

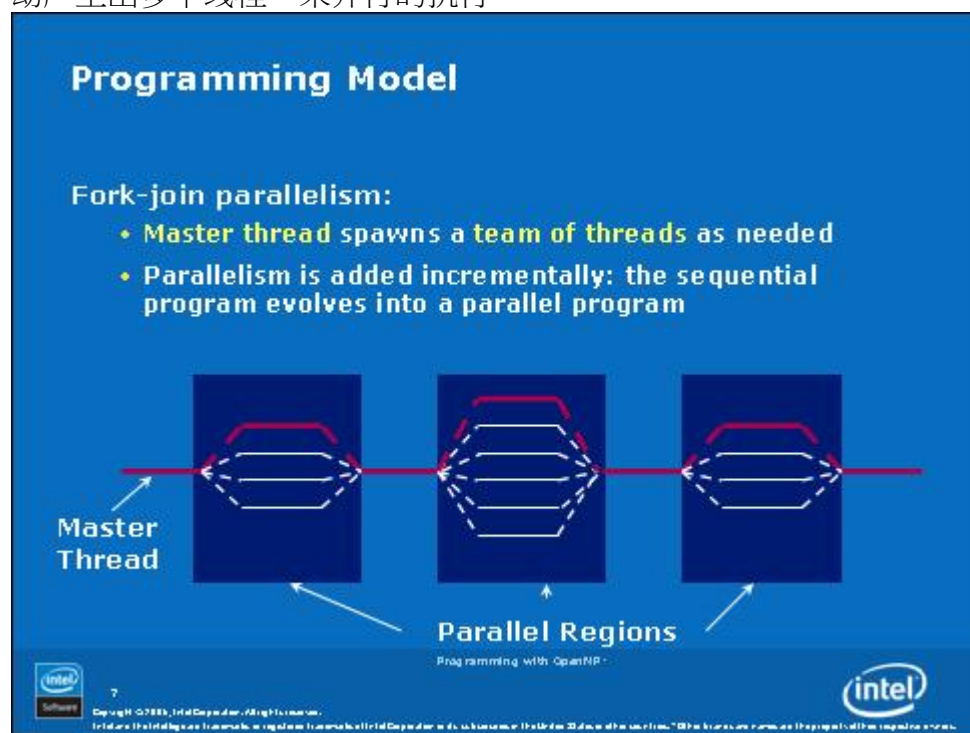
## OpenMP 入门

阅读(17) 评论(1) 发表时间：2009年05月06日 21:31

本文地址：<http://qzone.qq.com/blog/55032144-1241616677>

OpenMP是一个业界的标准，很早以前就有了，只是近一段时间才逐渐热起来。我们可以在C/C++和Fortran使用OpenMP、很容易的引入多线程。

下图是一个典型的OpenMP程序的示意图，我们可以看到它是由串行代码和并行代码交错组成的，并行代码的区域我们把它叫做“并行区”。主线程一旦进入并行区，就自动产生出多个线程，来并行的执行。



怎样在我们的代码中使用OpenMP呢？很简单，拿我们常用的C/C++代码来说，只需要插入如下**pragma**，然后我们选择不同的**construct**就可以完成不同的功能。

首先，我们来看一下如何创建一个并行区：增加一行代码**#pragma omp parallel**，然后用花括号把你需要放在并行区内的语句括起来，并行区就创建好了。并行区里每个线程都会去执行并行区中的代码。

## Parallel Regions

Defines **parallel region** over structured block of code

Threads are created as '**parallel**' pragma is crossed

Threads block at end of region

Data is shared among threads unless specified otherwise

**C/C++ :**

```
#pragma omp parallel
{
    block
}
```

Programming with OpenMP

Copyright © 2005, Intel Corporation. All rights reserved.  
Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Other brands are trademarks of their respective owners.

