_get_thread_num();
5     std::cout << "- ";
6 }

```



```
0_
```

这里0号线程执行了第4 5两行代码，其余三个线程执行了第5行代码。

nowait clause留到barrier directive时说，private, firstprivate, copyprivate clauses留到threadprivate directive时说。

### master

指示代码将仅被主线程执行，功能类似于single directive，但single directive时具体是哪个线程不确定（有可能是当时闲的那个）。

### critical

定义一个临界区，保证同一时刻只有一个线程访问临界区。观察如下代码及其结果：

```

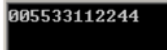
1 #pragma omp parallel num_threads(6)
2 {
3     std::cout << omp_get_thread_num() << omp_get_thread_num();
4 }

```

```
001152254433
```

5号线程执行第3行代码时被2号线程打断了（并不是每次运行都可能出现打断）。


```
1 #pragma omp parallel num_threads(6)
2 {
3     #pragma omp critical
4     std::cout << omp_get_thread_num() << omp_get_thread_num();
5 }
```




这次不管运行多少遍都不会出现某个数字不是连续两个出现，因为在第4行代码被一个线程执行期间，其他线程不能执行（该行代码是临界区）。

### barrier

定义一个同步，所有线程都执行到该行后，所有线程才继续执行后面的代码，请看例子：



```
1 #pragma omp parallel num_threads(6)
2 {
3     #pragma omp critical
4     std::cout << omp_get_thread_num() << " ";
5     #pragma omp critical
6     std::cout << omp_get_thread_num()+10 << " ";
7 }
```




```
1 #pragma omp parallel num_threads(6)
2 {
3     #pragma omp critical
4     std::cout << omp_get_thread_num() << " ";
5     #pragma omp barrier
6     #pragma omp critical
7     std::cout << omp_get_thread_num()+10 << " ";
8 }
```



可以看到，这时一位数数字打印完了才开始打印两位数数字，因为，所有线程执行到第5行代码时，都要等待所有线程都执行到第5行，这时所有线程再都继续执行第7行及以后的代码，即所谓同步。

再来说说for, sections, single directives的隐含barrier，以及**nowait** clause如下示例：



```
1 #pragma omp parallel num_threads(6)
2 {
3     #pragma omp for
4     for(int i=0; i<10; ++i){
5         #pragma omp critical
6         std::cout << omp_get_thread_num() << " ";
7     }
8     // There is an implicit barrier here.
9     #pragma omp critical
10    std::cout << omp_get_thread_num()+10 << " ";
11 }
```



```
0 4 5 3 3 0 2 2 1 1 11 14 15 13 10 12
```

```
1 #pragma omp parallel num_threads(6)
2 {
3     #pragma omp for nowait
4     for(int i=0; i<10; ++i){
5         #pragma omp critical
6         std::cout << omp_get_thread_num() << " ";
7     }
8     // The implicit barrier here is disabled by nowait.
9     #pragma omp critical
10    std::cout << omp_get_thread_num()+10 << " ";
11 }
```

```
0 0 10 2 2 12 5 15 1 1 11 3 3 13 4 14
```

sections, single directives是类似的。

### atomic

atomic directive保证变量被原子的更新，即同一时刻只有一个线程再更新该变量（是不是很像critical directive），见例子：

```
1 int m=0;
2 #pragma omp parallel num_threads(6)
3 {
4     for(int i=0; i<1000000; ++i)
5         ++m;
6 }
7 std::cout << "value should be: " << 1000000*6 << std::endl;
8 std::cout << "value is: " << m << std::endl;
```

```
value should be: 6000000
value is: 3385451
```

m实际值比预期要小，因为“++m”的汇编代码不止一条指令，假设三条：load, inc, mov（读RAM到寄存器、加1，写回RAM），有可能线程A执行到inc时，线程B执行了load（线程A inc后的值还没写回），接着线程A mov，线程B inc后再mov，原本应该加2就变成了加1。

使用atomic directive后可以得到正确结果：

