

## OpenMp 课堂编程练习实验

实验内容

实验步骤

本地电脑VS上运行

TaiShan服务器上运行

实验结果与分析

parallel Construct

Worksharing construct

Combined Parallel Worksharing Constructs

Combined Parallel Section Constructs

Synchronization Constructs

Critical

atomic

Variable and reduction

PI- parallel for Construct

PI- parallel Construct

矩阵计算

实验总结

功能指令

子句

API函数

环境变量

姓名：张煜玮      学号：19291255      班级：计科1905

# OpenMp 课堂编程练习实验

## 实验内容

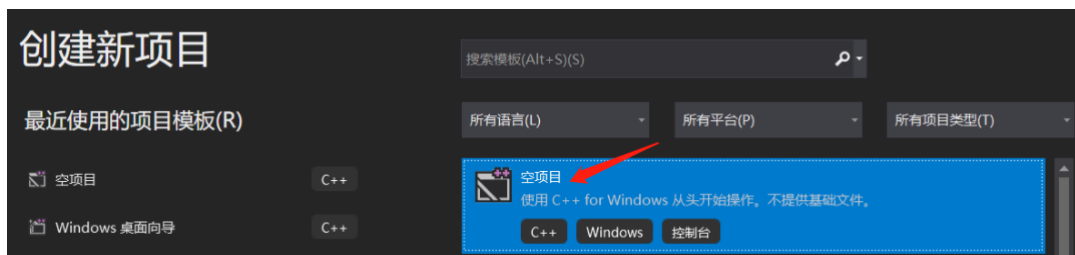
课上案例代码、运行结果截屏、结果分析，以上内容分别在个人电脑和TaiShan服务器上完成。

## 实验步骤

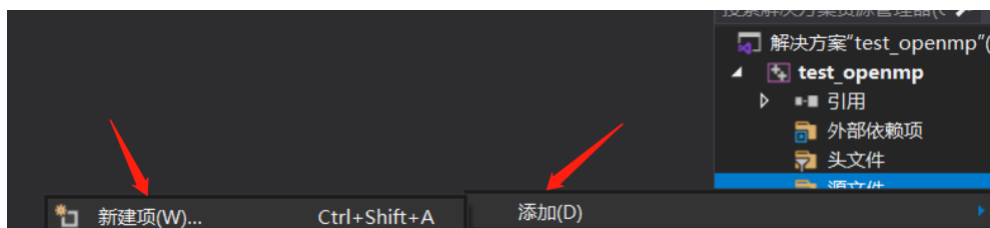
### 本地电脑VS上运行

以HelloWorld为例，演示如何在VS使用openMp进行并行编程。

1. 打开vs2019, 并创建一个空项目



2. 在【源文件】中添加【新建项】



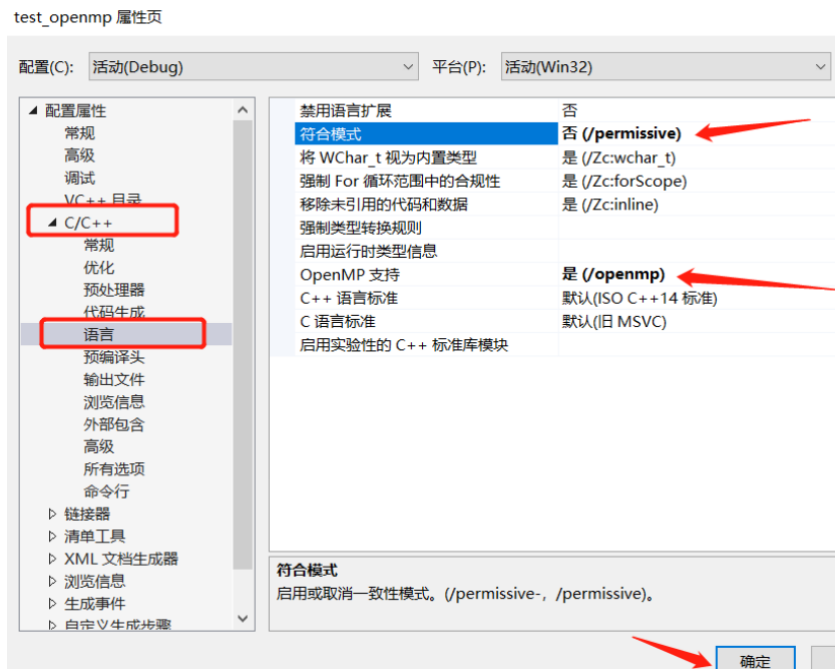
3. 创建.cpp文件
4. 添加如下测试代码

```
1 // parallel Construct
2 # include "omp.h"
3 # include "stdio.h"
4 int main()
5 {
6     int id, numb;
7     omp_set_num_threads(5);
8 #pragma omp parallel private(id, numb)
9     {
10         id = omp_get_thread_num();
11         numb = omp_get_num_threads();
12         printf("I am thread %d out of %d\n",id,numb );
13     }
14 }
```

5. 在【项目】中找到【属性】



5. 【符合模式】选择【否】，【OpenMP】选择【是】



6. 运行程序，出现如下结果（可以运行多次，每次的结果都不太一样）

```
Microsoft Visual Studio 调试控制台
I am thread 0 out of 5
I am thread 1 out of 5
I am thread 2 out of 5
I am thread 4 out of 5
I am thread 3 out of 5

C:\Users\zyw\source\repos\test_openmp\Debug\test_openmp.exe (进程 22224) 已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口。...
```

```
Microsoft Visual Studio 调试控制台
I am thread 2 out of 5
I am thread 1 out of 5
I am thread 0 out of 5
I am thread 3 out of 5
I am thread 4 out of 5

C:\Users\zyw\source\repos\test_openmp\Debug\test_openmp.exe (进程 12632) 已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口。...
```