下面我们将看一个最简单的OpenMP代码，看看并行区是怎样工作的。  
我们来看一个最简单的OpenMP程序例子：

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

int main() {
    #pragma omp parallel
    {
        int i;
        printf("Hello World\n");
        for(i=0;i <6;i++)
            printf("Iter:%d\n",i);
    }
    printf("GoodBye World\n");
}
=====
```

我们现在用Intel编译器来编译这个小程序，当然你也可以用VS2005来编译。

我们可以看到编译器在没有加 /Qopenmp 开关的时候，会忽略掉所有openmp pragma。我们可以利用这个特性来检查你的代码在串行的时候运行

结果是否正确。

```
D:\test>icl /nologo HelloWorlds.c
HelloWorlds.c
```

```
HelloWorlds.c(4): warning #161: unrecognized #pragma
      #pragma omp parallel
      ^
```

串行代码执行的结果如下：

```
D:\test>HelloWorlds.exe
Hello World
Iter:0
Iter:1
Iter:2
Iter:3
Iter:4
Iter:5
GoodBye World
```

下面我们加上 /Qopenmp 开关重新编译，我们会看到编译器给出提示说已经把并行区给做并行化了。

```
D:\test>icl /nologo /Qopenmp HelloWorlds.c
HelloWorlds.c
HelloWorlds.c(4): (col. 5) remark: OpenMP DEFINED REGION WAS
PARALLELIZED.
```

我的机器是4核，运行这个程序就打印出如下结果。我们可以看到每次运行打印出来的东西顺序是不一定的，这个符合多线程程序的特性。在4

核的机器上，并行区内就产生了4个线程同时执行for循环打印，默认线程数=核的个数。

```
D:\test>HelloWorlds.exe
Hello World
Iter:0
Iter:1
Iter:2
Iter:3
Iter:4
Iter:5
Hello World
Iter:0
Hello World
Iter:0
Iter:1
Iter:2
Iter:3
Iter:4
Iter:5
Hello World
Iter:0
Iter:1
```

```
Iter:1
Iter:2
Iter:2
Iter:3
Iter:3
Iter:4
Iter:4
Iter:5
Iter:5
GoodBye World
```

我们来修改一下线程的数量，然后再运行一下代码，看下运行结果

```
D:\test>set OMP_NUM_THREADS=2
```

```
D:\test>HelloWorlds.exe
```

```
Hello World
```

```
Iter:0
Iter:1
Iter:2
Iter:3
Iter:4
Iter:5
Hello World
Iter:0
Iter:1
Iter:2
Iter:3
Iter:4
Iter:5
GoodBye World
```

现在我们对并行区有了一定的了解，并行区里每个线程执行的代码是一样的，做的事情似乎也是一样的，但是实际工作中，我们希望把一部分

工作分配给不同线程来做，这应该怎么实现呢？请看下面。

我们来看一个新的openmp语句

**#pragma omp for**

使用这个语句，我们就可以把一个**for**循环的工作量（例如：**1...N**）分配给不同线程。这个语句后面必须紧跟一个**for**循环，他只能对循环的工作量进行划分、分配。

## Work-sharing Construct

```
#pragma omp parallel
#pragma omp for
    for (i=0; i<N; i++){
        Do_Work(i);
    }
```

Splits loop iterations into threads

Must be in the parallel region

Must precede the loop



L3

Copyright © 2009 Intel Corporation. All rights reserved.

Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Other brands and product names are the property of their respective owners.

Programming with OpenMP



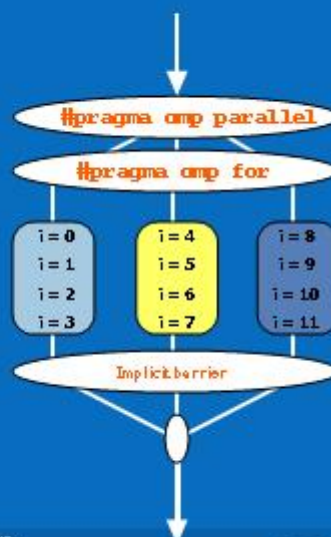
看下面这个例子，0...11的工作量被平均分配给了3个线程。当3个线程都完成了各自的工作后，程序才继续往下执行。

## Work-sharing Construct

```
#pragma omp parallel
#pragma omp for
    for(i = 0; i < 12; i++)
        c[i] = a[i] + b[i]
```

Threads are assigned an independent set of iterations

Threads must wait at the end of work-sharing construct



L3

Copyright © 2009 Intel Corporation. All rights reserved.

Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Other brands and product names are the property of their respective owners.

Programming with OpenMP



写2行openmp pragma实在有些麻烦，我们可以把2行或多行openmp pragma合并为一行，这样好多了，是吧。

## Combining pragmas

These two code segments are equivalent

