**实验报告**

课程名称：

学生姓名：

学号：

指导老师：

实验2：OpenMP 进阶实验

1. **实验目的**

掌握生产者-消费者模型，具备运用OpenMP相关知识进行综合分析，可实现实际工程背景下生产者-消费者模型的线程级负责均衡规划和调优。

1. **实验要求**
2. single与master语句

制导语句 single 和 master 都是指定相关的并行区域只由一个线程执行，区别在于使用 master 则由主线程（0 号线程）执行，使用 single 则由运行时的具体情况决定。两者还有一个区别是 single 在结束处隐含栅栏同步，而 master 没有。在没有特殊需求时，建议使用 single 语句。

程序代码见程序2-1

1. barrier语句

在多线程编程中必须考虑到不同的线程对同一个变量进行读写访问引起的数据竞争问题。如果线程间没有互斥机制，则不同线程对同一变量的访问顺序是不确定的，有可能导致错误的执行结果。OpenMP中有两种不同类型的线程同步机制，一种是互斥机制，一种是事件同步机制。其中事件同步机制的设计思路是控制线程的执行顺序，可以通过设置barrier同步路障实现。

1. atomic、critical与锁

通过 critical 临界区实现的线程同步机制也可以通过原子（atomic）和锁实现。后两者功能更具特点，并且使用更为灵活。

程序代码见程序2-2、2-3、2-4

1. schedule语句

在使用 parallel 语句进行累加计算时是通过编写代码来划分任务，再将划分后的任务分配给不同的线程去执行。后来使用 paralle for 语句实现是基于 OpenMP 的自动划分，如果有 n 次循环迭代 k 个线程，大致会为每一个线程分配[n/k]各迭代。由于 n/k 不一定是整数，所以存在轻微的负载均衡问题。我们可以通过子句 schedule 来对影响负载的调度划分方式进行设置。

1. 循环依赖性检查

以对π 的数值估计的方法为例子来探讨 OpenMP 中的循环依赖问题。圆周率π（Pi）是数学中最重要和最奇妙的数字之一，对它的计算方法也是多种多样，其中适合采用计算机编程来计算并且精确度较高的方法是通过使用无穷级数来计算π 值。常用的级数有两个，格雷戈里 -莱布尼茨无穷级数和 Nilakantha 级数，前者公式简单，但后者精度更高。

程序代码见程序2-5

1. **实验程序及结果截图**

2-1

#include<iostream>

#include<cstdlib>

#include<omp.h>

using namespace std;

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

{

#pragma omp parallel num\_threads(4)

{

cout<<"thread "<<omp\_get\_thread\_num()<<" in line "<< \_\_LINE\_\_<<endl;

#pragma omp master

cout<<"[master]thread "<<omp\_get\_thread\_num()<<" in line "<<\_\_LINE\_\_<<endl;