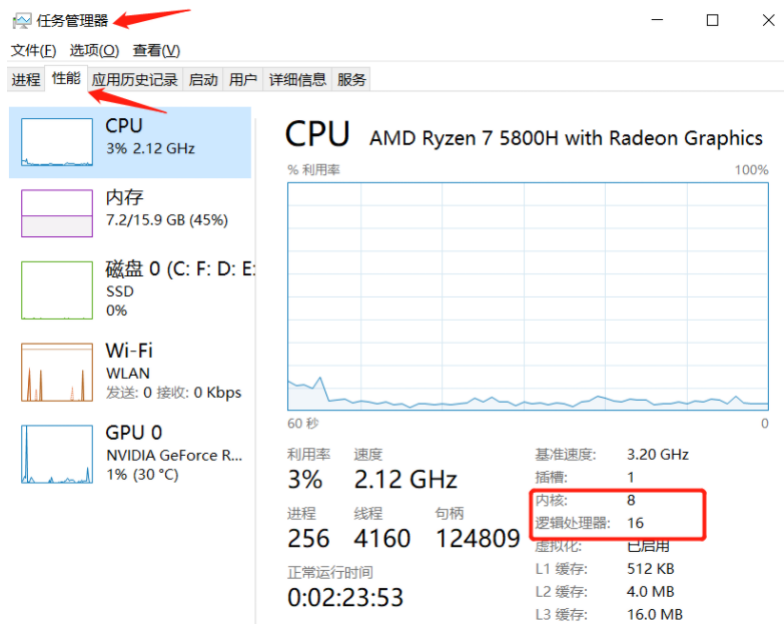
另外，将代码中的 `omp_set_num_threads(5)` 注释（即不指定创建的线程数），再次运行，得到的结果如下：

```
I am thread 0 out of 16
I am thread 6 out of 16
I am thread 4 out of 16
I am thread 5 out of 16
I am thread 3 out of 16
I am thread 2 out of 16
I am thread 1 out of 16
I am thread 10 out of 16
I am thread 7 out of 16
I am thread 8 out of 16
I am thread 9 out of 16
I am thread 11 out of 16
I am thread 12 out of 16
I am thread 14 out of 16
I am thread 15 out of 16
I am thread 13 out of 16
```

## 7. 结果分析

通过子句 `num_thread` 可以显式控制创建的线程数，如上图所示，创建的线程数为5，通过 `private` 指定id和numb在每个线程中都有它自己的私有副本。同时可以发现每次运行时线程ID出现的次序不同，说明多线程并发运行时，其速度并不是完全一样的而是随机的。

当不指定线程数时，自动创建的线程个数为16。这个自动创建的线程数与个人笔记本的电脑配置有关系。在任务管理器中查看，发现CPU的内核有8个，逻辑处理器有16个，因此自动创建的线程数与逻辑处理器的个数相等。



## TaiShan服务器上运行

我使用的辅助软件是Xftp和Xshell, 其中Xftp用来向远程的服务器发送文件，Xshell用来输入命令行控制远程的服务器。

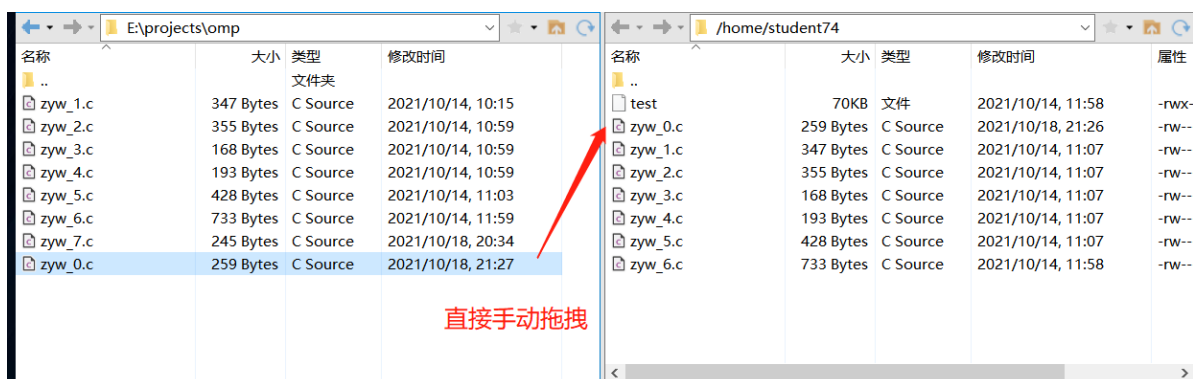
1. 打开Xftp，并新建一个对话



2. 填写会话的相关属性



3. 将本地的"zyw\_0"文件的测试文件拖动到服务器上



4. 打开Xshell, 并创建一个新的对话, 属性设置方法同Xftp.