```
#pragma omp parallel
{
    #pragma omp for
    for (i=0; i< MAX; i++) {
        res[i] = huge();
    }
}
```

```
#pragma omp parallel for
for (i=0; i< MAX; i++) {
    res[i] = huge();
}
```



L4

Programming with OpenMP



=====

数据环境（Data Environment）

=====

OpenMP属于共享内存的编程模型。在我们的多线程代码中，大部分数据都是可以共享的。共享内存给我们程序中数据的共享带来了极大的便利。因此在默认情况下，OpenMP将全局变量、静态变量设置为共享属性。

但是，还是有些变量需要是每个线程私有的，也就是每个线程有这些变量的独立拷贝，这样每个线程在使用这些变量时不会相互影响。需要私有的变量包括：

## Data Environment

But, not everything is shared...

- Stack variables in functions called from parallel regions are PRIVATE
- Automatic variables within a statement block are PRIVATE
- Loop index variables are private (with exceptions)
  - C/C+: The first loop index variable in nested loops following a `#pragma omp for`



L6

Copyright © 2015, Intel Corporation. All rights reserved.  
Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Other brands are names of their respective owners.

Programming with OpenMP



我们可以通过如下方法来改变OpenMP的变量默认属性，你可以把它设置为共享（shared）或无。也可以单独改变某几个变量的属性，把他们设置为shared或private。

## Data Scope Attributes

The default status can be modified with

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

Scoping attribute clauses

```
shared(varname, ...)
```

```
private(varname, ...)
```



L7

Copyright © 2015, Intel Corporation. All rights reserved.  
Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Other brands are names of their respective owners.

Programming with OpenMP



看看下面这个例子，循环变量*i*默认为私有，因为*x*和*y*是中间变量，应该设置为私有，否则线程之间的*x,y*会互相影响。



## The Private Clause

Reproduces the variable for each thread

- Variables are un-initialized; C++ object is default constructed
- Any value external to the parallel region is undefined

```
void* work(float* c, int N) {  
    float x, y; int i;  
    #pragma omp parallel for private(x,y)  
    for(i=0; i<N; i++) {  
        x = a[i]; y = b[i];  
        c[i] = x + y;  
    }  
}
```



L4

Copyright © 2009, Intel Corporation. All rights reserved.

Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Other brands and product names are the property of their respective owners.

Programming with OpenMP



再看看这个例子。变量sum定义在并行区之外，所以默认为共享，这个例子里又写了shared(sum)，没错，但是实际上是罗嗦了。那么这个例子里有什么错误呢？

## Example: Dot Product

```
float dot_prod(float* a, float* b, int N)  
{  
    float sum = 0.0;  
    #pragma omp parallel for shared(sum)  
    for(int i=0; i<N; i++) {  
        sum += a[i] * b[i];  
    }  
    return sum;  
}
```

What is Wrong?



L4

Copyright © 2009, Intel Corporation. All rights reserved.

Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Other brands and product names are the property of their respective owners.

Programming with OpenMP



如果以前做过多线程开发的话应该能看出来，sum不应该是共享的，但是设置为私有的也不对。我们的做法应该是将sum保护起来，防止多个线程同时对sum进行写操作。我们可以使用OpenMP的临界区来对sum进行保护。



## Protect Shared Data

Must protect access to shared, modifiable data

```
float dot_prod(float* a, float* b, int N)
{
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
        #pragma omp critical
        sum += a[i] * b[i];
    }
    return sum;
}
```



30

Copyright © 2009, Intel Corporation. All rights reserved. Intel and the Intel logo are trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Other brands are trademarks of their respective owners.

Programming with OpenMP



我们可以给临界区命名，在下面例子中，如果我们不给临界区命名，在任一时刻，只能有一个线程调用consum函数。而我们给临界区命名后，任一时刻可以有最多2个线程在调用consum函数（1个调用 consum(B, &R1)，另一个调用 consum(A, &R2)）。在这2句语句可以同时执行的情况下，我们通过临界区命名来尽可能减少线程等待时间。

## OpenMP\* Critical Construct

```
#pragma omp critical [(lock_name)]
```

Defines a critical region on a structured block

Threads wait their turn – at a time, only one calls consum() thereby protecting R1 and R2 from race conditions.

Naming the critical constructs is optional, but may increase performance.

