**实验报告**

课程名称：

学生姓名：

学号：

指导老师：

实验1：OpenMP 基础实验

1. **实验目的**
2. 了解OpenMP的运行环境
3. 掌握OpenMP编程的基本要素、编译方法，可运用相关知识独立完成一个基本的OpenMP程序的编写与调试过程。
4. **实验要求**
5. 掌握OpenMP运行环境

在ubuntu环境中打开一个终端界面。尝试在图形操作界面左侧寻找终端的图标进行点击，或直接使用快捷键**Ctrl+Alt+T**打开终端界面进行Shell环境。

1. 运行一个简单OpenMP程序

程序代码见程序1-1、1-2

1. OpenMP兼容性检查

通过检查预处理宏\_OPENMP 是否定义来进行条件编 译。如果定义了\_OPENMP，则包含omp.h 并调用OpenMP 库函数。

程序代码见程序1-3

1. 常用线程操作库函数语句

在OpenMP编程过程中，一旦涉及线程操作，有较大的概率使用三个常用的库函数，分别为：

(1) int omp\_get\_num\_threads(void) 获取当前线程组（team）的线程数量，

如果不在并行区调用，则返回1。

(2) int omp\_get\_thread\_num(void) 返回当前线程号。

(3) int omp\_get\_num\_procs(void) 返回可用的处理核个数。

注意区别这三个库函数的外形及意义，特别是前两个库函数，初始使用时很容易混淆。

程序代码见程序1-4

1. parallel语句的练习

parallel 用来构造一个并行区域，在这个区域中的代码会被多个线程（线程组）执行，在区域结束处 有默认的同步（隐式路障）。 我们可以在parallel 构造区域内使用分支语句，通过omp\_get\_thread\_num 获得的当前线程编号来指定不同线程执行区域内的不同代码。

程序代码见程序1-5、1-6

1. critical和reducation语句的练习

为了保证在多线程执行的程序中，出现资源竞争的时候能得到正确结果， OpenMP 提供了3种不同类型的多线程同步机制：排它锁、原子操作和临界区。其中临界区是最简单最常见，也是最不灵活效率最低的一种。 编译制导critical 声明一个临界区，临界区一次只允许一个线程执行，因此它在并行代码提供了一种串行的机制。critical 结束处没有隐式路障。

程序代码见程序1-7、1-8、1-9、1-10

1. parallel for语句的练习

我们在最求自动并行处理的道路上还想取得进一步的优化效果。使用OpenMP 的目的就是要尽可能地降低并行编程的难度。从程序1-10中我们可以看到，对于累加数据分配到线程的操作是通过线程编号计算得到的，能不能让OpenMP 自动去处理。这就是parallel for 子句的作用。

程序代码见程序1-11

1. **实验程序及结果截图**

1-1

#include<iostream>

#include<cstdlib>

using namespace std;

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

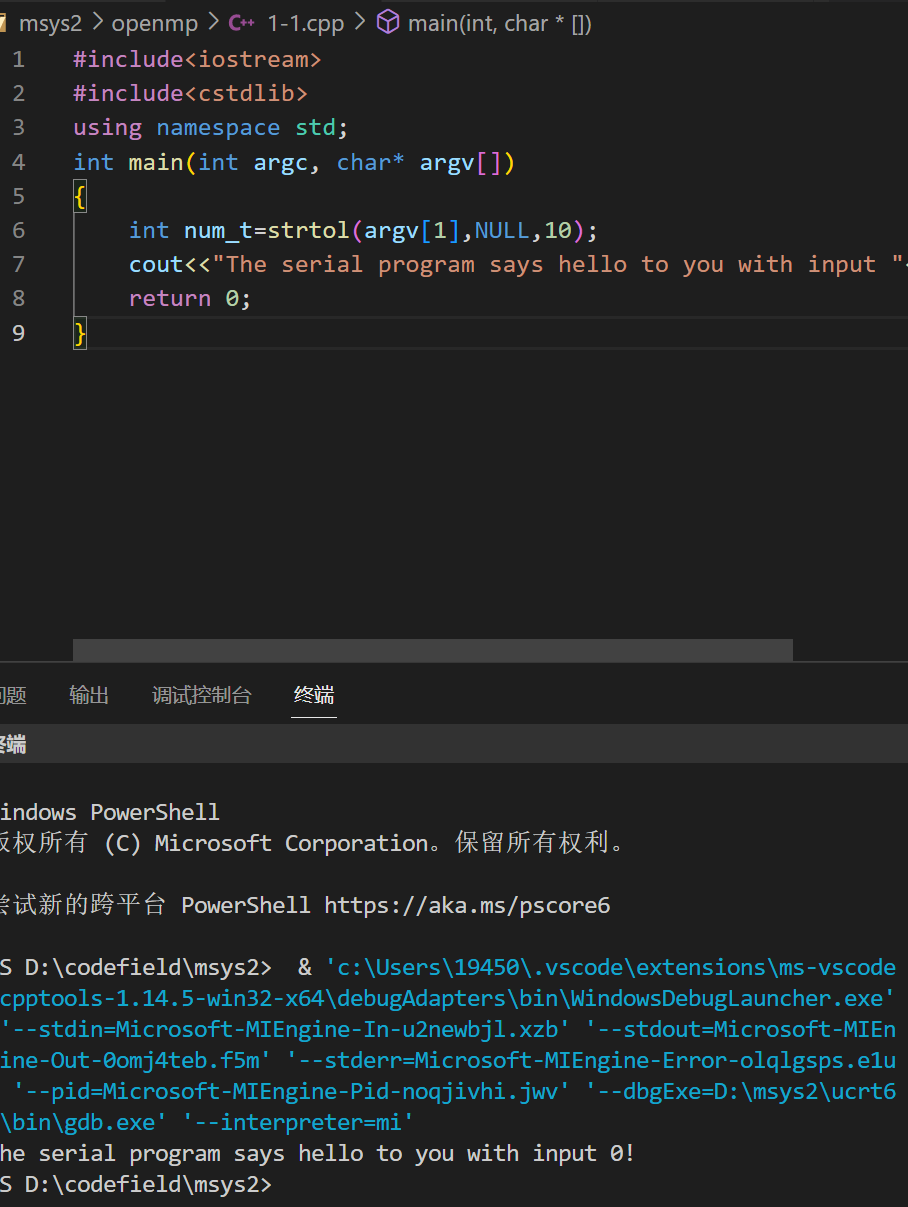
{

int num\_t=strtol(argv[1],NULL,10);

cout<<"The serial program says hello to you with input "<<num\_t<<"!\n";

return 0;