5. 输入编译命令 `gcc zyw_0.c -o test -fopenmp`

```
[student74@localhost ~]$ gcc zyw_0.c -o test -fopenmp
```

6. 输入执行命令 `./test` 运行结果如下图, 和在VS上运行的结果类似。

```
[student74@localhost ~]$ ./test
I am thread 0 out of 5
I am thread 4 out of 5
I am thread 1 out of 5
I am thread 2 out of 5
I am thread 3 out of 5
```

7. 输入命令 `nano zyw_0.c`, 编辑源程序。将代码中的 `omp_set_num_threads(5)` 注释 (即不指定创建的线程数), 再次运行, 得到的结果部分截图如下。可以看到泰山服务器自动创建的线程数为 64, 因此可以推测其逻辑处理器有 64 个, 是个人笔记本电脑的 4 倍。

```
[student74@localhost ~]$ nano zyw_0.c
[student74@localhost ~]$ gcc zyw_0.c -o test -fopenmp
[student74@localhost ~]$ ./test
I am thread 0 out of 64
I am thread 62 out of 64
I am thread 1 out of 64
I am thread 14 out of 64
I am thread 42 out of 64
I am thread 53 out of 64
I am thread 59 out of 64
I am thread 56 out of 64
```

# 实验结果与分析

## parallel Construct

实验目的：

测试指令 `parallel`

实验原理：

`parallel`：用在一个结构块之前，表示这段代码将被多个线程并行执行

实验代码：

```
1 // parallel Construct
2 # include "omp.h"
3 # include "stdio.h"
4 int main()
5 {
6     int id, numb;
7     omp_set_num_threads(5);
8 #pragma omp parallel private(id, numb)
9     {
10         id = omp_get_thread_num();
11         numb = omp_get_num_threads();
12         printf("I am thread %d out of %d\n",id,numb );
13     }
14 }
```

实验结果和分析：

该代码(zyw\_0.c)实验结果和分析已经在[实验步骤](#)模块有详细的介绍，见上。

## Worksharing construct

实验目的：

测试指令 `for`

实验原理：

`for`：用于for循环语句之前，表示将循环计算任务分配到多个线程中并行执行，以实现任务分担，必须由编程人员自己保证每次循环之间无数据相关性。

其基本格式如下：

```
1 #pragma omp for [clause[[],] clause] ...]
2 for-loops
```

`parallel for` 与 `parallel` 的区别：使用`parallel`制导指令只是产生了并行域，让多个线程分别执行相同的任务，并没有实际的使用价值。`parallel for`用于生成一个并行域，并将计算任务在多个线程之间分配，从而加快计算运行的速度。可以让系统默认分配线程个数，也可以使用`num_threads`子句指定线程个数。

实验代码：

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

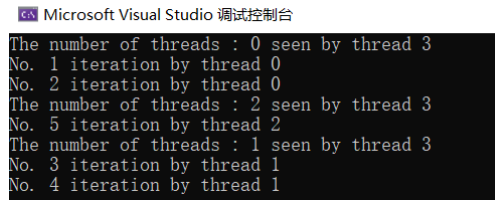
```

3
4 void main()
5 {
6     omp_set_num_threads(3);
7     #pragma omp parallel
8     {
9         printf("The number of threads : %d seen by thread %d\n",
omp_get_thread_num(), omp_get_num_threads());
10    #pragma omp for
11        for (int i = 1; i <= 5; ++i) {
12            printf("No. %d iteration by thread %d\n", i,
omp_get_thread_num());
13        }
14    }
15 }

```

### 实验结果:

- 个人电脑结果:

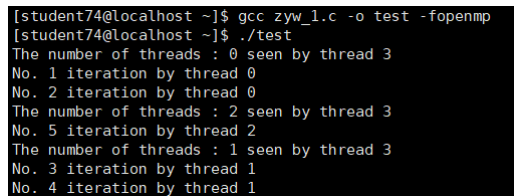


```

Microsoft Visual Studio 调试控制台
The number of threads : 0 seen by thread 3
No. 1 iteration by thread 0
No. 2 iteration by thread 0
The number of threads : 2 seen by thread 3
No. 5 iteration by thread 2
The number of threads : 1 seen by thread 3
No. 3 iteration by thread 1
No. 4 iteration by thread 1

```

- 华为服务器结果:



```

[student74@localhost ~]$ gcc zyw_1.c -o test -fopenmp
[student74@localhost ~]$ ./test
The number of threads : 0 seen by thread 3
No. 1 iteration by thread 0
No. 2 iteration by thread 0
The number of threads : 2 seen by thread 3
No. 5 iteration by thread 2
The number of threads : 1 seen by thread 3
No. 3 iteration by thread 1
No. 4 iteration by thread 1

```

### 结果分析:

由上图可以看出，在个人电脑和华为服务器上运行的结果相同。

运行时一共创建了三个线程，迭代了五次。其中线程0、线程1分别被使用两次，线程2使用一次。可以看出，循环过程中的n次任务在不同线程之间是相对均匀分配的。

### 对比实验:

1. 去掉程序代码中的 #pragma omp for 语句，编译运行结果如下。可以看出一共有三个线程，每个线程都迭代了5次。其结果说明，仅使用parallel制导指令只是产生了并行域，让多个线程分别执行相同的任务，并不能起到不同的线程共同分担任务的效果，是没有意义的。

```
Microsoft Visual Studio 调试控制台
The number of threads : 0 seen by thread 3
No. 1 iteration by thread 0
No. 2 iteration by thread 0
No. 3 iteration by thread 0
No. 4 iteration by thread 0
No. 5 iteration by thread 0
The number of threads : 1 seen by thread 3
No. 1 iteration by thread 1
No. 2 iteration by thread 1
No. 3 iteration by thread 1
No. 4 iteration by thread 1
No. 5 iteration by thread 1
The number of threads : 2 seen by thread 3
No. 1 iteration by thread 2
No. 2 iteration by thread 2
No. 3 iteration by thread 2
No. 4 iteration by thread 2
No. 5 iteration by thread 2
```