```
float R1, R2;
#pragma omp parallel
{ float A, B;
  #pragma omp for
  for(int i=0; i<niters; i++){
    B = big_job(i);
    #pragma omp critical (R1_lock)
    consum (B, &R1);
    A = bigger_job(i);
    #pragma omp critical (R2_lock)
    consum (A, &R2);
  }
}
```



31

Copyright © 2009, Intel Corporation. All rights reserved. Intel and the Intel logo are trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Other brands are trademarks of their respective owners.

Programming with OpenMP



~~归约 (Reduction)~~ 是个很有用的功能，可以简化我们的编程，op代表一个操作，list是执行这个操作的一个或多个变量。

## OpenMP\* Reduction Clause

**reduction (op : list)**

The variables in "list" must be shared in the enclosing parallel region

Inside parallel or work-sharing construct:

- A PRIVATE copy of each list variable is created and initialized depending on the "op"
- These copies are updated locally by threads
- At end of construct, local copies are combined through "op" into a single value and combined with the value in the original SHARED variable



33

Programming with OpenMP



Copyright © 2009, Intel Corporation. All rights reserved.  
Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Other brands are names of the proprietors of the respective products.

我们再看刚才上面的例子就清楚了。我们对sum这个变量使用归约操作，操作符是+。这样的话，每个线程就会有一个私有的sum变量，当所有线程的计算完成后，每个线程的私有的sum的值将被用"+"归约成一个总的sum，即 线程1的sum + 线程2的sum + ... + 线程n的sum -> 总的sum，这个总的sum值将被带出并行区并赋给全局的那个sum变量，因此，当这个并行区的代码执行完以后，我们的sum变量的值就是我们期望得到的值了。

是不是比前面用临界区的方法要好得多、代码也会快得多呢？

## Reduction Example

```
#pragma omp parallel for reduction(+:sum)
for(i=0; i<N; i++) {
    sum += a[i] * b[i];
}
```

Local copy of *sum* for each thread

All local copies of *sum* added together and stored in "global" variable



33

Programming with OpenMP



Copyright © 2009, Intel Corporation. All rights reserved.  
Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Other brands are names of the proprietors of the respective products.

下图是归约支持的操作符：

## C/C++ Reduction Operations

A range of associative and commutative operators can be used with reduction

Initial values are the ones that make sense

Operator	Initial Value
+	0
*	1
-	0
^	0

Operator	Initial Value
&	~0
	0
&&	1
	0

34

Copyright © 2015, Intel Corporation. All rights reserved. Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Other brands and names are the property of their respective owners.

好了，来给大家做个小作业。下面代码是一个串行的求Pi的代码，使用的是积分的办法。请大家把这个代码用OpenMP来做并行化。

## Numerical Integration Example

$$\int_0^1 \frac{4.0}{(1+x^2)} dx$$

```

static long num_steps=100000;
double step, pi;

void main()
{
    int i;
    double x, sum = 0.0;

    step = 1.0/(double) num_steps;
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
    
