**系 统 结 构**

**多核环境下OpenMP并行编程**

1. 实验目的及要求

1) 在Linux平台上编译和运行OpenMP程序；

2) 在Windows平台上编译和运行OpenMP程序。

3) 掌握OpenMP并行编程基础。

1. 实验环境

1) 硬件环境：计算机一台；

2) 软件环境：Linux、Windows、GCC、Visual Studio 2017；

1. 实验内容及步骤

实验一：

1. 配置实验环境——linux环境
   1. 安装gcc
      1. 我使用的Linux环境是亚马逊服务器，ubuntu系统.输入sudo apt-get install gcc.
   2. 将代码文件放入”/home/ubuntu/exp/”，输入”gcc -fopenmp -O2 -o openMp\_Linux.out openMp\_Linux.c”
   3. 设置线程数并添加为环境变量。输入”OMP\_NUM\_THREADS=10”,”export OMP\_NUM\_THREADS”
   4. 运行.out文件。

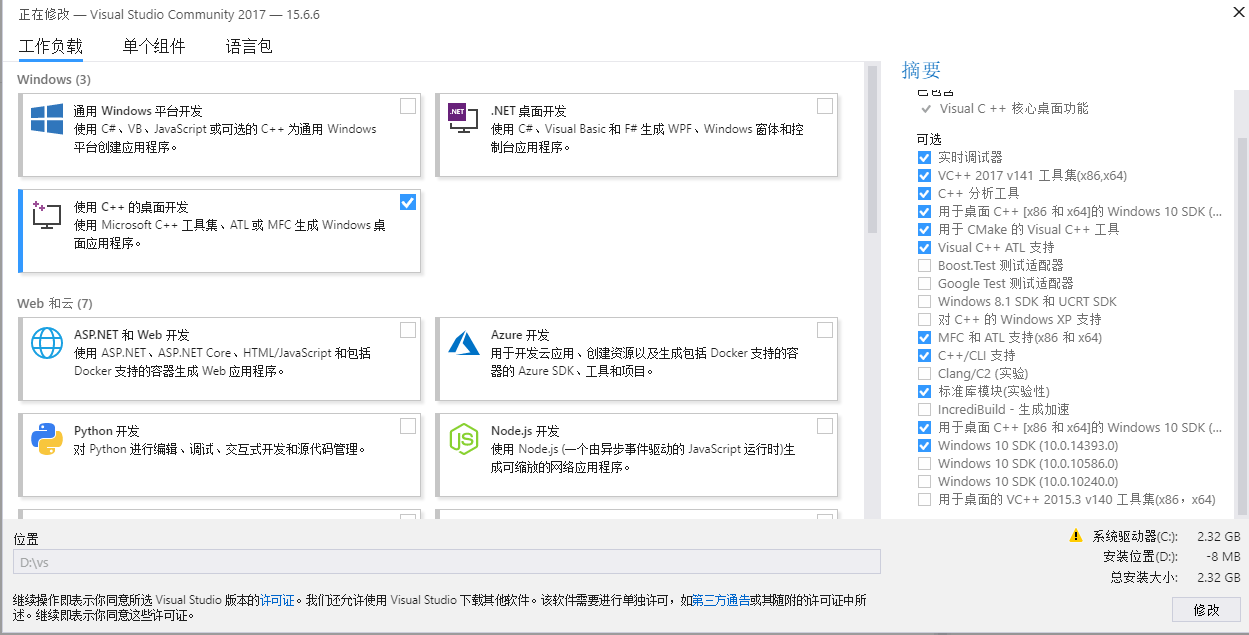
openMp\_Linux.c代码如：

|  |
| --- |
| #include <omp.h>  #include <stdio.h>  int main(){  int nthreads,tid;  #pragma omp parallel private(nthreads,tid)  {  tid=omp\_get\_thread\_num();  printf("Hello World from OMP thread %d\n",tid);  if(tid==0){  nthreads=omp\_get\_num\_threads();  printf("Number of threads is %d\n",nthreads);  }  }  } |

