并行编程实验设计文档

实验项目名称：OpenMp并行实验

姓名：江家玮

班级：计科2204班

学号：22281188

自我评价：

在本次实验中，我通过对代码的调试，我逐渐将ppt抽象的概念逐渐具象化，加深了我对OpenMp的并行实验的认识，也将之前未搞清楚OpenMp的private Variables逐渐弄清晰，通过实验和不断搜索相关文档，理解了其私有变量和共享变量的具体使用方法。同时也明白OpenMp中的section和reduction的使用。也具体操作了OpenMp中single,barrier和critical，atomic，理解了竞争现象。

成绩：

北京交通大学计算机与信息技术学院

一、 实验一 Parallel Construct

```
int main()
{
    int id, numb;
    omp_set_num_threads(3);
    #pragma omp parallel private(id, numb)
    {
    id = omp_get_thread_num();
    numb = omp_get_num_threads();
    printf("I am thread %d out of %d \n", id, numb);
    }
}
```

运行结果

```
I am thread  0 out of  3
I am thread  1 out of  3
I am thread  2 out of  3
```

运行结果分析

此为OpenMP的并行程序，omp_set_num_threads(3);代表并行区域中要使用的线程数量为3。则在 #pragma omp parallel 之后的代码块中将会有3个线程并行执行

在OpenMP库中的 omp_get_thread_num() 函数来获取当前线程的线程号，以及 omp_get_num_threads() 函数来获取并行区域中的线程总数。

最后打印出当前线程的线程号和并行区域中的线程总数。

二、 实验二 Worksharing Construct

```
omp_set_num_threads(3);
#pragma omp parallel
 {
    printf("The number of threads: %d seen by thread %d\n",
            omp_get_thread_num() , omp_get_num_threads());
    #pragma omp for
    for( int i = 1; i <= 5; ++i ){
        printf( " No. %d iteration by thread %d\n",i,omp_get_thread_num() );
    }
 }
```

运行结果

```
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: 1 seen by thread 3
 No. 3 iteration by thread 1
 No. 4 iteration by thread 1
The number of threads: 2 seen by thread 3
 No. 5 iteration by thread 2
```

运行结果分析：

首先每个线程打印出它们各自的线程号和并行区域中的线程总数，然后每个线程按顺序执行 for 循环的迭代，每次迭代都打印出当前迭代次数以及执行该迭代的线程号。

三、 实验三 Combined Parallel Worksharing Constructs

```c
#include <stdio.h>
#include <omp.h>
int main() {
    #pragma omp parallel for
    for(int i = 0; i < 10; ++i) {
        printf(" No. %d iteration by thread %d\n",i,omp_get_thread_num() );
    }
    return 0;
}
```

运行结果

```
 No. 0 iteration by thread 0
 No. 9 iteration by thread 9
 No. 3 iteration by thread 3
 No. 1 iteration by thread 1
 No. 2 iteration by thread 2
 No. 8 iteration by thread 8
 No. 4 iteration by thread 4
 No. 6 iteration by thread 6
 No. 7 iteration by thread 7
 No. 5 iteration by thread 5
```

运行结果分析

由于是通过并行的方式去执行这个程序，则不同的迭代被不同的线程执行，从而迭代的执行顺序可能不同，因此导致输出的结果会有不同

四、 实验四 Combined Parallel Section Constructs

```
#include <stdio.h>
#include <omp.h>
int main() {
    #pragma omp parallel sections
    {
        #pragma omp section
        for (int i = 0; i < 5; ++i) {
            printf( “section i : iteration %d by thread no. %d\n”,   i, omp_get_thread_num());
        }
        #pragma omp section
        for (int j = 0; j < 5; ++j) {
            printf( “ section j : iteration %d by thread no. %d\n”, j, omp_get_thread_num());
        }

    return 0;
}
```

运行结果

```
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.  3
section j : iteration  1 by thread no.  3
section j : iteration  2 by thread no.  3
section j : iteration  3 by thread no.  3
section j : iteration  4 by thread no.  3
```

#pragma omp parallel section创建了一个并行部分，其中的 section 将会并行执行。#pragma omp section用于标识一个并行部分的区域，则将会在一个单独的线程中并行执行。