```

35

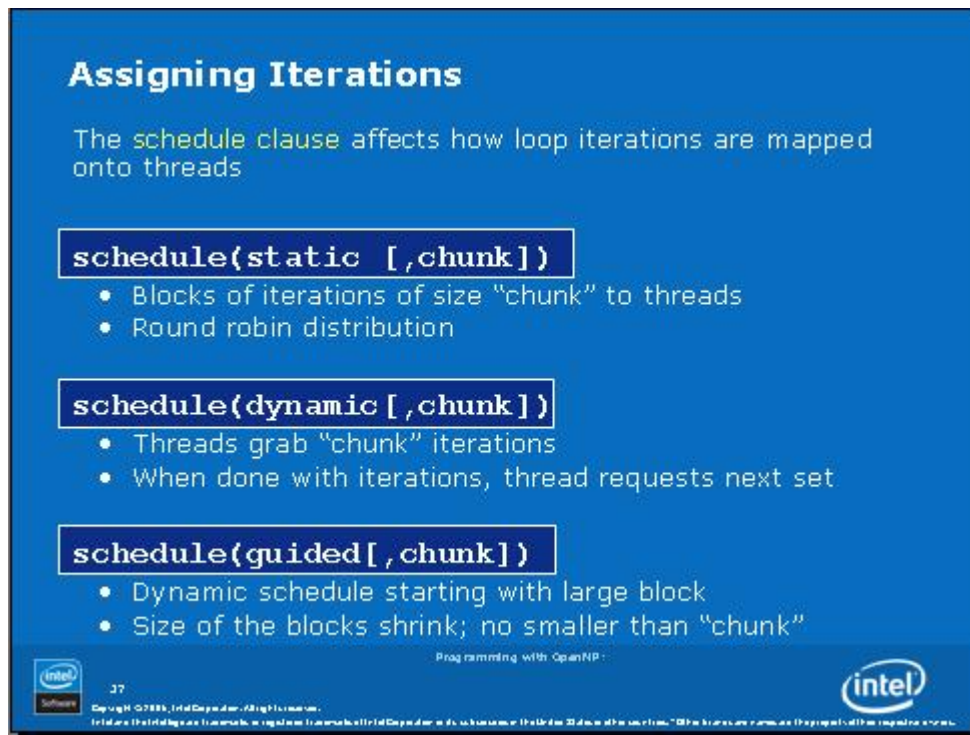
Copyright © 2015, Intel Corporation. All rights reserved. Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Other brands and names are the property of their respective owners.

=====

## OpenMP中工作量的划分与调度

=====

前面我看到在使用工作量共享(work-sharing)这种方式的时候，工作量是自动给我们划分好并分配给各个线程的。下面，我们来看看如何来控制工作量的划分与调度。



**Assigning Iterations**

The `schedule` clause affects how loop iterations are mapped onto threads

**`schedule(static [,chunk])`**

- Blocks of iterations of size "chunk" to threads
- Round robin distribution

**`schedule(dynamic [,chunk])`**

- Threads grab "chunk" iterations
- When done with iterations, thread requests next set

**`schedule(guided[,chunk])`**

- Dynamic schedule starting with large block
- Size of the blocks shrink; no smaller than "chunk"

Programming with OpenMP

Copyright © 2008, Intel Corporation. All rights reserved.  
Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Other brands and product names are trademarks of their respective owners.

如上图所示，工作量的划分与调度有3种方式：

- 1、静态：把循环的迭代按照每x次( $x = \text{chunk}$ )迭代分为一块，这样你的总工作量就被划分成了 $n/x$ 块( $n$ 为迭代次数、循环次数)，然后将这些块按照轮转法依次分配给各个线程。举个例子：比如我们有100次迭代， $x = \text{chunk} = 4$ ，那么我们的工作就被分为25块，假设我们有2个线程可以做工作，那么线程 1分到的块是1,3,5,7,...,25，线程2分到的块是2,4,6,...,24；
- 2、动态：迭代分块方法同上，但是工作块被放到一个队列中，每个线程每次拿一块，做好了才能到队列里去拿下一块；
- 3、Guided：这个方式是动态方式的改进。在这个方式里，分块的x是不固定的，一开始块的大小(x)比较大，随着剩余工作量的减小，块的大小也随之变小。

我们总结一下每种方式适合什么样的工作量

静态方式：比较适合每次迭代的工作量相近(主要指工作所需时间)的情况

动态方式：比较适合每次迭代的工作量非常不确定的情况

Guided方式：类似动态方式，但是队列相关的开销会比动态方式小

下面我们看一个例子：

```
#pragma omp parallel for schedule (static, 8)
for( int i = start; i <= end; i += 2 )
```