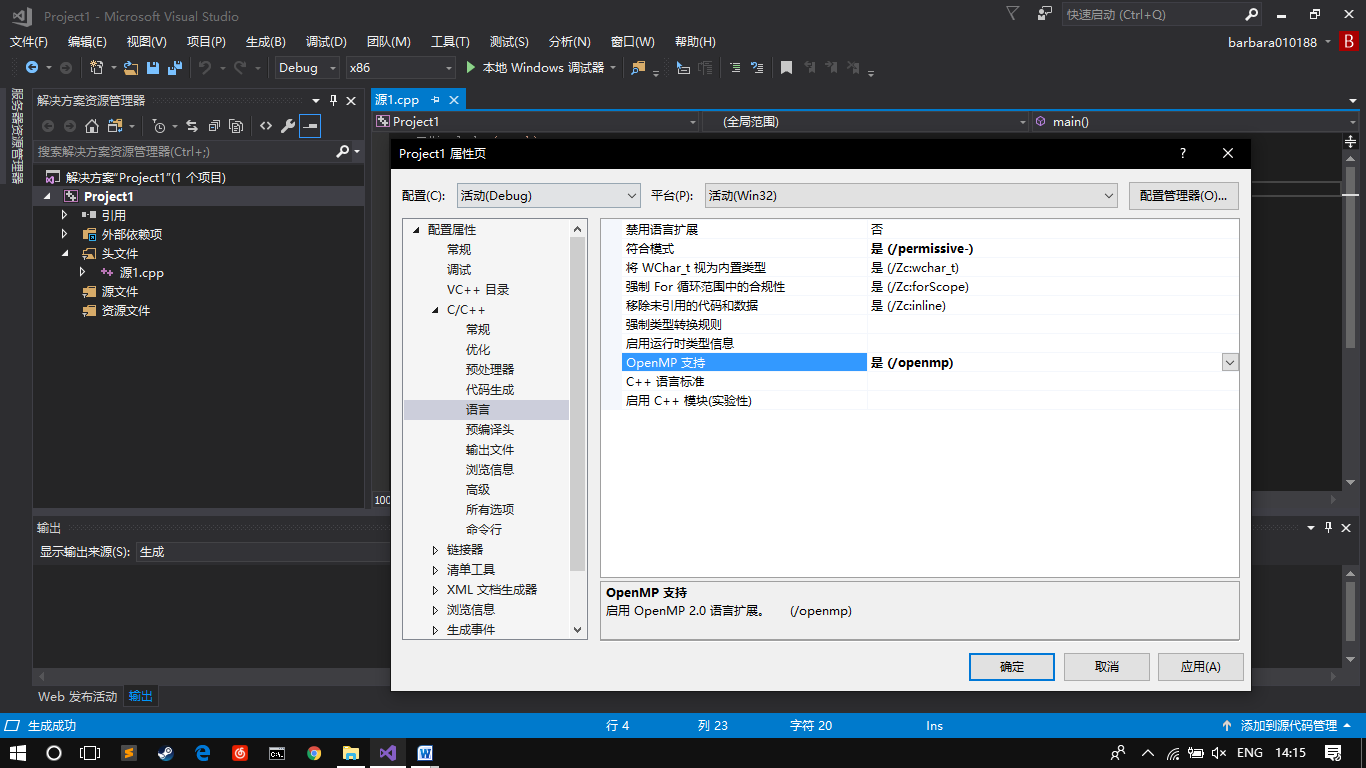
1. 配置实验环境——window环境
   1. 设置环境变量：

我的电脑 -> 属性 -> 高级 -> 环境变量，新建一个OMP\_NUM\_THREADS变量，值设为10，即为程序执行的线程数。

* 1. 启用visual studio installer，在修改里选择C++部分，添加MFChe ATL支持。



* 1. 在visual studio2017里新建visual C++空项目，添加源文件后右击属性，打开OpenMP支持选项。



* 1. openMP\_Windows.c代码如：

|  |
| --- |
| #include <omp.h>  #include <stdio.h>  int main(){  int nthreads,tid;  omp\_set\_num\_threads(8);  #pragma omp parallel private(nthreads,tid)  {  tid=omp\_get\_thread\_num();  printf("Hello World from OMP thread %d\n",tid);  if(tid==0){  nthreads=omp\_get\_num\_threads();  printf("Number of threads is %d\n",nthreads);  }  }  } |