}



1-2

#include<iostream>

#include<cstdlib>

#include<omp.h>

using namespace std;

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

{

int num\_t=strtol(argv[1],NULL,10);

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

{

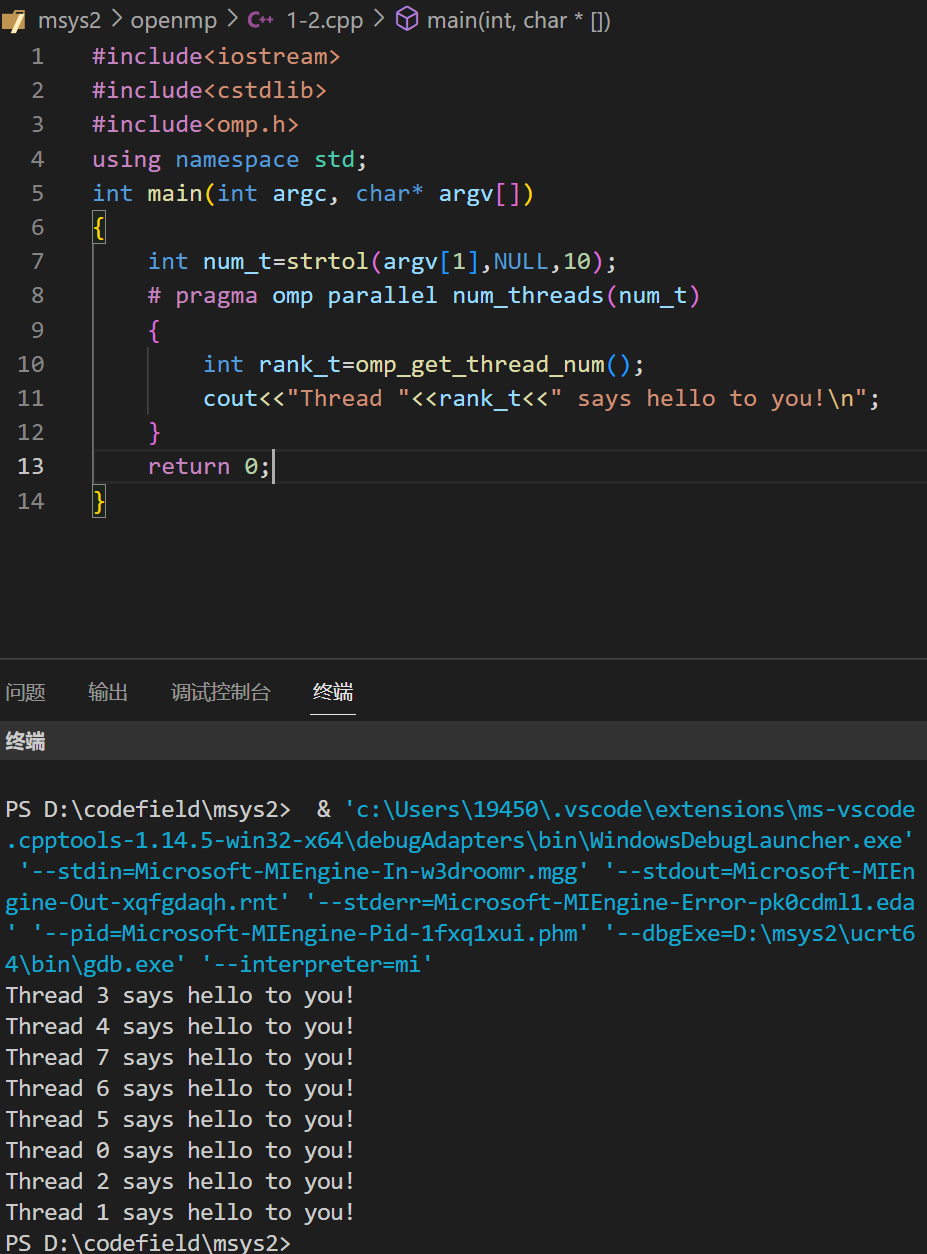
int rank\_t=omp\_get\_thread\_num();

cout<<"Thread "<<rank\_t<<" says hello to you!\n";

}

return 0;

}



1-3

#include<iostream>

#include<cstdlib>

#ifdef \_OPENMP

#include<omp.h>

#endif

using namespace std;

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

{

int num\_t=strtol(argv[1],NULL,10);

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

{

# ifdef \_OPENMP

int rank\_t=omp\_get\_thread\_num();

# else

int rank\_t=0;