```

{
    if ( TestForPrime(i) ) gPrimesFound++;
}

```

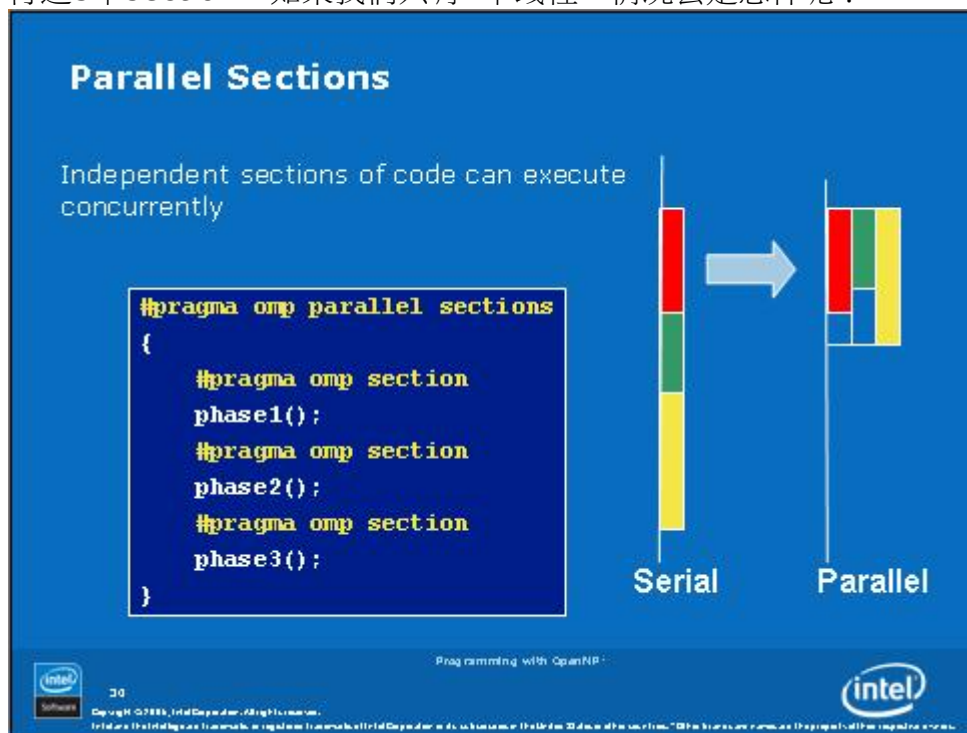
在这个例子中，如果start=3，那第一块就={3,5,7,9,11,13,15,17}

=====

另外几个OpenMP Construct

=====

下面我们来看一看Parallel Section，其实看看下面的图片就知道了，我们可以定义多个section，让这些section并行的执行。下面的例子是我们有足够的线程来同时执行这3个section，如果我们只有2个线程，情况会是怎样呢？



如果只有2个线程，那么肯定得有1个线程要比另一个线程勤劳一点，执行2个section了。

在使用Parallel Section时需要注意的是，每个section的工作之间应该是相互独立、没有依赖关系的。如果不满足这个要求的话，就不要对他们用并行了。

下面再介绍2个有可能会用到的OpenMP construct。

1、single：有时候在并行区里，我们希望有部分代码只能执行一次，也就是说只有一个线程去执行这部分代码。如下面的例子，ExchangeBoundaries() 这句语句前面我们加上 #pragma omp single，就保证只有一个线程去执行它。同时在single后面会有一个隐含的障碍（implicit barrier）。我们后面会具体介绍障碍这个概念。