```
1 int m=0;
2 #pragma omp parallel num_threads(6)
3 {
4     for(int i=0; i<1000000; ++i)
5         #pragma omp atomic
6         ++m;
7 }
8 std::cout << "value should be: " << 1000000*6 << std::endl;
9 std::cout << "value is: " << m << std::endl;
```

```
value should be: 6000000
value is: 6000000
```



那用critical directive行不行呢：

```
1 int m=0;
2 #pragma omp parallel num_threads(6)
3 {
4     for(int i=0; i<1000000; ++i)
5         #pragma omp critical
6             ++m;
7 }
8 std::cout << "value should be: " << 1000000*6 << std::endl;
9 std::cout << "value is: " << m << std::endl;
```

```
value should be: 6000000
value is: 6000000
```

差别为何呢，显然是效率啦，我们做个定量分析：

```
1 #pragma omp parallel num_threads(6)
2 {
3     for(int i=0; i<1000000; ++i) ;
4 }
5 int m;
6 double t, t2;
7 m = 0;
8 t = omp_get_wtime();
9 #pragma omp parallel num_threads(6)
10 {
11     for(int i=0; i<1000000; ++i)
12         ++m;
13 }
14 t2 = omp_get_wtime();
15 std::cout << "value should be: " << 1000000*6 << std::endl;
16 std::cout << "value is: " << m << std::endl;
17 std::cout << "time(S): " << t2-t << std::endl;
18 m = 0;
19 t = omp_get_wtime();
20 #pragma omp parallel num_threads(6)
21 {
22     for(int i=0; i<1000000; ++i)
23         #pragma omp critical
24             ++m;
25 }
26 t2 = omp_get_wtime();
27 std::cout << "value should be: " << 1000000*6 << std::endl;
28 std::cout << "value is: " << m << std::endl;
29 std::cout << "time of critical(S): " << t2-t << std::endl;
30 m = 0;
31 t = omp_get_wtime();
32 #pragma omp parallel num_threads(6)
33 {
34     for(int i=0; i<1000000; ++i)
35         #pragma omp atomic
36             ++m;
37 }
38 t2 = omp_get_wtime();
39 std::cout << "value should be: " << 1000000*6 << std::endl;
40 std::cout << "value is: " << m << std::endl;
41 std::cout << "time of atomic(S): " << t2-t << std::endl;
```

```
value should be: 6000000
value is: 1000000
time(S): 0.00621826
value should be: 6000000
value is: 6000000
time of critical(S): 0.974988
value should be: 6000000
value is: 6000000
time of atomic(S): 0.108478
=
```

按照惯例，需要列出机器配置：Intel Xeon Processor E5-2637 v2 (4核8线程 15M Cache, 3.50 GHz)，16GB RAM。上面代码需要在Release下编译运行以获得更为真实的运行时间（实际部署的程序不可能是Debug版本的），第一个parallel directive的用意是跳过潜在的创建线程的步骤，让下面三个parallel directives有相同的环境，以增加可比性。从结果可以看出，没有atomic clause或critical clause时运行时间短了很多，可见正确性是用性能置换而来的。不出所料，“大材小用”的critical clause运行时间比atomic clause要长很多。

### flush

指示所有线程对所有共享对象具有相同的内存视图（view of memory），该directive指示将对变量的更新直接写回内存（有时候给变量赋值可能只改变了寄存器，后来才写回内存，这是编译器优化的结果）。这不好理解，看例子，为了让编译器尽情的优化代码，需要在Release下编译运行如下代码：

```
1 int data, flag=0;
2 #pragma omp parallel sections num_threads(2) shared(data, flag)
3 {
4     #pragma omp section // thread 0
5     {
6         #pragma omp critical
7         std::cout << "thread:" << omp_get_thread_num() << std::endl;
8         for(int i=0; i<10000; ++i)
9             ++data;
10        flag = 1;
11    }
12    #pragma omp section // thread 1
13    {
14        while(!flag) ;
15        #pragma omp critical
16        std::cout << "thread:" << omp_get_thread_num() << std::endl;
17        -- data;
18        std::cout << data << std::endl;
19    }
20 }
```