由于每个 section 都是在一个单独的线程中并行执行，因此会有两个线程分别执行两个 section 中的 for 循环。

五、 实验五 Synchronization Constructs

```
#include <stdio.h>
#include <omp.h>
int main() {
    #pragma omp parallel
    {
    for (int i = 0; i < 10; ++i) {
        printf( “loop i : iteration %d by thread no. %d\n”, i,omp_get_thread_num() );
    }
    #pragma omp barrier
    for (int j = 0; j < 10; ++j) {
        printf( “loop  j : iteration %d by thread no. %d\n”, j,omp_get_thread_num() );
    }
    }
    return 0;
}
```

```
loop i : iteration  0 by thread no.  4
loop i : iteration  1 by thread no.  4
loop i : iteration  0 by thread no.  5
loop i : iteration  0 by thread no.  6
loop i : iteration  1 by thread no.  6
loop i : iteration  0 by thread no.  12
loop i : iteration  0 by thread no.  13
loop i : iteration  1 by thread no.  13
loop i : iteration  2 by thread no.  13
loop i : iteration  0 by thread no.  14
loop i : iteration  1 by thread no.  14
loop i : iteration  2 by thread no.  14
loop i : iteration  3 by thread no.  14
loop i : iteration  4 by thread no.  14
loop i : iteration  5 by thread no.  14
loop i : iteration  6 by thread no.  14
loop i : iteration  7 by thread no.  14
loop i : iteration  8 by thread no.  14
loop i : iteration  9 by thread no.  14
loop i : iteration  0 by thread no.  10
loop i : iteration  1 by thread no.  10
loop i : iteration  2 by thread no.  10
loop i : iteration  3 by thread no.  10
loop i : iteration  4 by thread no.  10
loop i : iteration  5 by thread no.  10
loop i : iteration  6 by thread no.  10
loop i : iteration  7 by thread no.  10
loop i : iteration  8 by thread no.  10
loop i : iteration  9 by thread no.  10
loop i : iteration  2 by thread no.  4
```

运行结果分析:

#pragma omp parallel：创建一个并行区域，多个线程会在这里同时执行代码块中的内容。

#pragma omp barrier：在所有线程都完成了前面的循环后才会继续执行后面的代码，它会创建一个同步点，确保前面的循环全部执行完成。

iteration i完成后才开始执行iteration j.

六、 实验六 Critical vs. Atomic

1,Critical

```
void critical() {
    int x = 0;
    #pragma omp parallel shared(x)
    {
        #pragma omp critical
        x += 1;
    }
    printf("x = %d\n", x);
}
```

Atomic

```
void atomic()
{
    int x = 0;
    #pragma omp parallel shared(x)
    {
        #pragma omp atomic
        x++;
    }
    printf("x = %d\n", x);
}
```

运行结果

```
x = 16
x = 16
```

运行结果分析

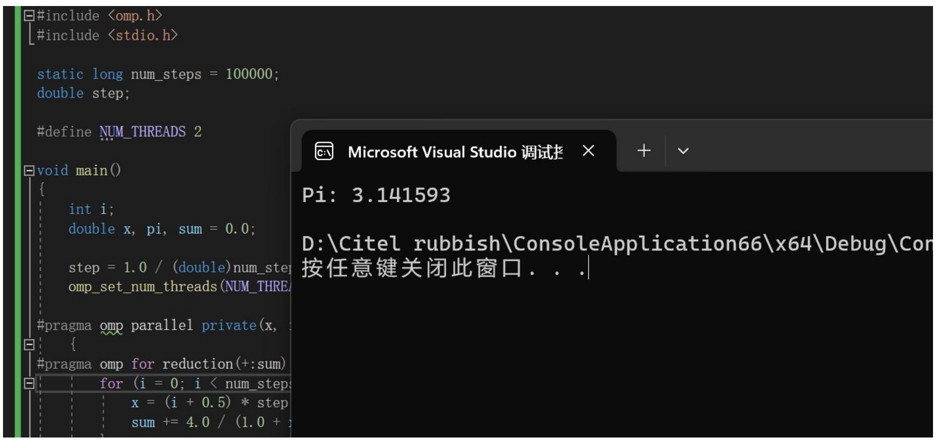
通过使用 critical 和 atomic 两个函数来处理共享变量的方式。#pragma omp atomic：这个指令将 x++ 这行代码标记为原子操作，确保在并行执行中每次只有一个线程能够执行该操作。#pragma omp critical：这个指令将 x += 1; 这行代码标记为临界区，确保同时只有一个线程可以执行这个操作。

七、 实验七 Variable and reduction

```
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{
    int i, id;
    double x, pi, sum;
    step = 1.0 / (double) num_steps;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel private(x, i, id) reduction(+:sum)
{
    id = omp_get_thread_num();
    for(i = id + 1; i <= num_steps; i = i + NUM_THREADS) {
        x = (i - 0.5) * step;
        sum = sum + 4.0 / (1.0 + x*x);
    }
}
    pi = sum * step;
}
```

由于无法直接执行该程序，简单更改代码更改如下：

```
#include <omp.h>
#include<stdio.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main()
{
    int i, id;
    double x, pi, sum = 0.0;
    step = 1.0 / (double)num_steps;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel private(x, i, id) reduction(+:sum)
    {
        id = omp_get_thread_num();
        for (i = id + 1; i <= num_steps; i = i + NUM_THREADS) {
            x = (i - 0.5) * step;
            sum = sum + 4.0 / (1.0 + x * x);
        }
    }
    pi = sum * step;
    printf("pi:%f\n", pi);
}
```



实验结果分析：

#pragma omp parallel private(x, i, id) reduction(+:sum) 指令创建了并行区域。private(x, i, id) 指定了每个线程私有的变量，而 reduction(+:sum) 则指定了 sum 变量在并行区域中归约操作，以确保多个线程能够正确地累加计算结果。

在并行区域内，每个线程通过 omp_get_thread_num() 获取线程编号 id。然后，通过 for 循环，每个线程从自己的起始索引 id + 1 开始，以步长为 NUM_THREADS 进行迭代计算。在每次迭代中，根据当前的 i 和步长 step 计算对应的 x 值，通过公式 sum = sum + 4.0 / (1.0 + x * x) 计算部分圆周率的近似值。