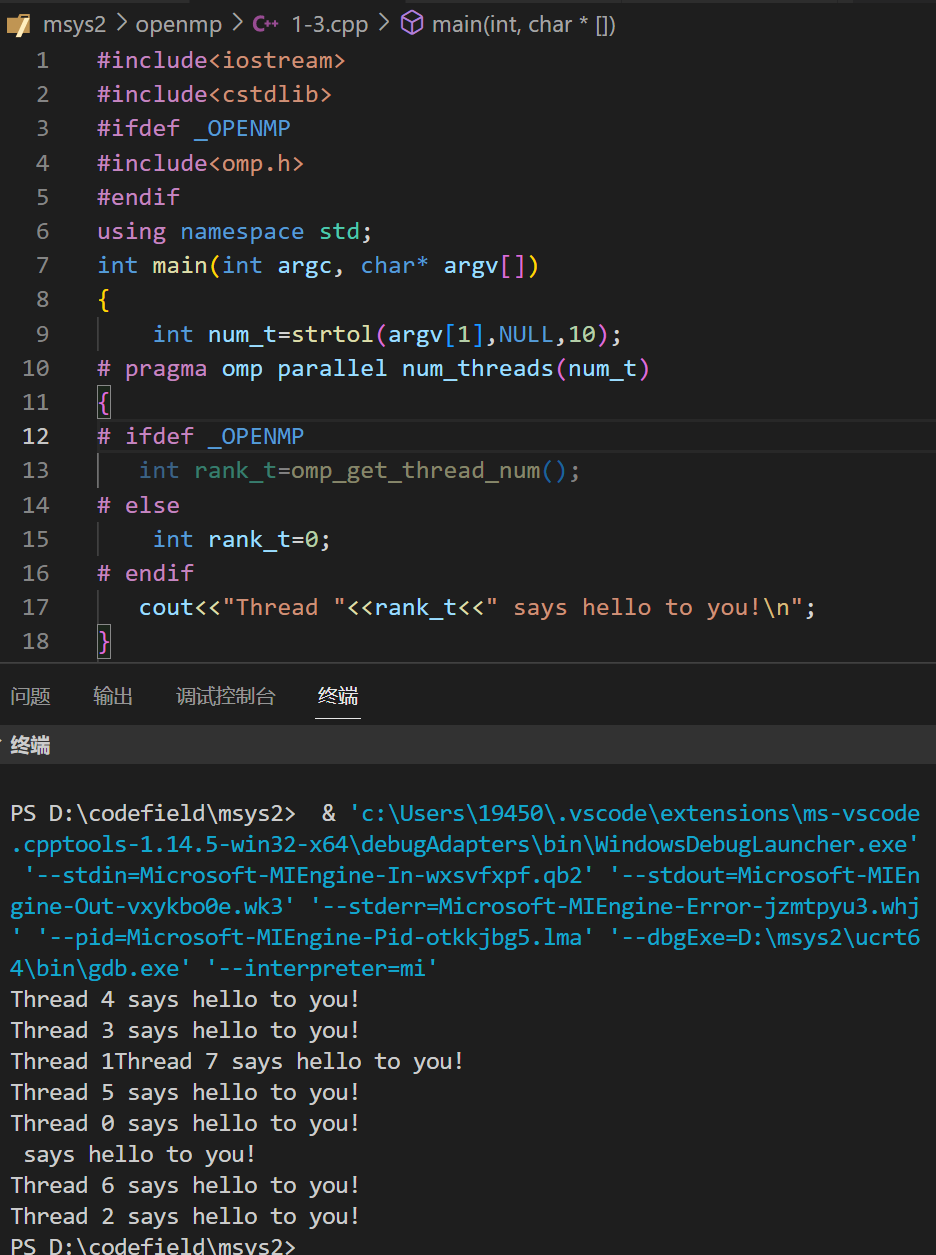
# endif

cout<<"Thread "<<rank\_t<<" says hello to you!\n";

}

return 0;

}



1-4

#include<iostream>

#include<cstdlib>

#include<omp.h>

using namespace std;

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

{

int num\_t=strtol(argv[1],NULL,10);

cout<<"proc count "<<omp\_get\_num\_procs()<<" \n";

cout<<"thread count "<<omp\_get\_num\_threads()<<" \n";

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

{

cout<<"Thread "<<omp\_get\_thread\_num()<<" says hello to you!\n";

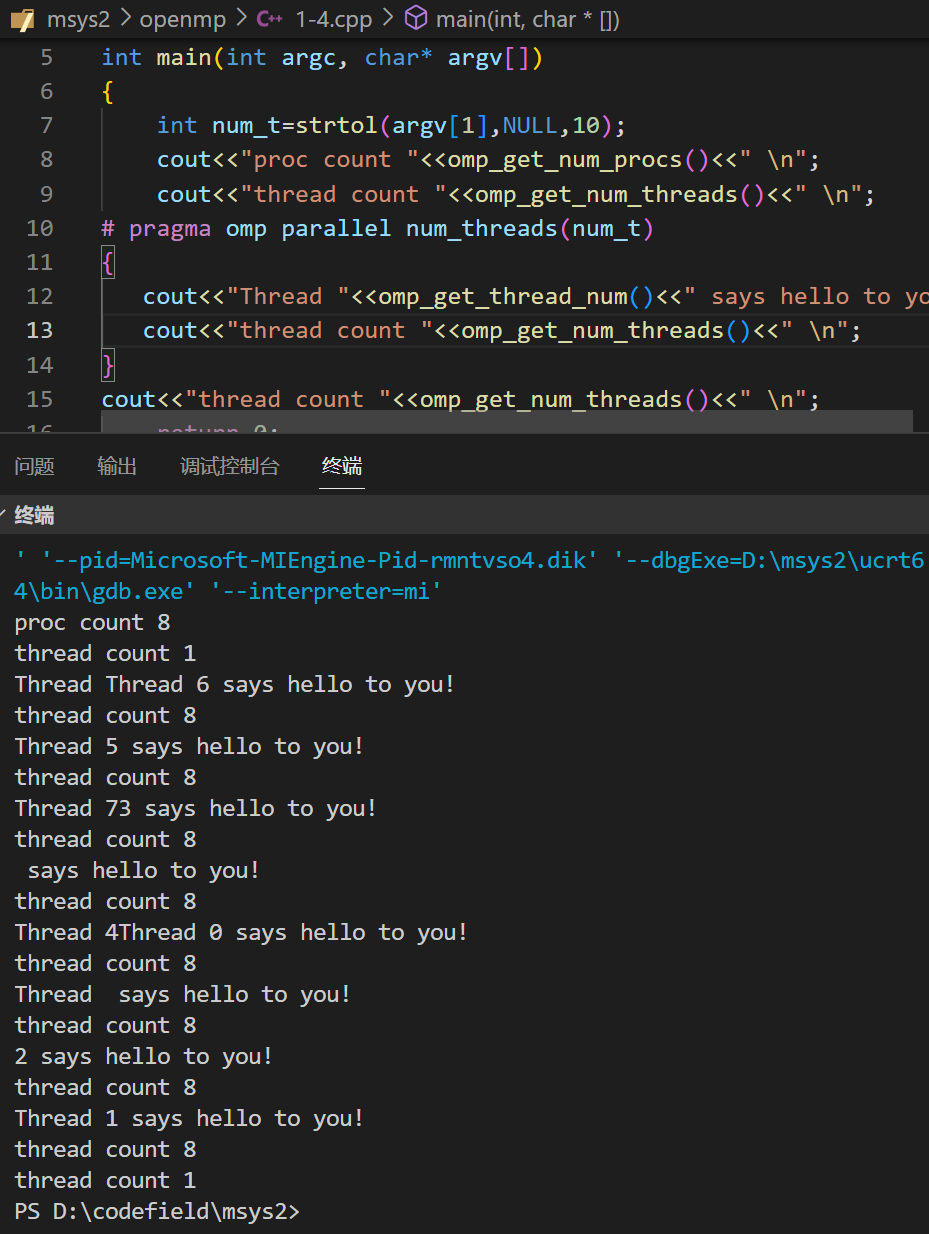
cout<<"thread count "<<omp\_get\_num\_threads()<<" \n";