```
thread:0
```

程序进入了死循环..... 我们的初衷是，用flag来做手动同步，线程0修改data的值，修改好了置flag，线程1反复测试flag检查线程0有没有修改完data，线程1接着再修改data并打印结果。这里进入死循环的可能原因是，线程1反复测试的flag只是读到寄存器中的值，因为线程1认为，只有自己在访问flag（甚至以为只有自己这1个线程），在自己没有修改内存之前不需要重新去读flag的值到寄存器。用flush directive修改后：

```
1 int data=0, flag=0;
2 #pragma omp parallel sections num_threads(2) shared(data, flag)
3 {
4     #pragma omp section // thread 0
5     {
6         #pragma omp critical
7         std::cout << "thread:" << omp_get_thread_num() << std::endl;
8         for(int i=0; i<10000; ++i)
9             ++data;
10        #pragma omp flush(data)
```

```

11     flag = 1;
12     #pragma omp flush(flag)
13 }
14 #pragma omp section // thread 1
15 {
16     while(!flag){
17         #pragma omp flush(flag)
18     }
19     #pragma omp critical
20     std::cout << "thread:" << omp_get_thread_num() << std::endl;
21     #pragma omp flush(data)
22     -- data;
23     std::cout << data << std::endl;
24 }
25 }

```



```

thread:0
thread:1
9999

```

这回结果对了，解释一下，第10行代码告诉编译器，确保data的新值已经写回内存，第17行代码说，重新从内存读flag的值。

### ordered

使用在有**ordered** clause的for directive ( 或parallel for ) 中，确保代码将被按迭代次序执行（像串行程序一样），例子：



```

1 #pragma omp parallel num_threads(8)
2 {
3     #pragma omp for ordered
4     for(int i=0; i<10; ++i){
5         #pragma omp critical
6         std::cout << i << " ";
7         #pragma omp ordered
8         {
9             #pragma omp critical
10            std::cout << "-" << i << " ";
11        }
12    }
13 }

```



```

2 0 -0 1 -1 6 8 9 7 5 4 -2 3 -3 -4 -5 -6 -7 -8 -9 _

```

只看前面有“-”的数字，是不是按顺序的，而没有“-”的数字则没有顺序。值得强调的是for directive的ordered clause只是配合ordered directive使用，而不是让迭代有序执行的意思，后者的代码是这样的：

```

1 #pragma omp for ordered
2 for(int i=0; i<10; ++i)
3     #pragma omp ordered{
4     ; // all the C++ for code
5 }

```

### threadprivate

将全局或静态变量声明为线程私有的。为理解线程共享和私有变量，看如下代码：



```

1 int a;
2 std::cout << omp_get_thread_num() << ": " << &a << std::endl;

```

```

3 #pragma omp parallel num_threads(8)
4 {
5     int b;
6     #pragma omp critical
7     std::cout << omp_get_thread_num() << ": " << &a << " " << &b << std::endl;
8 }

```



```

0: 001AF850
0: 001AF850 001AF640
2: 001AF850 0223FAC8
4: 001AF850 0267FBCC
1: 001AF850 00C0F870
3: 001AF850 0249FA28
5: 001AF850 0286FD44
6: 001AF850 02A9FB08
7: 001AF850 02C7F7A0

```

记住第3-7行代码要被8个线程执行8遍，变量a是线程之间共享的，变量b是每个线程都有一个（在线程自己的栈空间）。

怎么区分哪些变量是共享的，哪些是私有的呢。在parallel region内定义的变量（非堆分配）当然是私有的。没有特别用clause指定的（上面代码就是这样），在parallel region前（parallel region后的不可见，这点和纯C++相同）定义的变量是共享的，在堆（用new或malloc函数分配的）上分配的变量是共享的（即使是在多个线程中使用new或malloc，当然指向这块堆内存的指针可能是私有的），for directive作用的C++ for的循环变量不管在哪里定义都是私有的。

好了，回到threadprivate directive，看例子：