实验二：

1. 配置实验环境——linux环境
   1. 其他步骤如1
   2. 多次配置线程数并添加为环境变量，重新编译后查看运行时间以优化性能。
2. 配置实验环境——windows环境
   1. 其他步骤如2.
   2. 多次更改环境变量，运行后查看运行时间以优化性能。

代码如：

|  |
| --- |
| #include<stdio.h>  #include<omp.h>  #include<time.h>  void comput(float\* A, float\* B, float\* C)//两个矩阵相乘传统方法  {  int x, y;  for (y = 0;y<4;y++)  {  for (x = 0;x<4;x++)  {  C[4 \* y + x] = A[4 \* y + 0] \* B[4 \* 0 + x] + A[4 \* y + 1] \* B[4 \* 1 + x] +  A[4 \* y + 2] \* B[4 \* 2 + x] + A[4 \* y + 3] \* B[4 \* 3 + x];  }  }  }  int main()  {  double duration;  clock\_t s, f;  int x = 0;  int y = 0;  int n = 0;  int k = 0;  float A[] = { 1,2,3,4,  5,6,7,8,  9,10,11,12,  13,14,15,16 };  float B[] = { 0.1f,0.2f,0.3f,0.4f,  0.5f,0.6f,0.7f,0.8f,  0.9f,0.10f,0.11f,0.12f,  0.13f,0.14f,0.15f,0.16f };  float C[16];  s = clock();  //#pragma omp parallel if(false)  for (n = 0;n<1000000;n++)  {  comput(A, B, C);  }  f = clock();  duration = (double)(f - s) / CLOCKS\_PER\_SEC;  printf("s---1,000,000 :%f\n", duration);  for (y = 0;y<4;y++)  {  for (x = 0;x<4;x++)  {  printf("%f,", C[y \* 4 + x]);  }  printf("\n");  }  printf("\n======================\n");  s = clock();  //parallel 2  #pragma omp parallel for  for (n = 0;n<2;n++)////CPU是核线程的  {  for (k = 0;k<500000;k++)//每个线程管个循环  {  comput(A, B, C);  }  }  f = clock();  duration = (double)(f - s) / CLOCKS\_PER\_SEC;  printf("p2- 1,000,000:%f\n", duration);  //parallel 3  s = clock();  #pragma omp parallel for  for (n = 0;n<4;n++) //CPU是核线程的  {  for (k = 0;k<250000;k++)//每个线程管个循环  {  comput(A, B, C);  }  }  f = clock();  duration = (double)(f - s) / CLOCKS\_PER\_SEC;  printf("p3- 1,000,000:%f\n", duration);  //parallel 1  s = clock();  #pragma omp parallel for  for (n = 0;n<1000000;n++)  {  comput(A, B, C);  }  f = clock();  duration = (double)(f - s) / CLOCKS\_PER\_SEC;  printf("p1- 1,000,000 :%f\n", duration);  for (y = 0;y<4;y++)  {  for (x = 0;x<4;x++)  {  printf("%f,", C[y \* 4 + x]);  }  printf("\n");  }  return 0;  } |