```

#pragma omp parallel
{
    DoManyThings();
    #pragma omp single

```

```

{
    ExchangeBoundaries();
} // threads wait here for single
DoManyMoreThings();
}

```

2、master：master跟single很类似。在下面例子中，只有主线程会去执行ExchangeBoundaries() 这条语句。但是master没有隐含的障碍，因此如果其他线程遇到 #pragma omp master，就会跳过去，直接执行master后面的语句。

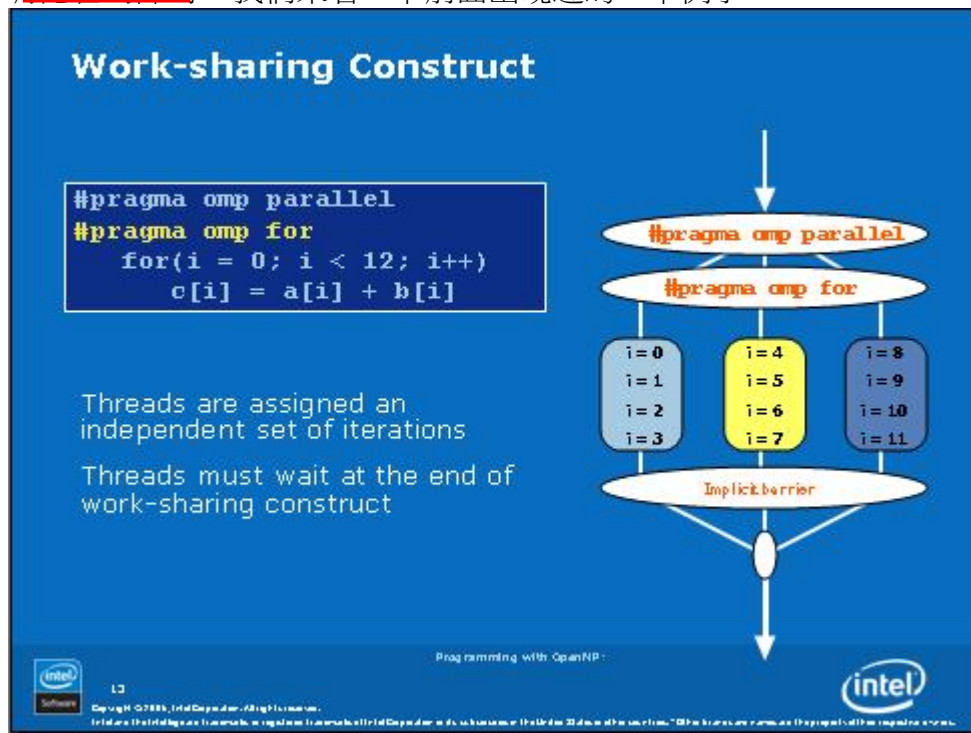
```

#pragma omp parallel
{
    DoManyThings();
    #pragma omp master
    { // if not master skip to next stmt
        ExchangeBoundaries();
    }
    DoManyMoreThings();
}

```

障碍

在我们前面介绍过的OpenMP Construct，如parallel, for, single，他们自身都带有隐含的障碍。我们来看一下前面出现过的一个例子：



sections

在这个例子里面，线程1做0-3的迭代，线程2做4-7的迭代，线程3做8-11的迭代，

如果每次迭代的工作量不同，那么线程1、2、3完成他们各自的工作 所需的时间是不同的，也就是说，某个线程可能比另外2个线程提前完成工作，但是这个线程不能继续往下走去执行并行区后面的工作，因为**#pragma omp for**里面带了隐含的障碍，这个障碍的意思就是说，所有的线程做完了自己的工作后必须在这里等，直到所有的线程都完成了各自的工作，大家才能往下走。汇编 里的**memory fence**与这个有点神似。

为什么这些**construct**要带了个隐含的障碍呢？障碍不是让程序执行速度变慢了吗？因为它怕你程序里面后面的代码对这块代码有依赖关系，如果这块代码的工作没完成就去执行后面的代码，可能会引起错误。

那如果你后面的代码对这块代码没有依赖，可以用 **nowait** 来把这个隐含的障碍给去掉。比如：

例子1：

C/C++ code **#pragma omp for nowait for(...) {...};**

例子2：

C/C++ code **#pragma single nowait{ [...] }**

例子3：

C/C++ code **#pragma omp for schedule(dynamic,1) nowait for(int i=0; i<n; i++) a = bigFunc1(i);#pragma omp for schedule(dynamic,1) for(int j=0; j<m; j++) b[j] = bigFunc2(j);**

=====  
介绍几个OpenMP API  
=====

在大多数情况下，我们不会用到**OpenMP API**。一般只有在调试和某些情况下，才需要用到**API**。

如果你需要使用 **OpenMP API**，记得先包含**OpenMP**头文件

C/C++ code **#include <omp.h>**

最常用的2个**API**是：

C/C++ code **int omp\_get\_thread\_num(void);int omp\_get\_num\_threads(void);**

在并行区里调用**omp\_get\_thread\_num**返回的是当前线程的线程**ID**，一般是0到(N-1)，N是并行区里的总线程数。

在并行区里调用**omp\_get\_num\_threads**返回的是并行区里的总线程数。

使用这2个**API**，我们可以只用 **#pragma omp parallel** 就实现每个线程完成循环中不同的迭代。比如我们有这样一段代码，怎样在不使用 **#pragma omp parallel for** 的情况下，改写代码、把工作量划分给各个线程呢？



C/C++ code `#pragma omp parallel { for(i=0;i<N;i++) { c = a + b; } }`

下面再介绍几个数据环境相关的construct。前面我们看过了private, public，现在来看另外3个。

1、firstprivate：变量属性为private，同时每个线程的这个变量的初始值为全局变量的值。比如下面例子中，我们先给全局变量incr设置了一个值0，然后再并行区，每个线程都有自己的incr，而这些incr的初始值也为0（与并行区之前的全局量incr的值一致）。

C/C++ code `incr=0;#pragma omp parallel for firstprivate(incr)for (I=0;I<=MAX;I++)  
{ if ((I%2)==0) incr++; A(I)=incr;}`

2、lastprivate：当退出并行区时，最后一次迭代内的lastprivate变量的值将被带出并行区赋给全局的同名变量。在下面例子中，i循环的最后一次迭代中的x的值将被赋给全局的x，所以退出并行区后，全局量x被赋值了。

C/C++ code `void sq2(int n, double *lastterm){ double x; int i; #pragma omp parallel  
#pragma omp for lastprivate(x) for (i = 0; i < n; i++){ x = a*a + b*b; b =  
sqrt(x); } lastterm = x;}`

3、threadprivate：用于指定某变量为线程私有的全局变量。threadprivate和private的区别是private只在并行区中有效，而threadprivate属性是全局范围内有效的。

copyin：把全局变量的值拷贝到各线程中同名的threadprivate变量中去。

C/C++ code `struct Astruct A;#pragma omp threadprivate(A)...#pragma omp parallel  
copyin(A) do_something_to(&A);...#pragma omp parallel do_something_else_to(&A);`

下面我们看个程序加强理解

C/C++ code `#include <stdio.h>int main(){ int i, x = 100; #pragma omp  
parallel for private(x) for (i=0; i<8; i++) { x += i;  
printf("x = %d\n", x); } printf("global x = %d\n", x); return 1;}`

因为我的cpu是4核的，所以默认有4个线程，线程1跑i=0、1，线程2跑i=2、3，线程3跑i=4、5，线程4跑i=6、7，所以运行结果为：