```

1 #include<omp.h>
2 #include<iostream>
3 int a;
4 #pragma omp threadprivate(a)
5 int main()
6 {
7     std::cout << omp_get_thread_num() << ": " << &a << std::endl;
8     #pragma omp parallel num_threads(8)
9     {
10         int b;
11         #pragma omp critical
12         std::cout << omp_get_thread_num() << ": " << &a << " " << &b << std::endl;
13     }
14     std::cin.get();
15     return 0;
16 }

```



```

0: 003E41FC
0: 003E41FC 0025F780
4: 003EAC14 0233FC1C
3: 003EA97C 0220FA70
2: 003EA4C4 0204F99C
1: 003EA00C 01F1F6D0
5: 003EB5F4 024BFD20
6: 003EB0FC 0264FE84
7: 003EB85C 0284FACC

```

下面是最后几个没有讲的clauses：private, firstprivate, lastprivate, shared, default, reduction, copyin, copyprivate clauses，先看**private** clause：



```

1 int a = 0;
2 std::cout << omp_get_thread_num() << ": " << &a << std::endl;
3 #pragma omp parallel num_threads(8) private(a)

```

```

4 {
5     #pragma omp critical
6     std::cout << omp_get_thread_num() << " : * " << &a << " " << a << std::endl;
7 }

```



```

0: 0015F808
2: *00D1F8C8 -858993460
4: *0253FE84 -858993460
7: *02A8F8B4 -858993460
5: *026AFAC4 -858993460
6: *0285F864 -858993460
1: *00ADFB38 -858993460
0: *0015F600 -858993460
3: *023AF9DC -858993460
=

```

private clause将变量a由默认线程共享变为线程私有的，每个线程会调用默认构造函数生成一个变量a的副本（当然这里int没有构造函数）。

**firstprivate** clause和private clause的区别是，会用共享版本变量a来初始化。**lastprivate** clause在private基础上，将执行最后一次迭代（for）或最后一个section块（sections）的线程的私有副本拷贝到共享变量。**shared** clause和private clause相对，将变量声明为共享的。如下例子，其中的shared clause可以省略：

```

1 int a=10, b=11, c=12, d=13;
2 std::cout << "abcd's values: " << a << " " << b << " " << c << " " << d << std::endl;
3 #pragma omp parallel for num_threads(8) \
4     firstprivate(a) lastprivate(b) firstprivate(c) lastprivate(c) shared(d)
5 for(int i=0; i<8; ++i){
6     #pragma omp critical
7     std::cout << "thread " << omp_get_thread_num() << " acd's values: "
8         << a << " " << c << " " << d << std::endl;
9     a = b = c = d = omp_get_thread_num();
10 }
11 std::cout << "abcd's values: " << a << " " << b << " " << c << " " << d << std::endl;

```

```

abcd's values: 10 11 12 13
thread 1 acd's values: 10 12 13
thread 0 acd's values: 10 12 1
thread 2 acd's values: 10 12 0
thread 3 acd's values: 10 12 2
thread 4 acd's values: 10 12 3
thread 6 acd's values: 10 12 4
thread 7 acd's values: 10 12 6
thread 5 acd's values: 10 12 7
abcd's values: 10 7 7 5
=

```

每个线程都对a,b,c,d的值进行了修改。因为d是共享的，所以每个线程打印d前可能被其他线程修改了。parallel region结束，a的共享版本不变，b,c由于被lastprivate clause声明了，所以执行最后一次迭代的那个线程用自己的私有b,c更新了共享版本的b,c，共享版本d的值取决于那个线程最后更新d。

**default**(shared|none)：参数shared同于将所有变量用share clause定义，参数none指示对没有用private, shared, reduction, firstprivate, lastprivate clause定义的变量报错。

**reduction** clause用于归约，如下是一个并行求和的例子：