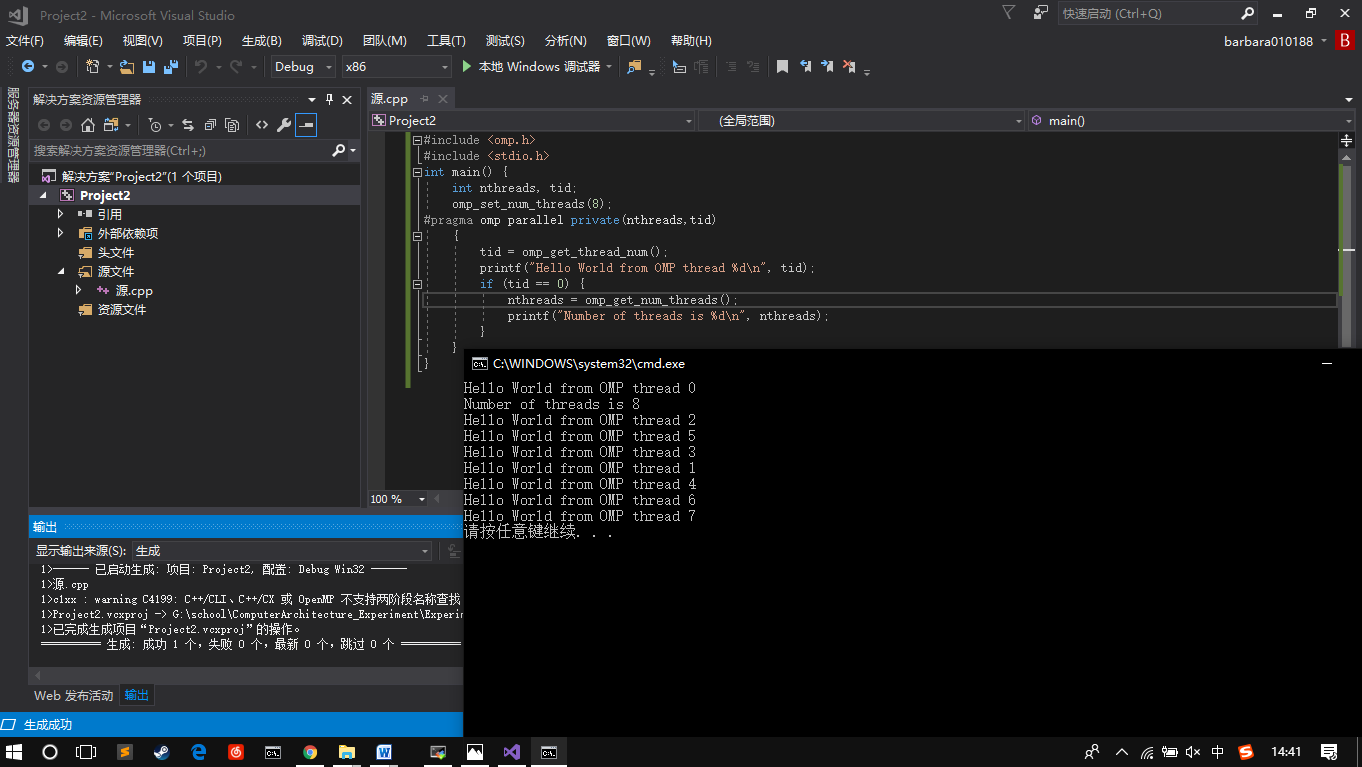
实验三：

1. 求π：windows：
   1. 这是一个项目文件，包括三个方法，分别是传统串行， 并行，优化后的并行。

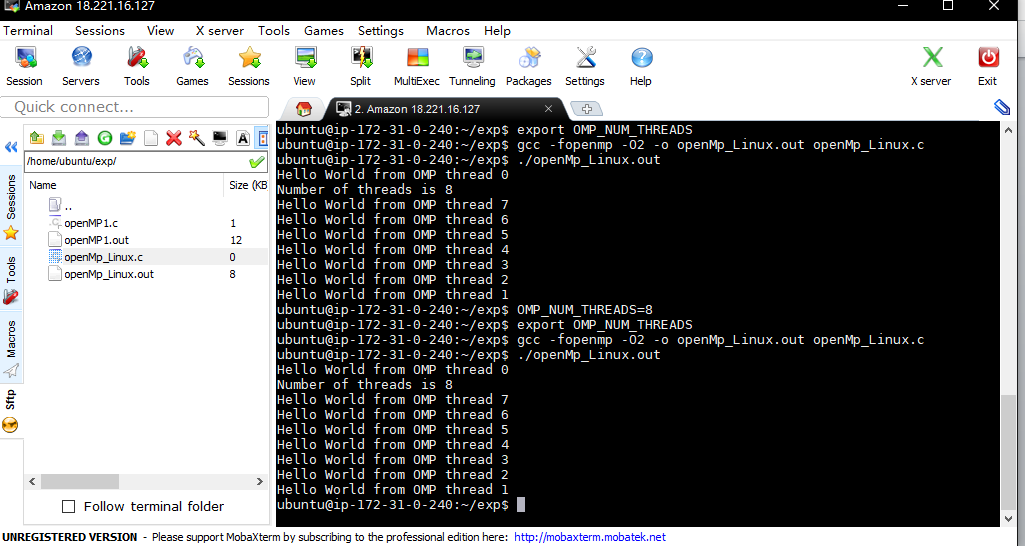
代码如：

|  |
| --- |
| // 求Pi.cpp : 定义控制台应用程序的入口点。  #include "stdafx.h"  #include<omp.h>  #include<iostream>  using namespace std;  void GetPi();  void GetPi\_MP();  void Pi\_MP\_2();  int \_tmain(int argc, \_TCHAR\* argv[])  {  GetPi();  GetPi\_MP();  Pi\_MP\_2();  return 0;  } |
| // File :Pi.cpp  #include "stdafx.h"  #include<omp.h>  #include<iostream>  using namespace std;  //函数fx  double function01(int i,int n)  {  double x=(double)(2\*i+1)/(2\*n);  return 1/(1+x\*x);  }  //计算π  void GetPi()  {  int n;  cout<<"请输入计算次数：\n";  cin>>n;  double total = 0;  double startTime=omp\_get\_wtime();  //#pragma omp parallel for reduction(+:total)  for(int i=0;i<n;i++)  {  total = total + function01(i,n);  }  double endTime=omp\_get\_wtime();  total=total\*4/n;  cout<<"计算结果为："<<total<<"\n";  cout<<"计算所需时间："<<endTime-startTime<<"\n";  } |
| //File: Pi\_MP.cpp  #include "stdafx.h"  #include<omp.h>  #include<iostream>  using namespace std;  //函数fx  double function02(int i,int n)  {  double x=(double)(2\*i+1)/(2\*n);  return 1/(1+x\*x);  }  //计算π  void GetPi\_MP()  {  int i,n;  cout<<"修改后的并行程序！