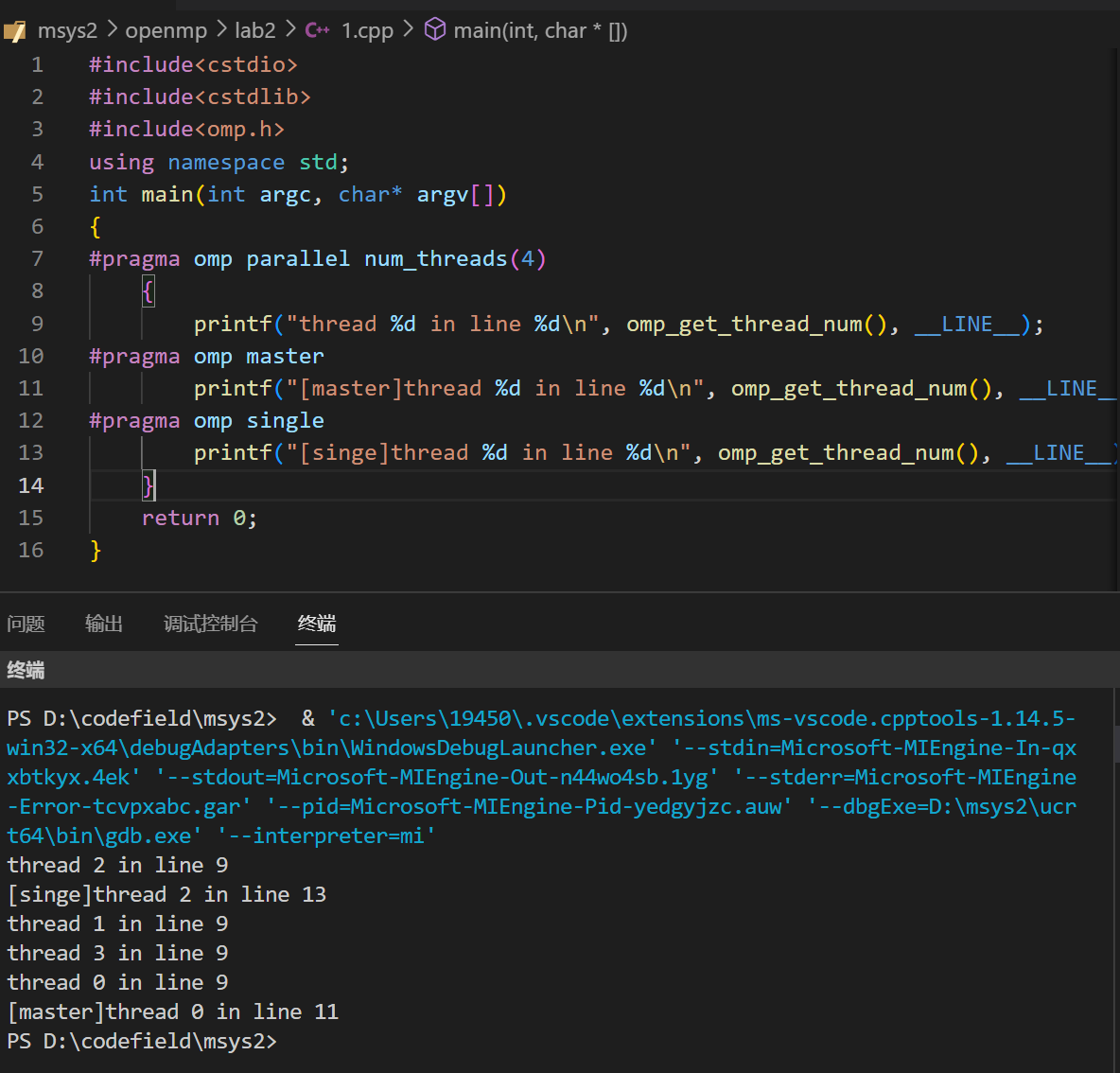
#pragma omp single

cout<<"[singe]thread "<<omp\_get\_thread\_num()<<" in line "<<\_\_LINE\_\_<<endl;

}

return 0;

}



2-2

#include<cstdio>

#include<cstdlib>

#include<omp.h>

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

{

int sumx = 1;

int sumy = 2;

#pragma omp parallel num\_threads(4) shared(sumx, sumy)

{

#pragma omp atomic

sumx += omp\_get\_thread\_num();