```

1 int sum=0;
2 std::cout << omp_get_thread_num() << "：" << &sum << std::endl << std::endl;
3 #pragma omp parallel num_threads(8) reduction(+:sum)
4 {
5     #pragma omp critical
6     std::cout << omp_get_thread_num() << "：" << &sum << std::endl;
7     #pragma omp for
8     for(int i=1; i<=10000; ++i){
9         sum += i;
10    }
11 }
12 std::cout << "sum's valuse: " << sum << std::endl;

```

```

0:0039FEA0
0:0039FC88
1:009EFBCC
4:0251F8EC
3:00D5FAA0
2:00BBF908
7:027DF844
5:026BF8CC
6:0295FAA8
sum's valuse: 50005000

```

可以看到变量sum在parallel region中是线程私有的，每个线程用自己的sum求一部分和，最后将所有线程的私有sum加起来赋值给共享版本的sum。只有 +, \*, -, &, |, ^, &&, || 八种。除法运算不满足结合率，无法规约。

**copyin** clause让threadprivate声明的变量的值和主线程的值相同，如下例子：

```

1 #include<omp.h>
2 #include<iostream>
3 int a;
4 #pragma omp threadprivate(a)
5 int main()
6 {
7     a = 99;
8     std::cout << omp_get_thread_num() << "：" << &a << std::endl << std::endl;
9     #pragma omp parallel num_threads(8) copyin(a)
10    {
11        #pragma omp critical
12        std::cout << omp_get_thread_num() << "：" << &a << " " << a << std::endl;
13    }
14    std::cin.get();
15    return 0;
16 }

```

```

0: 005941FC
2: *0059A4C4 99
3: *0059A97C 99
4: *0059ABE4 99
1: *0059A00C 99
5: *0059B0CC 99
6: *0059B584 99
7: *0059B894 99
0: *005941FC 99

```

如果第9行代码修改为去掉copyin clause，结果如下：

```
0: 005C41FC
0: *005C41FC 99
3: *005CA75C 0
5: *005CB134 0
2: *005CA2A4 0
1: *005CA00C 0
7: *005CB85C 0
4: *005CAC14 0
6: *005CB3D4 0
```

**copyprivate** clause让不同线程中的私有变量的值在所有线程中共享，例子：

```
1 int a = 0;
2 #pragma omp parallel num_threads(8) firstprivate(a)
3 {
4     #pragma omp single copyprivate(a)
5     a = omp_get_thread_num()+10;
6     #pragma omp critical
7     std::cout << omp_get_thread_num() << " : " << &a << " " << a << std::endl;
8 }
```

```
0: *003CF528 14
2: *0214F710 14
1: *01F5FB08 14
7: *02A5F91C 14
3: *0231F71C 14
5: *0278F96C 14
4: *025BFB00 14
6: *0295FD64 14
```

能写在copyprivate里的变量必须是线程私有的，变量a符合这个条件，从上面结果可以看出，single directive的代码是被第4号线程执行的，虽然第4号线程赋值的a只是这个线程私有的，但是该新值将被广播到其他线程的a，这就造成了上面的结果。

如果去掉copyprivate clause，结果变为：

```
3: *00F8FC94 0
7: *0295F8C0 0
0: *002DFBEC 10
2: *00E1FA2C 0
5: *0283FE64 0
6: *026AF80C 0
1: *00B2FDE8 0
4: *0258FB00 0
```

这次single directive的代码是被第0号线程执行的。

呼，终于说完了，未尽事宜，见另一篇文章：[OpenMP共享内存并行编程总结表](#)。

## 6. 加速比

加速比即同一程序串行执行时间除以并行执行时间，即并行化之后比串行的性能提高倍数。理论上，加速比受这些因素影响：程序可并行部分占比、线程数、负载是否均衡（可以查查Amdahl定律），另外，由于实际执行时并行程序可能存在的总线冲突，使得内存访问称为瓶颈（还有Cache命中率的问题），实际加速比一般低于理论加速比。

为了看看加速比随线程数增加的变化情况，编写了如下代码，需要在Release下编译运行代码：