}

cout<<"thread count "<<omp\_get\_num\_threads()<<" \n";

return 0;

}



1-5

#include<iostream>

#include<cstdlib>

#include<omp.h>

using namespace std;

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

{

int max\_num=strtol(argv[1],NULL,10);

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

int sum=0;

int item=max\_num/num\_t;

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

{

int id=omp\_get\_thread\_num();

int count\_t=omp\_get\_num\_threads();

for(int i=item\*id+1;i<=(id==count\_t-1?max\_num:item\*(id+1));++i)

sum+=i;

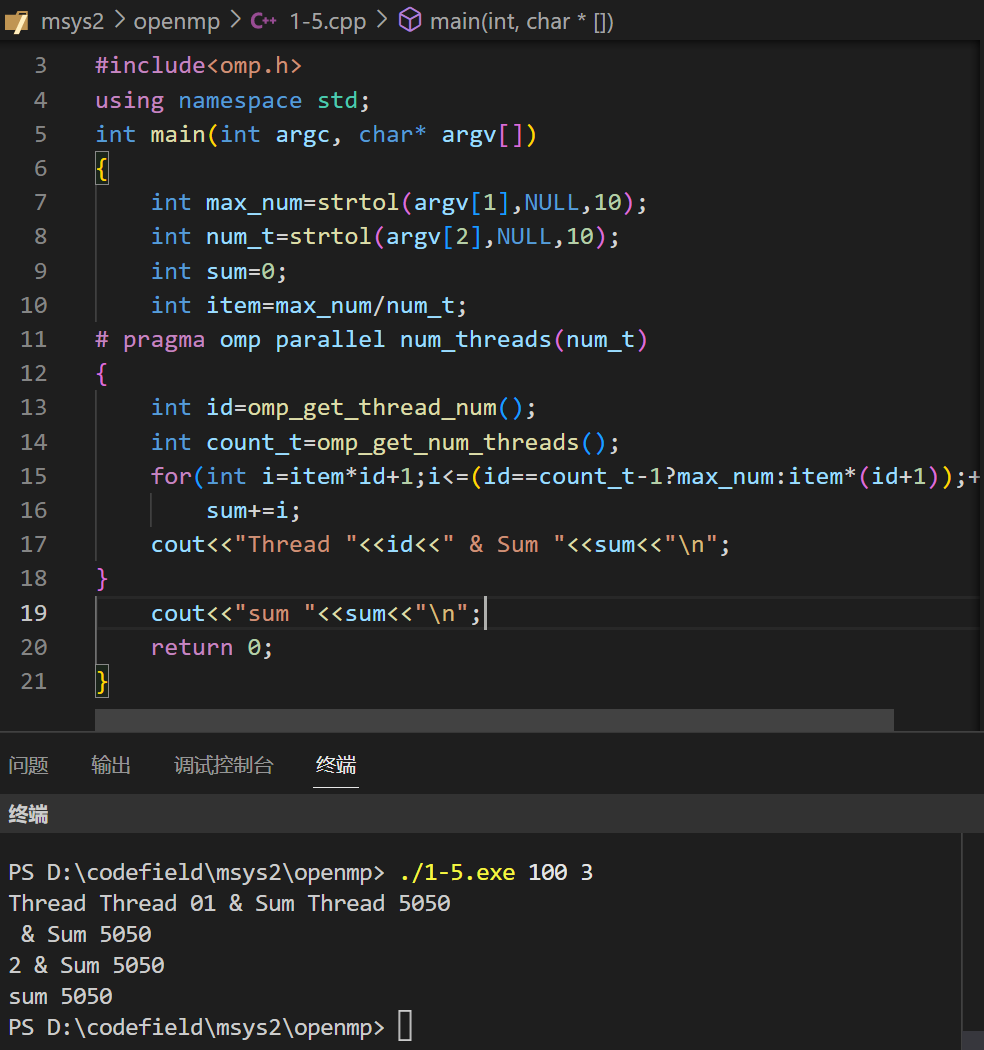
cout<<"Thread "<<id<<" & Sum "<<sum<<"\n";

}

cout<<"sum "<<sum<<"\n";

return 0;

}



1-6

#include<iostream>

#include<cstdlib>

#include<omp.h>

using namespace std;

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

{

int max\_num=strtol(argv[1],NULL,10);

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

int sum=0;

int item=max\_num/num\_t;

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

{

int id=omp\_get\_thread\_num();

int count\_t=omp\_get\_num\_threads();

for(int i=item\*id+1;i<=(id==count\_t-1?max\_num:item\*(id+1));++i)

sum+=i;