2. 如果去掉 `#pragma omp parallel`，编译运行结果如下。只有一个线程被调用，这个线程迭代了5次。说明如果没有并行域，那么只有一个线程会被调用，该线程完成全部的5次迭代任务。

```
Microsoft Visual Studio 调试控制台
The number of threads : 0 seen by thread 1
No. 1 iteration by thread 0
No. 2 iteration by thread 0
No. 3 iteration by thread 0
No. 4 iteration by thread 0
No. 5 iteration by thread 0
```

## Combined Parallel Worksharing Constructs

**实验目的：**

测试指令 `parallel for`

**实验原理：**

`parallel for`：parallel和for指令的结合，也是用在for循环语句之前，表示for循环体的代码将被多个线程并行执行，它同时具有并行域的产生和任务分担两个功能。

其基本格式如下：

```
1 | #pragma omp parallel for [clause[[,] clause] ...]
2 | for-loop
```

**实验代码：**

```
1 | #include <stdio.h>
2 | #include <omp.h>
3 | int main() {
4 |     #pragma omp parallel for
5 |     for(int i = 0; i < 10; ++i) {
6 |         printf(" No. %d iteration by thread %d\n",i,omp_get_thread_num() )5
7 |     }
8 |     return 0;
9 | }
```

**实验结果：**

- 个人电脑的结果



```
Microsoft Visual Studio 调试控制台
No. 0 iteration by thread 0
No. 1 iteration by thread 1
No. 3 iteration by thread 3
No. 2 iteration by thread 2
No. 6 iteration by thread 6
No. 8 iteration by thread 8
No. 9 iteration by thread 9
No. 4 iteration by thread 4
No. 7 iteration by thread 7
No. 5 iteration by thread 5
```

- 华为服务器结果

```
[student74@localhost ~]$ gcc zyw_7.c -o test -fopenmp
[student74@localhost ~]$ ./test
No. 0 iteration by thread 0
No. 1 iteration by thread 1
No. 2 iteration by thread 2
No. 3 iteration by thread 3
No. 4 iteration by thread 4
No. 7 iteration by thread 7
No. 5 iteration by thread 5
No. 6 iteration by thread 6
No. 9 iteration by thread 9
No. 8 iteration by thread 8
```

### 结果分析：

由上图可以看出，在个人电脑和华为服务器上运行的结果相同。

该程序一共设置了10个循环，每个循环任务由一个线程承担。说明了 `parallel for` 指令同时具有并行域的产生和任务分担两个功能。

### 对比实验：

如果在 `#pragma omp parallel for` 之前添加 `omp_set_num_threads(5);`

实验结果如下：

```
Microsoft Visual Studio 调试控制台
No. 0 iteration by thread 0
No. 1 iteration by thread 0
No. 4 iteration by thread 2
No. 5 iteration by thread 2
No. 2 iteration by thread 1
No. 3 iteration by thread 1
No. 8 iteration by thread 4
No. 9 iteration by thread 4
No. 6 iteration by thread 3
No. 7 iteration by thread 3
```

从上图可以看出，当指定的线程数（5个）大于总共的循环次数（10次）时，每个线程分担相等的循环任务（2次）

## Combined Parallel Section Constructs

### 实验目的：

测试指令 `section`

### 实验原理：

`sections`：用在可被并行执行的代码段之前，用于实现多个结构块语句的任务分担，可并行执行的代码段各自用 `section` 指令标出（注意区分 `sections` 和 `section`）

`parallel sections`：`parallel` 和 `sections` 两个语句的结合，类似于 `parallel for`。

其格式如下：

```

1 | #pragma omp parallel sections
2 | [clause[[],] clause] ...] new-line {
3 | [#pragma omp section new-line]
4 |   structured-block
5 | [#pragma omp section new-line ]
6 |   structured-block
7 | }

```

### 实验代码：

```

1 | #include <stdio.h>
2 | #include <omp.h>
3 | int main() {
4 |     #pragma omp parallel sections
5 |     {
6 |         #pragma omp section
7 |         for (int i = 0; i < 5; ++i) {
8 |             printf("section i : iteration % d by thread no. % d\n", i,
omp_get_thread_num());
9 |         }
10 |        #pragma omp section
11 |        for (int j = 0; j < 5; ++j) {
12 |            printf("section j : iteration % d by thread no. % d\n", j,
omp_get_thread_num());
13 |        }
14 |    }
15 | }

```

### 实验结果

- 本地电脑结果

Microsoft Visual Studio 调试控制台

```

section i : iteration 0 by thread no. 0
section i : iteration 1 by thread no. 0
section i : iteration 2 by thread no. 0
section i : iteration 3 by thread no. 0
section i : iteration 4 by thread no. 0
section j : iteration 0 by thread no. 5
section j : iteration 1 by thread no. 5
section j : iteration 2 by thread no. 5
section j : iteration 3 by thread no. 5
section j : iteration 4 by thread no. 5

section i : iteration 0 by thread no. 0
section i : iteration 1 by thread no. 0
section i : iteration 2 by thread no. 0
section i : iteration 3 by thread no. 0
section i : iteration 4 by thread no. 0
section j : iteration 0 by thread no. 6
section j : iteration 1 by thread no. 6
section j : iteration 2 by thread no. 6
section j : iteration 3 by thread no. 6
section j : iteration 4 by thread no. 6