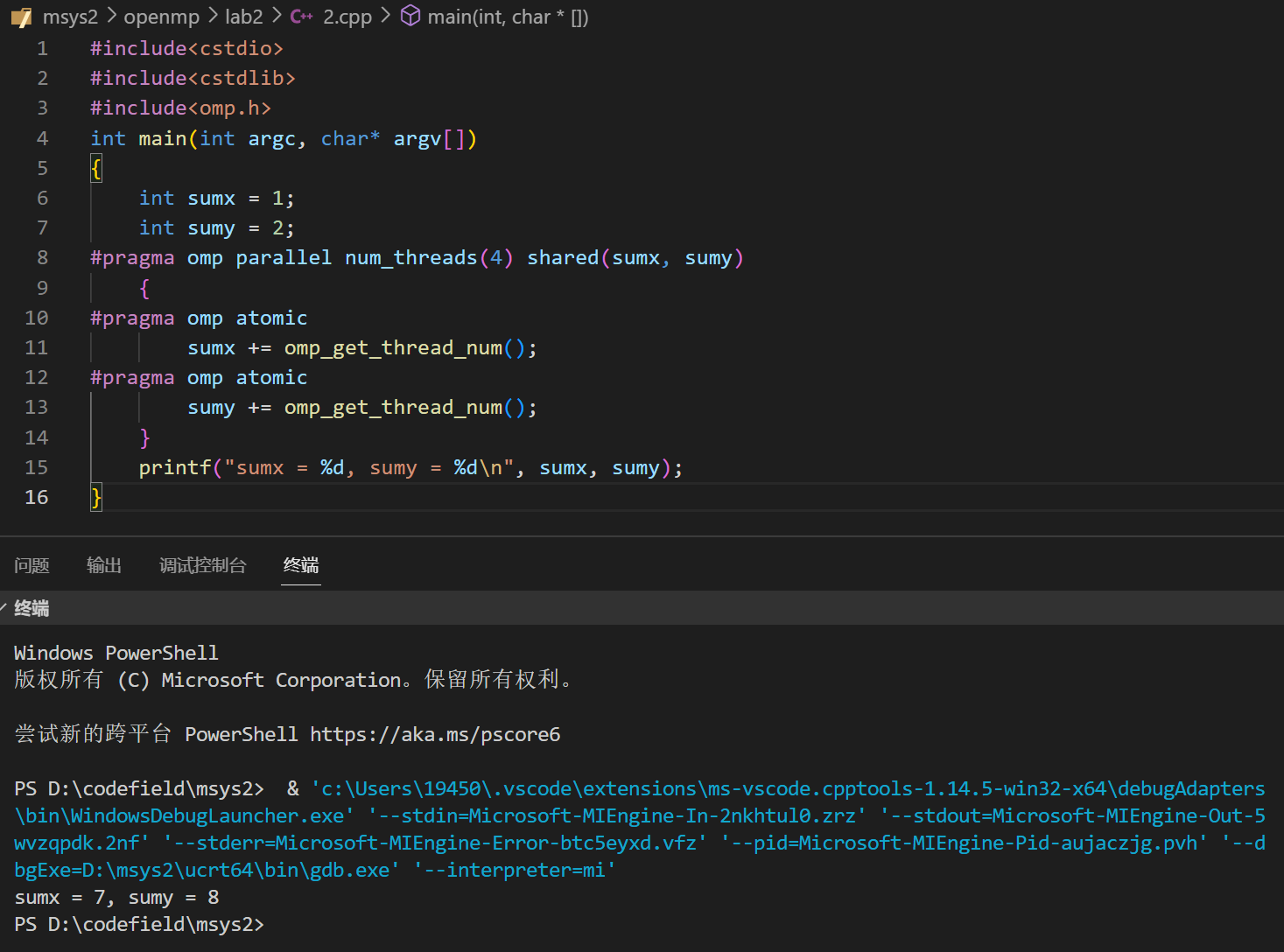
#pragma omp atomic

sumy += omp\_get\_thread\_num();

}

printf("sumx = %d, sumy = %d\n", sumx, sumy);

}



2-3

#include<stdio.h>

#include<stdlib.h>

#include<omp.h>

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

{

int sumx = 1;

int sumy = 2;

omp\_lock\_t lock;

omp\_init\_lock(&lock);

#pragma omp parallel num\_threads(4) shared(sumx, sumy)

{

omp\_set\_lock(&lock);

sumx += omp\_get\_thread\_num();

sumy += omp\_get\_thread\_num();