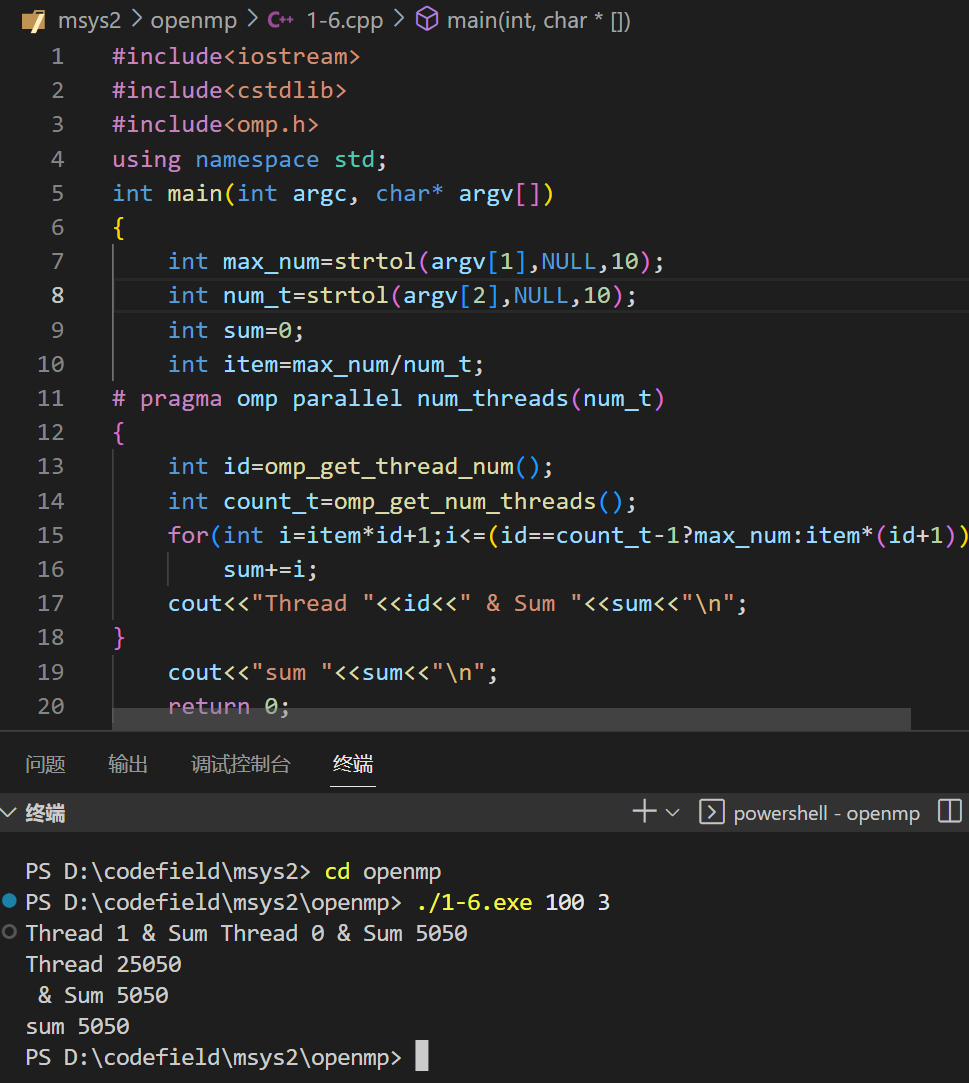
cout<<"Thread "<<id<<" & Sum "<<sum<<"\n";

}

cout<<"sum "<<sum<<"\n";

return 0;

}



1-7

#include<iostream>

#include<cstdlib>

#include<omp.h>

using namespace std;

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

{

int max\_num=strtol(argv[1],NULL,10);

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

int sum=0;

int item=max\_num/num\_t;

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

{

int id=omp\_get\_thread\_num();

int count\_t=omp\_get\_num\_threads();

for(int i=item\*id+1;i<=(id==count\_t-1?max\_num:item\*(id+1));++i)

# pragma omp critical

sum+=i;

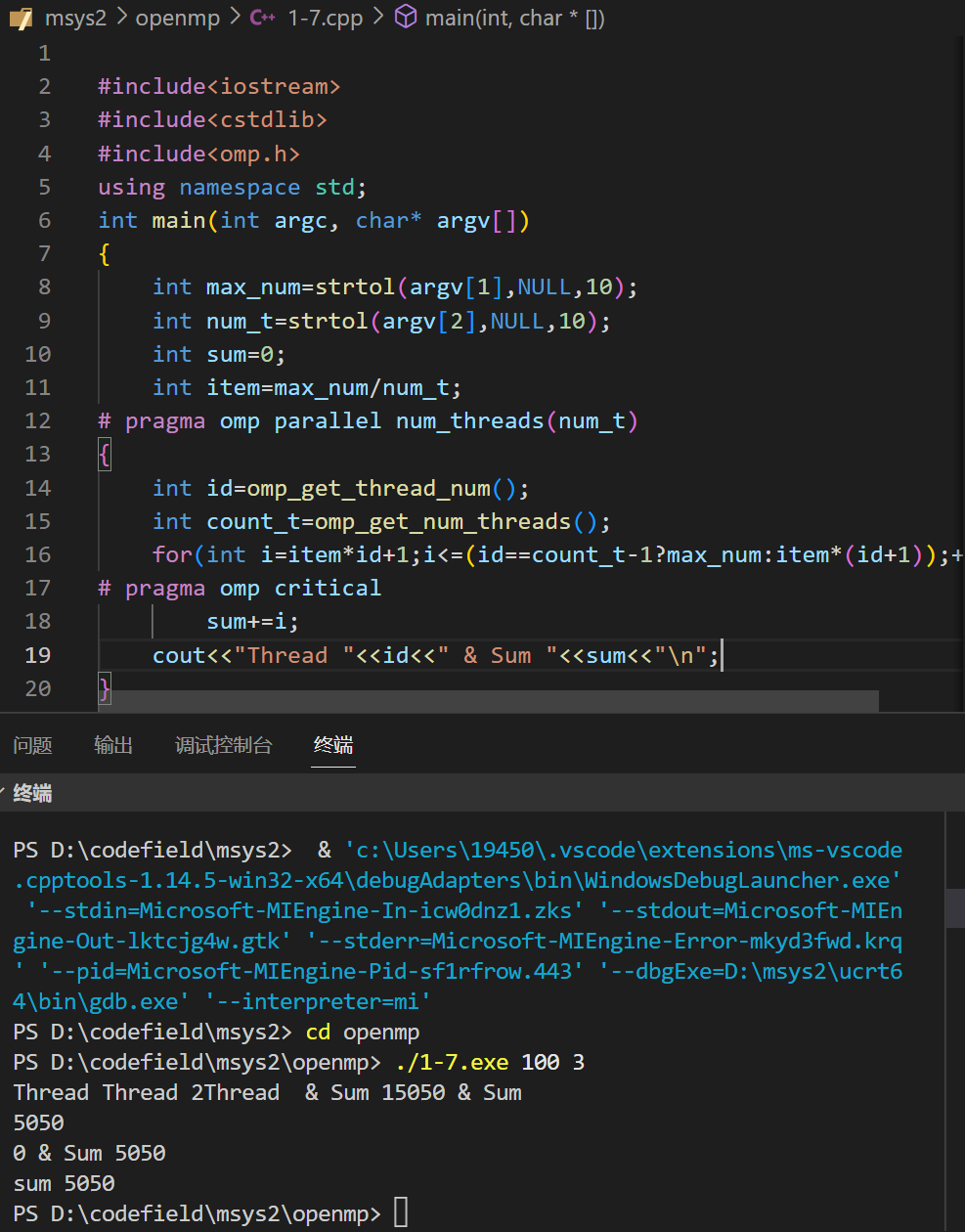
cout<<"Thread "<<id<<" & Sum "<<sum<<"\n";