```

- 远程服务器结果

```

[student74@localhost ~]$ gcc zyw_8.c -o test -fopenmp
[student74@localhost ~]$ ./test
section i : iteration 0 by thread no. 0
section i : iteration 1 by thread no. 0
section i : iteration 2 by thread no. 0
section i : iteration 3 by thread no. 0
section i : iteration 4 by thread no. 0
section j : iteration 0 by thread no. 50
section j : iteration 1 by thread no. 50
section j : iteration 2 by thread no. 50
section j : iteration 3 by thread no. 50
section j : iteration 4 by thread no. 50

```

```
[student74@localhost ~]$ ./test
section i : iteration 0 by thread no. 0
section i : iteration 1 by thread no. 0
section i : iteration 2 by thread no. 0
section i : iteration 3 by thread no. 0
section j : iteration 0 by thread no. 56
section j : iteration 1 by thread no. 56
section j : iteration 2 by thread no. 56
section i : iteration 4 by thread no. 0
section j : iteration 3 by thread no. 56
section j : iteration 4 by thread no. 56
```

## 结果分析

由上图的实验结果可以看出，每个section 模块分配有一个线程，在单个section的模块中，按照次序迭代5次，但是不同的section 模块i和j中的迭代语句可以交叉，即并行运行。说明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。

## Synchronization Constructs

### 实验目的：

测试指令 `barrier`

### 实验原理：

`barrier`：用于并行域内代码的线程同步，线程执行到barrier时要停下等待，直到所有线程都执行到barrier时才继续往下执行

格式如下：

```
1 | #pragma omp barrier
```

### 实验代码：

```
1 | #include <omp.h>
2 | #include <stdio.h>
3 |
4 | int main(){
5 |     #pragma omp parallel
6 |     {
7 |         for (int i = 0; i < 10; i++) {
8 |             printf("loop i: iteration %d by thread no.%d\n", i,
omp_get_thread_num());
9 |         }
10 | #pragma omp barrier
11 |         for (int j = 0; j < 10; j++) {
12 |             printf("loop j: iteration %d by thread no.%d\n", j,
omp_get_thread_num());
13 |         }
14 |     }
15 |     return 0;
16 | }
```

### 实验结果：

- 本地测试结果

```

loop i: iteration 4 by thread no.12
loop i: iteration 5 by thread no.12
loop i: iteration 0 by thread no.1
loop i: iteration 1 by thread no.1
loop i: iteration 2 by thread no.1
loop i: iteration 3 by thread no.1
loop i: iteration 4 by thread no.1
loop i: iteration 5 by thread no.1
loop i: iteration 6 by thread no.1
loop i: iteration 7 by thread no.1
loop i: iteration 8 by thread no.1
loop i: iteration 9 by thread no.1
loop i: iteration 6 by thread no.12
loop i: iteration 7 by thread no.12
loop i: iteration 8 by thread no.12
loop i: iteration 9 by thread no.12
loop j: iteration 0 by thread no.1
loop j: iteration 1 by thread no.1
loop j: iteration 2 by thread no.1
loop j: iteration 3 by thread no.1
loop j: iteration 4 by thread no.1
loop j: iteration 5 by thread no.1
loop j: iteration 6 by thread no.1
loop j: iteration 7 by thread no.1
loop j: iteration 8 by thread no.1
loop j: iteration 9 by thread no.1
loop j: iteration 0 by thread no.0
loop j: iteration 1 by thread no.0
loop j: iteration 2 by thread no.0
loop j: iteration 3 by thread no.0

```

- 华为服务器测试结果

```

loop i: iteration 9 by thread no.46
loop i: iteration 9 by thread no.61
loop i: iteration 9 by thread no.26
loop i: iteration 0 by thread no.55
loop i: iteration 1 by thread no.55
loop i: iteration 2 by thread no.55
loop i: iteration 3 by thread no.55
loop i: iteration 4 by thread no.55
loop i: iteration 5 by thread no.55
loop i: iteration 6 by thread no.55
loop i: iteration 7 by thread no.55
loop i: iteration 8 by thread no.55
loop i: iteration 9 by thread no.55
loop i: iteration 9 by thread no.49
loop i: iteration 8 by thread no.44
loop i: iteration 9 by thread no.44
loop j: iteration 0 by thread no.46
loop j: iteration 0 by thread no.55
loop j: iteration 0 by thread no.60
loop j: iteration 0 by thread no.17
loop j: iteration 1 by thread no.17
loop j: iteration 0 by thread no.54
loop j: iteration 1 by thread no.54
loop j: iteration 2 by thread no.54
loop j: iteration 3 by thread no.54
loop j: iteration 0 by thread no.0
loop j: iteration 0 by thread no.3
loop j: iteration 0 by thread no.52
loop j: iteration 0 by thread no.27
loop j: iteration 0 by thread no.4
loop j: iteration 0 by thread no.36
loop j: iteration 1 by thread no.46
loop j: iteration 0 by thread no.6
loop j: iteration 0 by thread no.7
loop j: iteration 0 by thread no.31
loop j: iteration 4 by thread no.54
loop j: iteration 5 by thread no.54
loop j: iteration 6 by thread no.54