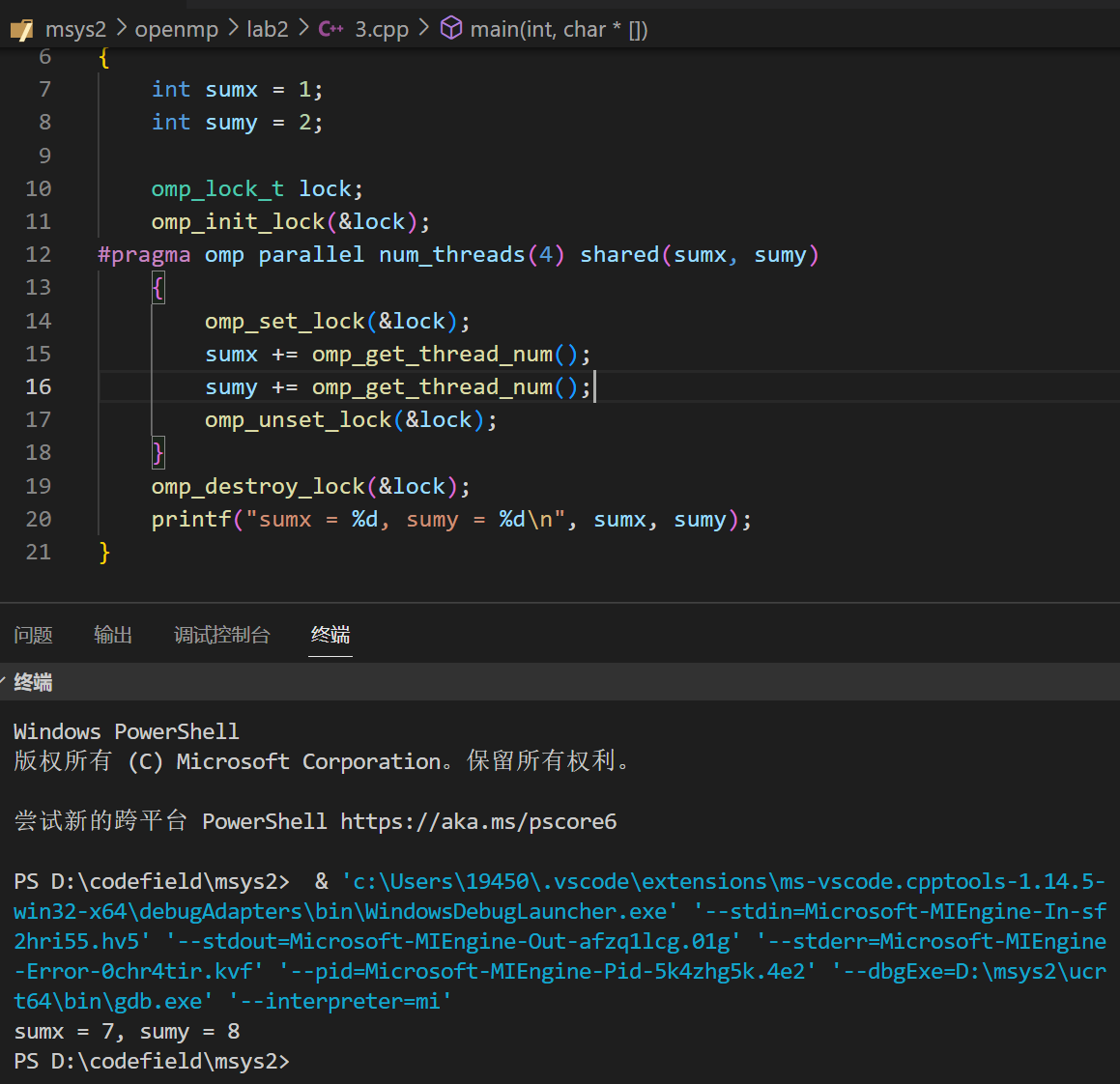
omp\_unset\_lock(&lock);

}

omp\_destroy\_lock(&lock);

printf("sumx = %d, sumy = %d\n", sumx, sumy);

}



2-4

#include<cstdio>

#include<cstdlib>

#include<cmath>

#include<omp.h>

typedef long long ll;

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

{

ll n, i;

double factor = 1.0;

double sum = 0.0;

n = strtoll(argv[1], NULL, 10);

for (int i = 0; i < n; i++)

{

sum += factor / ((2 \* i + 2) \* (2 \* i + 3) \* (2 \* i + 4));

factor = -factor;

}

sum = 3.0 + 4.0 \* sum;