}

cout<<"sum "<<sum<<"\n";

return 0;

}



1-8

#include<iostream>

#include<cstdlib>

#include<omp.h>

using namespace std;

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

{

int max\_num=strtol(argv[1],NULL,10);

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

int sum=0;

int item=max\_num/num\_t;

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

{

int id=omp\_get\_thread\_num();

int count\_t=omp\_get\_num\_threads();

int localsum=0;

for(int i=item\*id+1;i<=(id==count\_t-1?max\_num:item\*(id+1));++i)

localsum+=i;

# pragma omp critical

sum+=localsum;

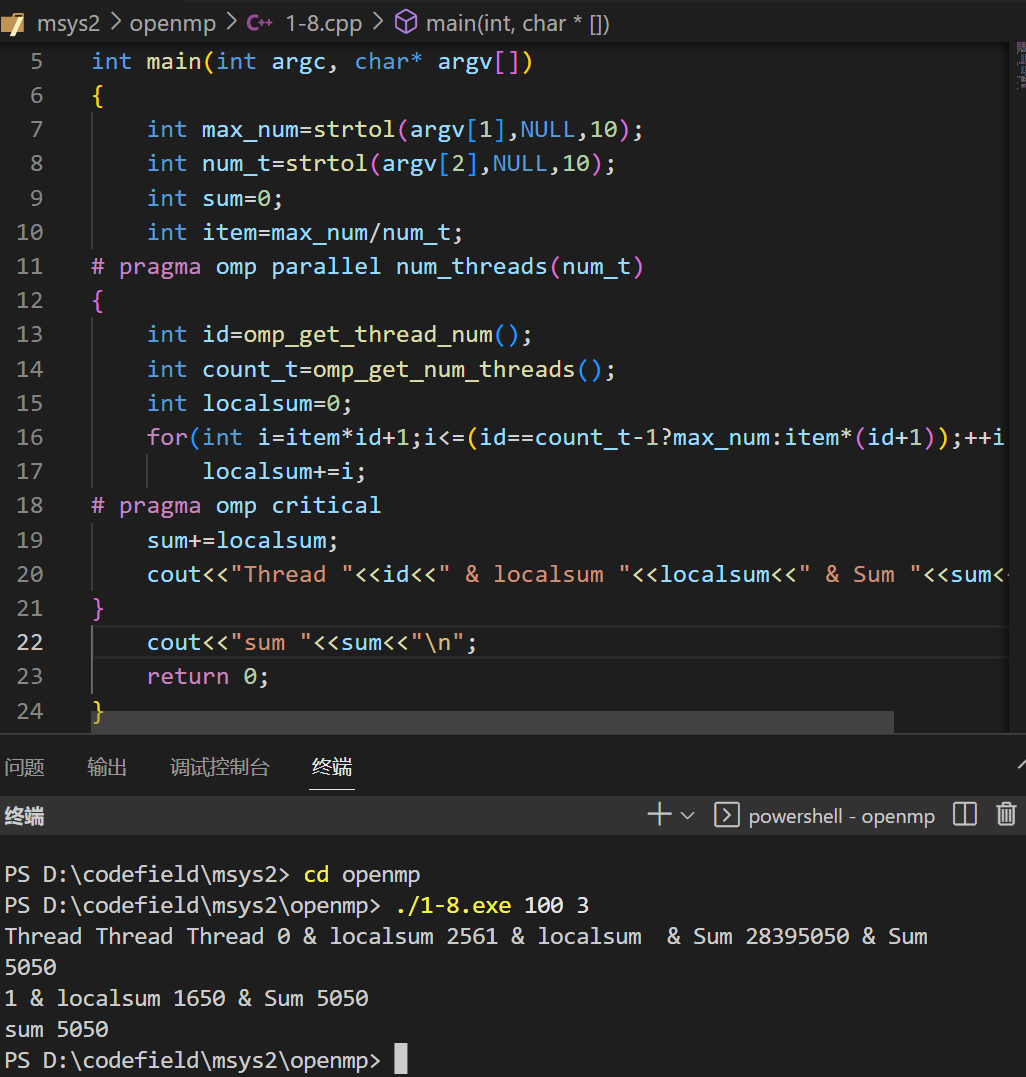
cout<<"Thread "<<id<<" & localsum "<<localsum<<" & Sum "<<sum<<"\n";

}

cout<<"sum "<<sum<<"\n";

return 0;

}



1-9

#include<iostream>

#include<cstdlib>

#include<omp.h>

using namespace std;

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

{

int max\_num=strtol(argv[1],NULL,10);

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

int sum=0;

int item=max\_num/num\_t;

int localsum=0;

# pragma omp parallel num\_threads(num\_t) private(localsum) shared(sum)

{

int id=omp\_get\_thread\_num();

int count\_t=omp\_get\_num\_threads();

for(int i=item\*id+1;i<=(id==count\_t-1?max\_num:item\*(id+1));++i)

localsum+=i;

# pragma omp critical

sum+=localsum;