```

### 结果分析:

由上图结果可以看出，在一个循环内，每个线程都被迭代10次，这个过程是并行的。但循环之间是按照次序进行的，即loop i 全部执行完后才执行loop j的内容。说明了线程执行到barrier时要停下等待，直到所有线程都执行到barrier时才继续往下执行。

除此之外，还可以发现在本地服务器上运行时，最大线程数量为16；华为服务器上运行时，最大线程数量为64。同时也验证了华为服务器的逻辑处理器的数量为64而我的电脑的处理器数量为16。

# Critical

## 实验目的：

测试 `critical`

## 实验原理：

`critical`：用在一段代码临界区之前，保证每次只有一个OpenMP线程进入。临界区指的是一个访问共用资源（例如：共用设备或是共用存储器）的程序片段，而这些共用资源又无法同时被多个线程访问的特性。当有线程进入临界区段时，其他线程或是进程必须等待

`shared`：指定一个或多个变量为多个线程间的共享变量；

## 实验代码：

```
1  #include <omp.h>
2  #include <stdio.h>
3
4  int main() {
5      int x; x = 0;
6      #pragma omp parallel shared(x)
7      {
8          #pragma omp critical
9              x += 1;
10     }
11     printf("x=%d\n", x);
12 }
```

## 实验结果：

- 本地测试结果

Microsoft Visual Studio 调试控制台

```
x=16
C:\Users\zyw\source\repos\Project1\x64\Debug\Project1.exe (进程 11236)已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口。...
```

- 华为服务器测试结果

```
[student74@localhost ~]$ gcc zyw_3.c -o test -fopenmp
[student74@localhost ~]$ ./test
x=64
```

## 结果分析：

其中x被设置为多个线程之间的共享变量，由多个线程分别执行一次迭代，在每次的迭代过程中x的值加1，因此最后的x的值等于线程的个数。因此在本地电脑上的结果x=16, 在华为服务器上的结果是64。

该结果说明了临界区在同一时间只能有一个线程执行它,其它线程要执行临界区则需要排队来执行它,这时等同于多个线程串行执行。

# atomic

## 实验目的：

测试指令 `atomic`

## 实验原理：

`atomic`：用于指定一个数据操作需要原子性地完成，原子操作是指不会被线程调度机制打断的操作；这种操作一旦开始，就一直运行到结束。

`shared`：指定一个或多个变量为多个线程间的共享变量

实验代码：

```
1  #include <omp.h>
2  #include <stdio.h>
3
4  int main() {
5      int x; x = 0;
6      //omp_set_num_threads(4);
7      #pragma omp parallel shared(x)
8      {
9          #pragma omp atomic
10         x += 1;
11     }
12     printf("x=%d\n", x);
13 }
```

实验结果：

- 本地测试结果：

- 华为服务器测试结果

```
[student74@localhost ~]$ gcc zyw_4.c -o test -fopenmp
[student74@localhost ~]$ ./test
x=64
```

结果分析：

该结果与上一题中使用critical的结果一样，相当于不同线程间串行运行。其中的critical 和atomic 的区别在于：critical 可以对代码块进行临界区设置，而atomic只能对代码语句进行加持。原子操作是要独占处理器，其他的线程必须等原子操作完了才可以运行。

## Variable and reduction

### PI- parallel for Construct

实验目的：

测试子句 `reduction`，并计算pi的值

实验原理：

模拟计算圆周率的数学公式如下：

$$\frac{\pi}{4} = \int_0^1 \frac{1}{1+x^2} dx \quad (1)$$

把0-1下面积分为n个小矩形，再在每个处理器上处理一部分面积，最后加起来。

`#pragma omp parallel for` 用在一个for循环的前面，表示下面的一行代码或代码块要分配到多个执行单元中并行计算。

`private(local)` 默认情况下定义在并行代码之外的变量为各并行的执行单元所共享，使用private限制，表示每个执行单元创建该变量的一个副本

**reduction**：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归约运算，并将结果返回给主线程同名变量。

格式如下：

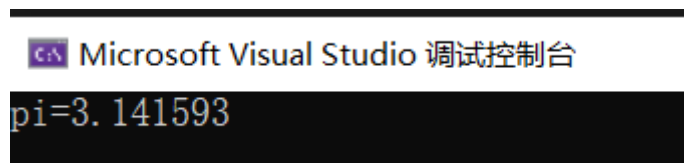
```
1 #pragma omp parallel for [clause[[,] clause] ...]
2 for-loop
3 private(list)
4 shared(list)
5 reduction(operator:list)
```

实验代码：

```
1 #include <omp.h>
2 #include <stdio.h>
3
4 static long num_steps = 100000;
5 double step;
6 #define NUM_THREADS 2
7 void main() {
8     int i;
9     double x, pi, sum = 0.0;
10    step = 1.0 / (double)num_steps;
11    omp_set_num_threads(NUM_THREADS);
12    #pragma omp parallel for reduction(+:sum) private(x)
13    for (i = 1; i <= num_steps; i++) {
14        x = (i - 0.5) * step;
15        sum += 4.0 / (1.0 + x * x);
16    }
17    pi = step * sum;
18    printf("pi=%f\n", pi);
19 }
```