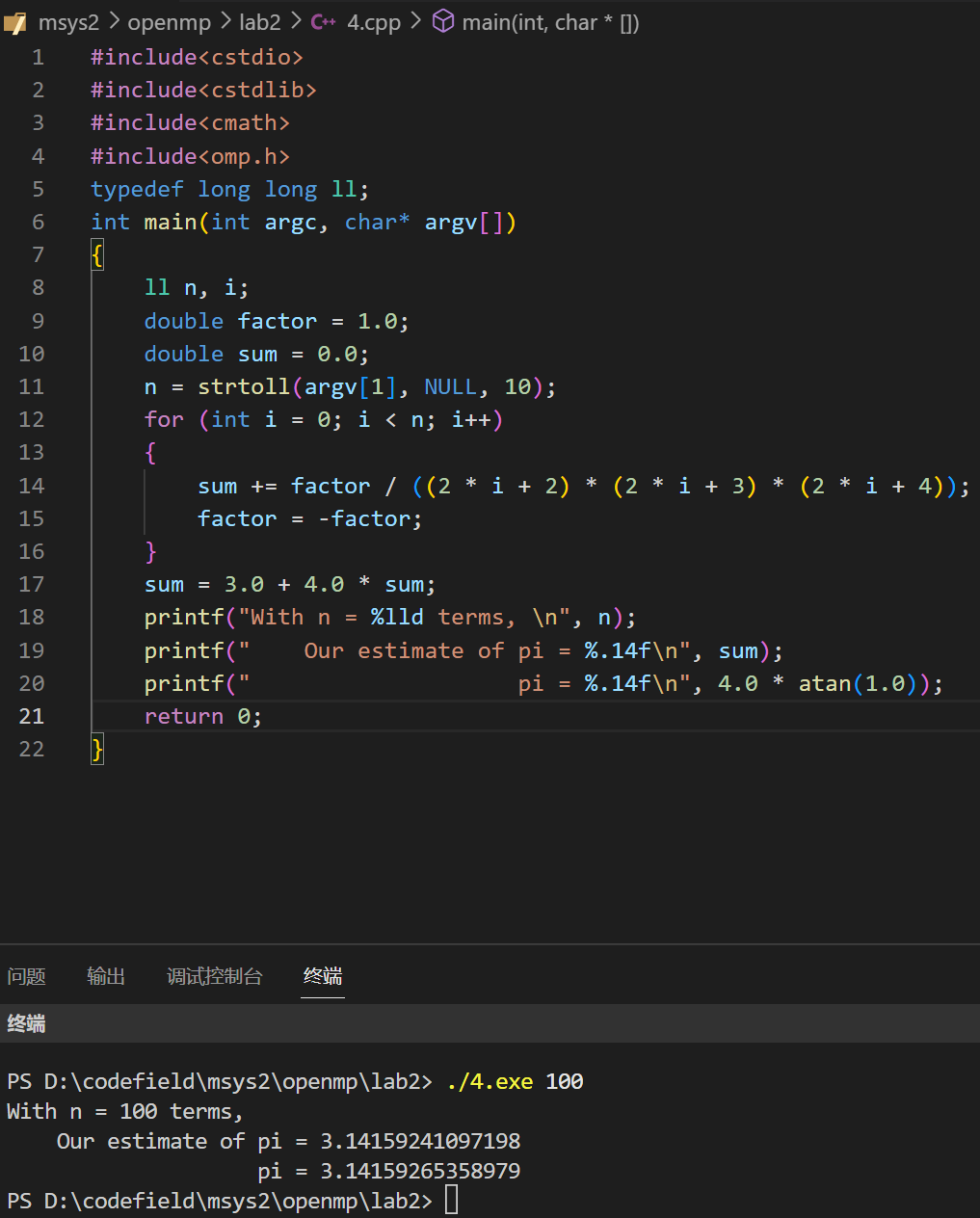
printf("With n = %lld terms, \n", n);

printf(" Our estimate of pi = %.14f\n", sum);

printf(" pi = %.14f\n", 4.0 \* atan(1.0));

return 0;

}



2-5

#include<cstdio>

#include<cstdlib>

#include<cmath>

#include<omp.h>

typedef long long ll;

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

{

ll n, i;

int count\_t;

double factor = 1.0;

double sum = 0.0;

n = strtoll(argv[1], NULL, 10);

count\_t = strtol(argv[2], NULL, 10);

#pragma omp parallel for num\_threads(count\_t) \

reduction(+: sum)

for ( i = 0; i < n; i++)

{

sum += factor / ((2 \* i + 2) \* (2 \* i + 3) \* (2 \* i + 4));

factor = -factor;

}

sum = 3.0 + 4.0 \* sum;