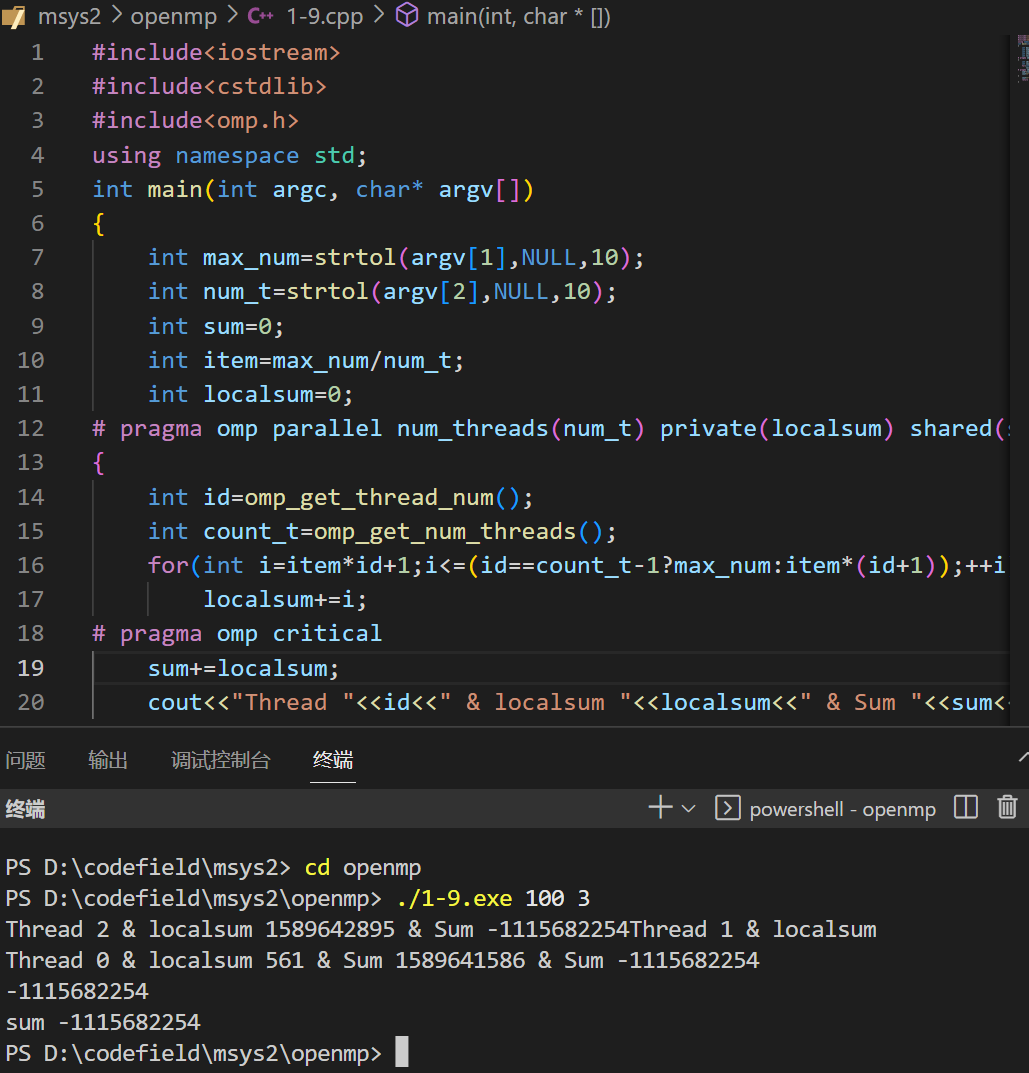
cout<<"Thread "<<id<<" & localsum "<<localsum<<" & Sum "<<sum<<"\n";

}

cout<<"sum "<<sum<<"\n";

return 0;

}



1-10

#include<iostream>

#include<cstdlib>

#include<omp.h>

using namespace std;

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

{

int max\_num=strtol(argv[1],NULL,10);

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

int sum=0;

int item=max\_num/num\_t;

# pragma omp parallel num\_threads(num\_t) reduction(+: sum)

{

int id=omp\_get\_thread\_num();

int count\_t=omp\_get\_num\_threads();

for(int i=item\*id+1;i<=(id==count\_t-1?max\_num:item\*(id+1));++i)

sum+=i;

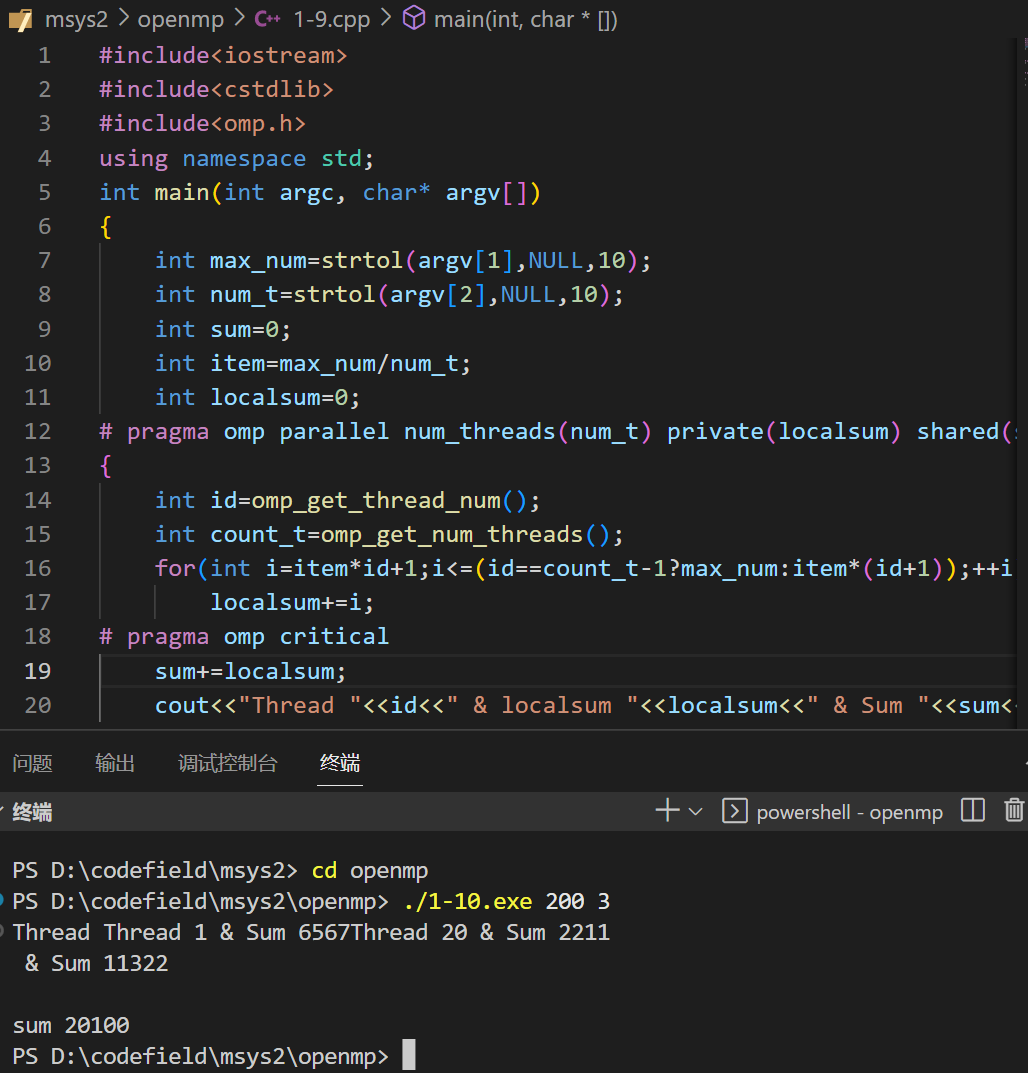
cout<<"Thread "<<id<<" & Sum "<<sum<<"\n";