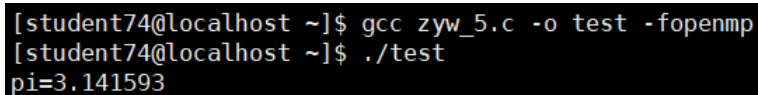
实验结果：

- 本地电脑结果



```
C:\ Microsoft Visual Studio 调试控制台
pi=3.141593
```

- 华为服务器结果



```
[student74@localhost ~]$ gcc zyw_5.c -o test -fopenmp
[student74@localhost ~]$ ./test
pi=3.141593
```

结果分析：

由上图可知，两者都得到了正确的圆周率的估计值“3.141593”

为该程序分配多个线程，其中指定x为private,即每个线程创建一个x的副本。reduction(+:sum)表示并行代码执行完毕后对各个执行单元中的sum进行相加操作，说明每个线程计算一个小长方形的面积，根据积分的原理，最后所有的小长方形的面积之和近似等于圆周率的值。

## PI- parallel Construct

### 实验目的：

对比实验，上一题中使用 `parallel for` 命令，本题中使用 `parallel` 来计算圆周率的值。

### 实验原理：

模拟计算圆周率的数学公式如下：

$$\frac{\pi}{4} = \int_0^1 \frac{1}{1+x^2} dx \quad (2)$$

把0-1下面积分为n个小矩形，再在每个处理器上处理一部分面积，最后加起来。

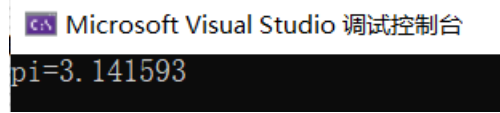
**reduction**：在reduction子句中，编译器为每个线程创建变量sum的私有副本。当循环完成后，将这些值加在一起并把结果放到原始的变量sum中；Reduction中的op操作必须满足算术结合律和交换律

### 实验代码

```
1  #include <omp.h>
2  #include <stdio.h>
3
4  static long num_steps = 100000;
5  double step;
6  #define NUM_THREADS 2
7  void main() {
8      int i, id;
9      double x, pi, sum = 0.0;
10     step = 1.0 / (double)num_steps;
11     omp_set_num_threads(NUM_THREADS);
12     #pragma omp parallel private(x,i,id) reduction(+:sum)
13     {
14         id = omp_get_thread_num();
15         for (i = 1+id; i <= num_steps; i=i+NUM_THREADS) {
16             x = (i - 0.5) * step;
17             sum += 4.0 / (1.0 + x * x);
18         }
19     }
20     pi = step * sum;
21     printf("pi=%f\n", pi);
22 }
```

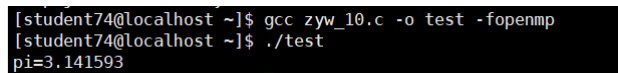
### 运行结果：

- 本地测试结果



```
Microsoft Visual Studio 调试控制台
pi=3.141593
```

- 华为服务器测试结果



```
[student74@localhost ~]$ gcc zyw_10.c -o test -fopenmp
[student74@localhost ~]$ ./test
pi=3.141593
```

### 结果分析：

通过 `for (i = 1+id; i <= num_steps; i=i+NUM_THREADS)` 来分配每个处理器计算的矩形id，然后每个线程并行计算，最后的和累加即可得到圆周率的估计值 `pi=3.141593`。



# 矩阵计算

## 实验目的：

比较二维数组按照行遍历的速度和按照列遍历的速度

## 实验原理：

程序中定义的数组存放在内存中，在CPU中设有高速缓冲区Cache, 高速缓存区的读写速度要比内存快3到10倍。当程序需要访存时，系统会将该元素连带紧跟着它后面的若干个元素（构成一个block）一起放入高速缓存区。在接下来的访存过程中会优先在缓冲区里寻找，如果找到则称为命中；如果没找到就要去存储器中寻找，称为未命中，运行速度相对较慢。

在二维数组中，其物理存储位置是按照行依次存储的。按行访问之所以快，就是因为程序在访问了第一个元素之后，接下来的连续若干元素都被存放在了缓冲区中，因此接下来的访问都会是命中的，访问速度变快。如果按照列遍历，由于数组的规模非常的大，缓冲区中难以装下很多的存储单元，因此在每一次的访问中命中率很低，需要到存储器操作，速度大大降低。

## 实验代码：

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <time.h>
4  #include <stdlib.h>
5
6  float array[5000][5000];
7  int main()
8  {
9      int i,j;
10     clock_t start1,start2,finish1,finish2;
11     double durement;
12     printf("traverse in row\n");
13     start1=clock();
14     for(i=0;i<5000;i++){
15         for(j=0;j<5000;j++){
16             array[i][j]=(i+j)*2;
17         }
18     }
19     finish1=clock();
20     durement=(double)(finish1-start1)/CLOCKS_PER_SEC;
21     printf("time: %f\n",durement);
22     printf("traverse in collumn\n");
23     start2=clock();
24     for(j=0;j<5000;j++){
25         for(i=0;i<5000;i++){
26             array[i][j]=(i+j)*2;
27         }
28     }
29     finish2=clock();
30     durement=(double)(finish2-start2)/CLOCKS_PER_SEC;
31     printf("time: %f\n",durement);
32     return 0;
33 }
```

## 运行结果：

- 本地运行结果

Microsoft Visual Studio 调试控制台

```
traverse in row
time: 0.202000
traverse in collumn
time: 0.276000
```

- 华为服务器运行结果