```
1 #include<iostream>
2 #include<omp.h>
3 int main(int argc, char* arg[])
4 {
5     const int size = 1000, times = 10000;
```

```
6     long long int data[size], dataValue=0;
7     for(int j=1; j<=times; ++j)
8         dataValue += j;
9
10    #pragma omp parallel num_threads(16)
11        for(int i=0; i<1000000; ++i) ;
12
13    bool wrong; double t, tsigle;
14    for(int m=1; m<=16; ++m){
15        wrong = false;
16        t = omp_get_wtime();
17        for(int n=0; n<100; ++n){
18            #pragma omp parallel for num_threads(m)
19                for(int i=0; i<size; ++i){
20                    data[i] = 0;
21                    for(int j=1; j<=times; ++j)
22                        data[i] += j;
23                    if(data[i] != dataValue)
24                        wrong = true;
25                }
26        }
27        t = omp_get_wtime()-t;
28        if(m==1) tsigle=t;
29        std::cout << "num_threads(" << m << ") runtime: " << t << " s.\n";
30        std::cout << "wrong=" << wrong << "\tspeedup: " << tsigle/t << "\tefficiency: " << tsigle/t/m << "\n\n";
31    }
32
33    std::cin.get();
34    return 0;
35 }
```





```
num_threads(1) runtime: 2.13531 s.  
wrong=0 speedup: 1 efficiency: 1  
  
num_threads(2) runtime: 1.0134 s.  
wrong=0 speedup: 2.10707 efficiency: 1.05354  
  
num_threads(3) runtime: 0.678363 s.  
wrong=0 speedup: 3.14773 efficiency: 1.04924  
  
num_threads(4) runtime: 0.539407 s.  
wrong=0 speedup: 3.95862 efficiency: 0.989654  
  
num_threads(5) runtime: 0.455536 s.  
wrong=0 speedup: 4.68746 efficiency: 0.937492  
  
num_threads(6) runtime: 0.380201 s.  
wrong=0 speedup: 5.61625 efficiency: 0.936042  
  
num_threads(7) runtime: 0.326633 s.  
wrong=0 speedup: 6.53732 efficiency: 0.933903  
  
num_threads(8) runtime: 0.286388 s.  
wrong=0 speedup: 7.45599 efficiency: 0.931998  
  
num_threads(9) runtime: 0.47411 s.  
wrong=0 speedup: 4.50382 efficiency: 0.500424  
  
num_threads(10) runtime: 0.429944 s.  
wrong=0 speedup: 4.96648 efficiency: 0.496648  
  
num_threads(11) runtime: 0.434278 s.  
wrong=0 speedup: 4.91691 efficiency: 0.446992  
  
num_threads(12) runtime: 0.531302 s.  
wrong=0 speedup: 4.01901 efficiency: 0.334917  
  
num_threads(13) runtime: 0.512578 s.  
wrong=0 speedup: 4.16581 efficiency: 0.320447  
  
num_threads(14) runtime: 0.48439 s.  
wrong=0 speedup: 4.40823 efficiency: 0.314874  
  
num_threads(15) runtime: 0.495439 s.  
wrong=0 speedup: 4.30992 efficiency: 0.287328  
  
num_threads(16) runtime: 0.410763 s.  
wrong=0 speedup: 5.19839 efficiency: 0.324899  
  
=
```

可以看到，由于我们的程序是在操作系统层面上运行，而非直接在硬件上运行，上面的测试结果出现了看似不可思议的结果——效率竟然有时能大于1！最好的加速比出现在num\_threads(8)时，为7.4左右，已经很接近物理核心数8了，充分利用多核原来如此简单。

问题：1、for并行中还能嵌套for吗？2、for并行的颗粒度太小会怎么样？

标签: [OpenMP](#)