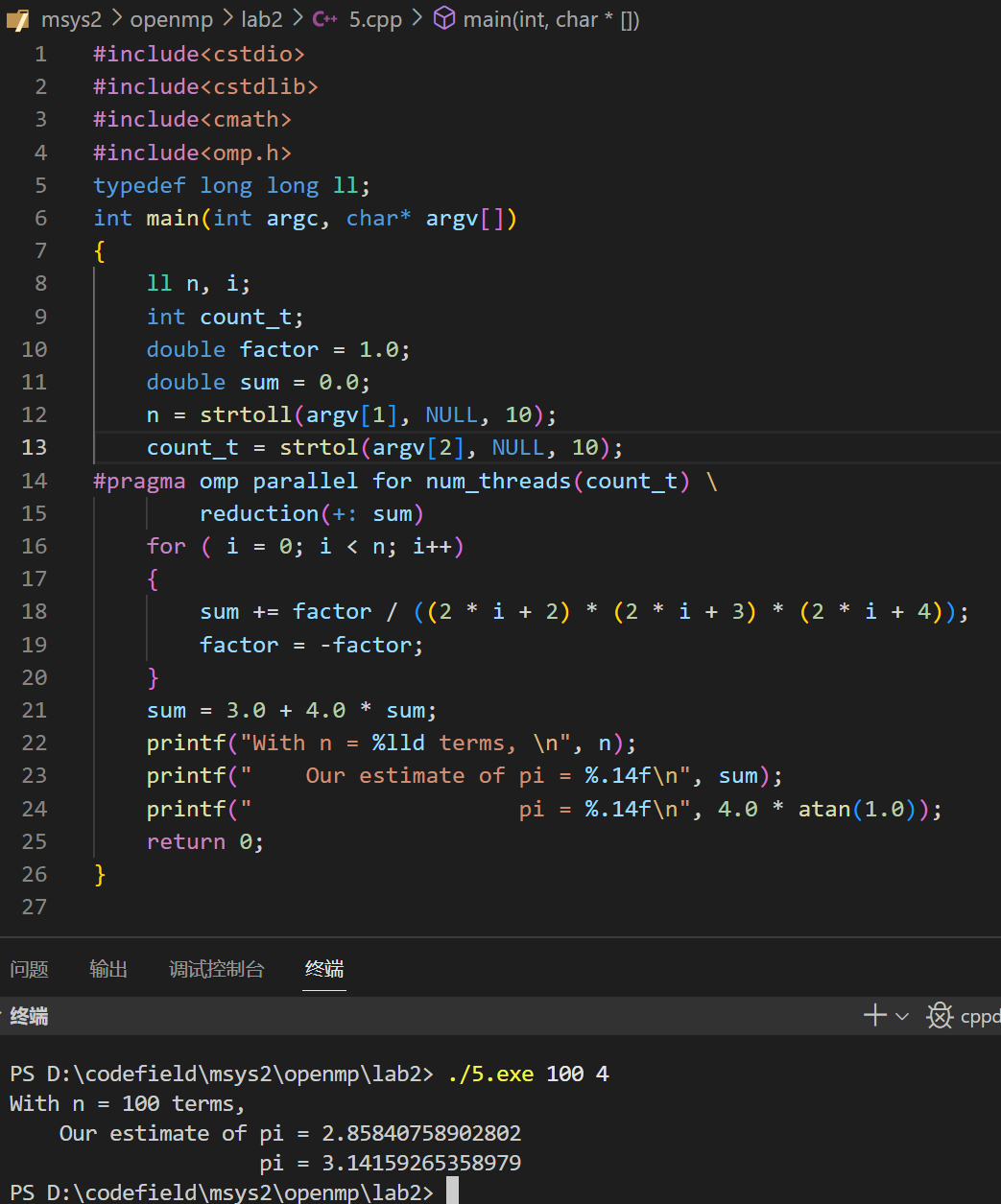
printf("With n = %lld terms, \n", n);

printf(" Our estimate of pi = %.14f\n", sum);

printf(" pi = %.14f\n", 4.0 \* atan(1.0));

return 0;

}



2-6

#include<cstdio>

#include<cstdlib>

#include<cmath>

#include<omp.h>

typedef long long ll;

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

{

ll n, i;

int count\_t;

double factor = 1.0;

double sum = 0.0;

n = strtoll(argv[1], NULL, 10);

count\_t = strtol(argv[2], NULL, 10);

#pragma omp parallel for num\_threads(count\_t) \

reduction(+: sum) private(factor)

for ( i = 0; i < n; i++)

{

factor = (i % 2 == 0) ? 1.0 : -1.0;

sum += factor / ((2 \* i + 2) \* (2 \* i + 3) \* (2 \* i + 4));