```
[student74@localhost ~]$ gcc zyw_6.c -o test -fopenmp
[student74@localhost ~]$ ./test
traverse in row
time: 0.099527
traverse in column
time: 0.176341
```

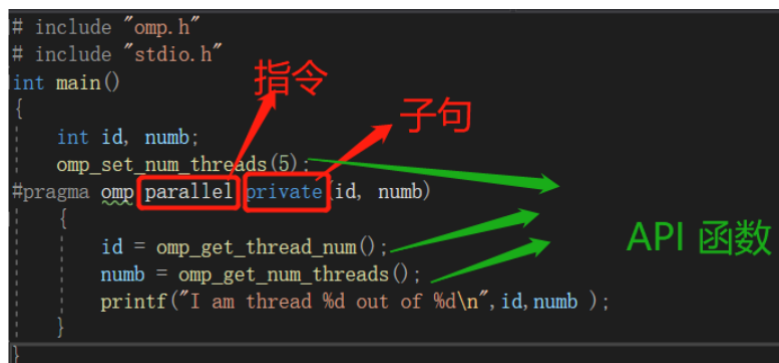
### 结果分析：

由上图可以，无论是在本地运行还是在服务器上运行，先按照行遍历的速度要大于先按照列遍历的速度，验证了其实验原理。

## 实验总结

OpenMP编程模型以线程为基础，通过编译制导指令制导并行化，有三种编程要素可以实现并行化控制，他们分别是编译制导、API函数集和环境变量。

其中编译制导指令以制导标识符 `#pragma omp` 开始，后边跟具体的功能指令，格式如：`#pragma omp 指令[子句[,子句] ...]`。



并行域制导就是指制导命令后的花括号中的区域。在并行域结尾有一个隐式同步（barrier）。

子句（clause）用来说明并行域的附加信息。C/C++子句间用空格分开。

一个并行域就是一个能被多个线程并行执行的程序段，因此该代码段中的输出语句可以有多个线程同时完成。并行域的基本的格式如下：

```
1 | #pragma omp parallel [clauses]
2 | {
3 |   BLOCK
4 | }
```

## 功能指令

- `parallel`：用在一个结构块之前，表示这段代码将被多个线程并行执行；
- `for`：用于for循环语句之前，表示将循环计算任务分配到多个线程中并行执行，以实现任务分担，必须由编程人员自己保证每次循环之间无数据相关性；
- `parallel for`：parallel和for指令的结合，也是用在for循环语句之前，表示for循环体的代码将被多个线程并行执行，它同时具有并行域的产生和任务分担两个功能；
- `sections`：用在可被并行执行的代码段之前，用于实现多个结构块语句的任务分担，可并行执行的代码段各自用section指令标出（注意区分sections和section）；
- `parallel sections`：parallel和sections两个语句的结合，类似于parallel for；
- `single`：用在并行域内，表示一段只被单个线程执行的代码；
- `critical`：用在一段代码临界区之前，保证每次只有一个OpenMP线程进入；

- `flush`：保证各个OpenMP线程的数据影像的一致性；
- `barrier`：用于并行域内代码的线程同步，线程执行到barrier时要停下等待，直到所有线程都执行到barrier时才继续往下执行；
- `atomic`：用于指定一个数据操作需要原子性地完成；
- `master`：用于指定一段代码由主线程执行；
- `threadprivate`：用于指定一个或多个变量是线程专用，后面会解释线程专有和私有的区别。

## 子句

- `private`：指定一个或多个变量在每个线程中都有它自己的私有副本；
- `firstprivate`：指定一个或多个变量在每个线程都有它自己的私有副本，并且私有变量要在进入并行域或任务分担域时，继承主线程中的同名变量的值作为初值；
- `lastprivate`：是用来指定将线程中的一个或多个私有变量的值在并行处理结束后复制到主线程中的同名变量中，负责拷贝的线程是for或sections任务分担中的最后一个线程；
- `reduction`：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归约运算，并将结果返回给主线程同名变量；
- `nowait`：指出并发线程可以忽略其他制导指令暗含的路障同步；
- `num_threads`：指定并行域内的线程的数目；
- `schedule`：指定for任务分担中的任务分配调度类型；
- `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任务分担域内指定代码段需要按照串行循环次序执行；
- `copyprivate`：配合single指令，将指定线程的专有变量广播到并行域内其他线程的同名变量中；
- `copyin`：用来指定一个threadprivate类型的变量需要用主线程同名变量进行初始化；
- `default`：用来指定并行域内的变量的使用方式，缺省是shared。

## API函数

- `omp_in_parallel`：判断当前是否在并行域中
- `omp_get_thread_num`：返回线程号
- `omp_set_num_threads`：设置后续并行域中的线程格式
- `omp_get_num_threads`：返回当前并行域中的线程数
- `omp_get_max_threads`：获得并行域中可用的最大线程数
- `omp_get_num_procs`：返回系统中处理器的个数
- `omp_get_dynamic`：判断是否支持动态改变线程的数目
- `omp_set_dynamic`：启动或者关闭线程数目的动态改变
- `omp_get_nested`：判断系统是否支持并行嵌套
- `omp_set_nested`：启动或者关闭并行嵌套
- `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

## 环境变量

- `OMP_SCHEDULE`：用于for循环并行化后的调度，它的值就是循环调度的类型；
- `OMP_NUM_THREADS`：用于设置并行域中的线程数；
- `OMP_DYNAMIC`：通过设定变量值，来确定是否允许动态设定并行域内的线程数；
- `OMP_NESTED`：指出是否可以并行嵌套