```
x = 0  
x = 1  
x = 2  
x = 5  
x = 6  
x = 13  
x = 4  
x = 9  
global x = 100
```

下面我们把private换成firstprivate，结果就变成了

```
x = 100  
x = 101  
x = 102  
x = 105  
x = 106  
x = 113  
x = 104  
x = 109
```

**global x = 100**

看出区别了吧，每个线程的x的初值都变成100了。

如果我们再加上个**lastprivate**：

```
C/C++ code #include <stdio.h>int main(){    int i, x = 100;    #pragma omp
parallel for firstprivate(x) lastprivate(x)    for (i=0; i<8; i++)    {        x +=
i;        printf("x = %d\n", x);    }    printf("global x = %d\n", x);    return
1;}
```

结果如下：

x = 100

x = 101

x = 102

x = 105

x = 106

x = 113

x = 104

x = 109

**global x = 113**

全局的x的值最后变成了i=7的x的值，i=7是由线程4来做的。线程4在做完i=6时，私有x=106，做完i=7时，私有x=113，因为是**lastprivate**，113被带给全局量x了。

公布一下家庭作业留得问题的答案，答案有很多种，我提供2个做参考，这2种的区别是不同的工作量划分方法：

```
C/C++ code #pragma omp parallel{    int i, istart, iend;    int Nthrds =
omp_get_num_thread(), id = omp_get_thread_num();    istart = id * N / Nthrds;    iend
= (id+1) * N / Nthrds;    for(i=istart; i<iend; i++)    {        c[i] = a[i] + b[i];    }}
```

```
C/C++ code #pragma omp parallel{    int i, istart, iend;    int Nthrds =
omp_get_num_thread(), id = omp_get_thread_num();    for(i=id; i<iend; i+=Nthrds)
{        c[i] = a[i] + b[i];    }}
```