}

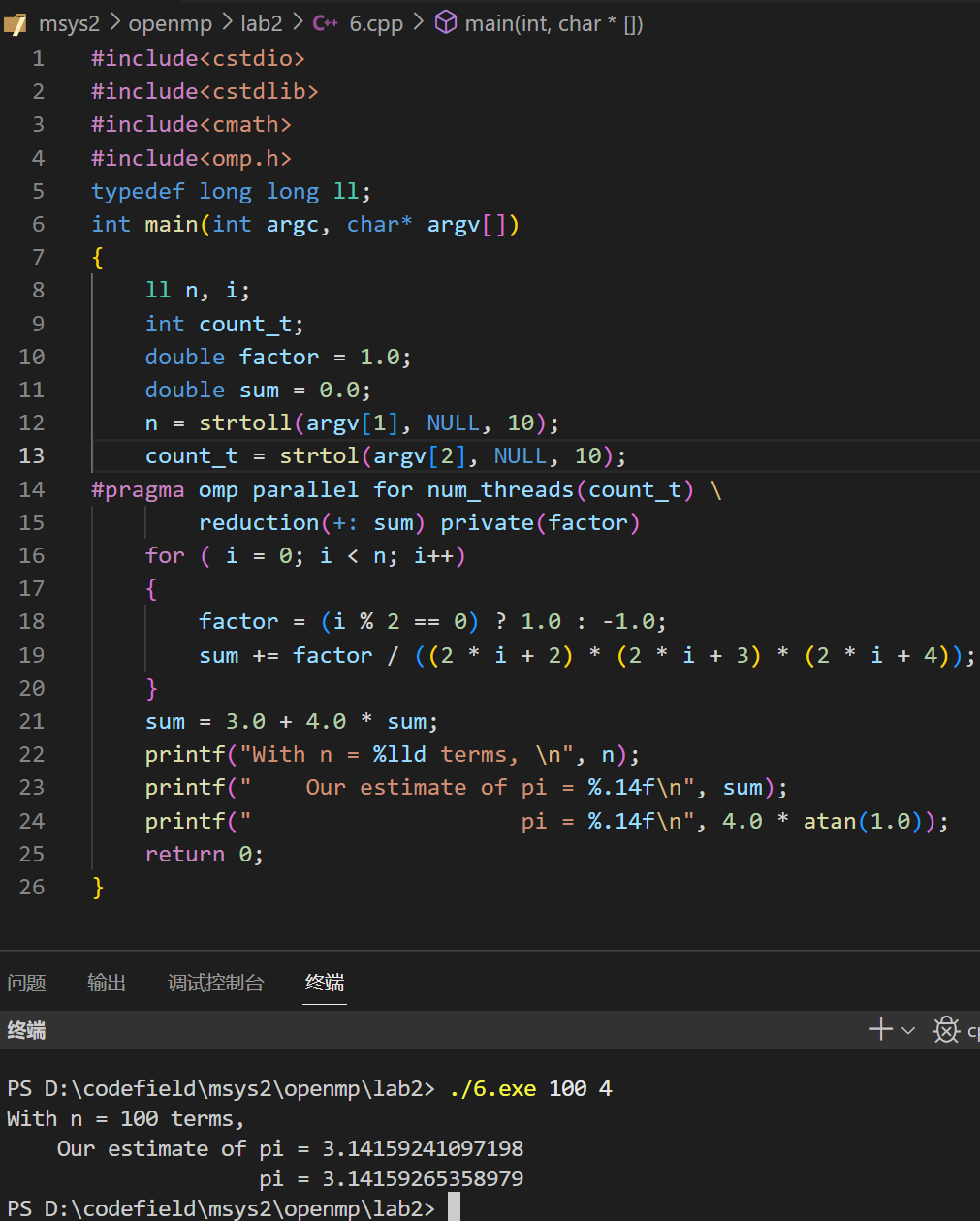
sum = 3.0 + 4.0 \* sum;

printf("With n = %lld terms, \n", n);

printf(" Our estimate of pi = %.14f\n", sum);

printf(" pi = %.14f\n", 4.0 \* atan(1.0));

return 0;

}

1. **总结**

谈谈实验中遇到的问题及如何解决的，在实验中的一些个人体会。

通过本次实验，我深刻了解了生产者-消费者模型的线程级负责均衡规划和调优。同时，我也学会了使用OpenMP相关知识进行综合分析，如制导语句single和master语句、barrier语句、atomic、critical与锁、schedule语句等。此外，我还了解了循环依赖性检查的重要性，以及如何使用无穷级数来计算圆周率π的值。这次实验让我对多线程编程有了更深入的认识和理解。