}

cout<<"sum "<<sum<<"\n";

return 0;

}



1-11

#include<iostream>

#include<cstdlib>

#include<omp.h>

using namespace std;

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

{

int end\_num=strtol(argv[1],NULL,10);

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

int sum=0;

# pragma omp parallel for num\_threads(num\_t) reduction(+: sum)

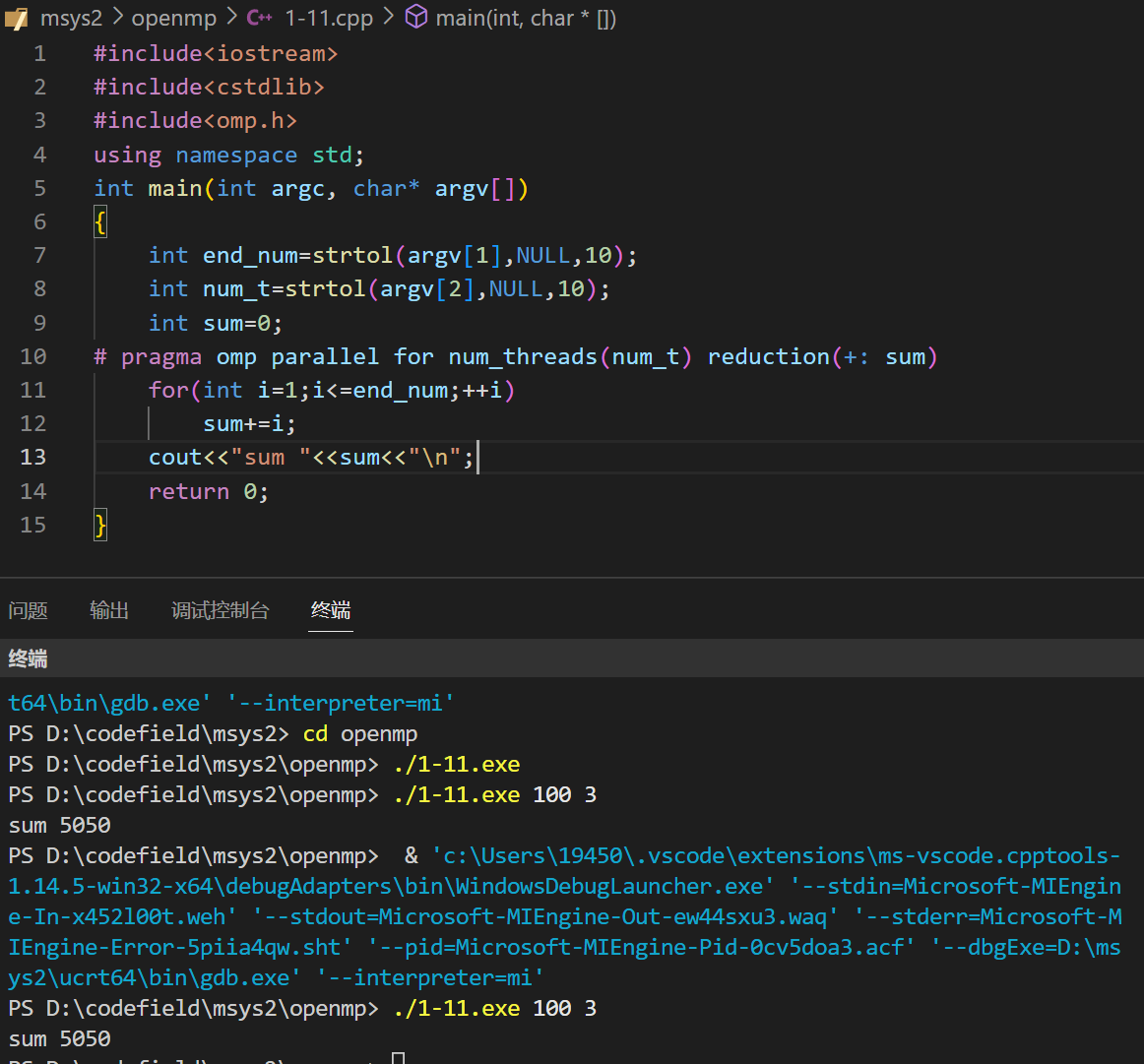
for(int i=1;i<=end\_num;++i)

sum+=i;

cout<<"sum "<<sum<<"\n";

return 0;

}



1. **总结**

通过学习OpenMP并行编程，我深刻体会到并行编程能够显著提高程序运行效率，同时也需要注意线程同步和资源竞争等问题。OpenMP提供了丰富的语句和库函数，使得并行编程变得更加简单和高效。同时，使用parallel for子句可以自动处理数据分配，进一步提高了程序的并行性