\n请输入计算次数：\n";  cin>>n;  double total = 0,tempTotal;    int cpuCount=omp\_get\_max\_threads(); //获取线程最大数目  int chunkSize =n/cpuCount;  cout << chunkSize;  double startTime=omp\_get\_wtime();  #pragma omp parallel for private(tempTotal)  //#pragma omp parallel for shared(total) private(i,tempTotal) schedule(static,chunkSize)  for(i=0;i<n;i++)  {  tempTotal=function02(i,n);  #pragma omp critical  total = total +tempTotal;  }  double endTime=omp\_get\_wtime();  total=total\*4/n;  cout<<"计算结果为："<<total<<"\n";  cout<<"计算所需时间："<<endTime-startTime<<"\n";  } |
| //File: Pi\_MP\_2.cpp  #include"stdafx.h"  #include<omp.h>  #include<iostream>  using namespace std;  //函数fx  double function03(int start,int end,int n)  {  double total=0,x;  for(int i=start;i<end;i++)  {  x=(double)(2\*i+1)/(2\*n);  total+=1/(1+x\*x);  }  return total;  }  void Pi\_MP\_2()  {  int n;  cout<<"这是修改后的并行程序!\n请输入计算次数：\n";  cin>>n;  double startTime=omp\_get\_wtime();  int cpuCount=omp\_get\_max\_threads(); //获取线程最大数目  double total = 0,tempTotal;  int tempN=n/cpuCount;  #pragma omp parallel for private(tempTotal)  for(int j=0;j<cpuCount;j++)  {  int tempI=j\*tempN;  tempTotal=function03( tempI,tempI+tempN,n);  #pragma omp critical  total +=tempTotal;  }  double endTime=omp\_get\_wtime();  total=total\*4/n;  printf("计算结果为：%E \n",total);  cout<<"计算所需时间："<<endTime-startTime<<"\n";  } |

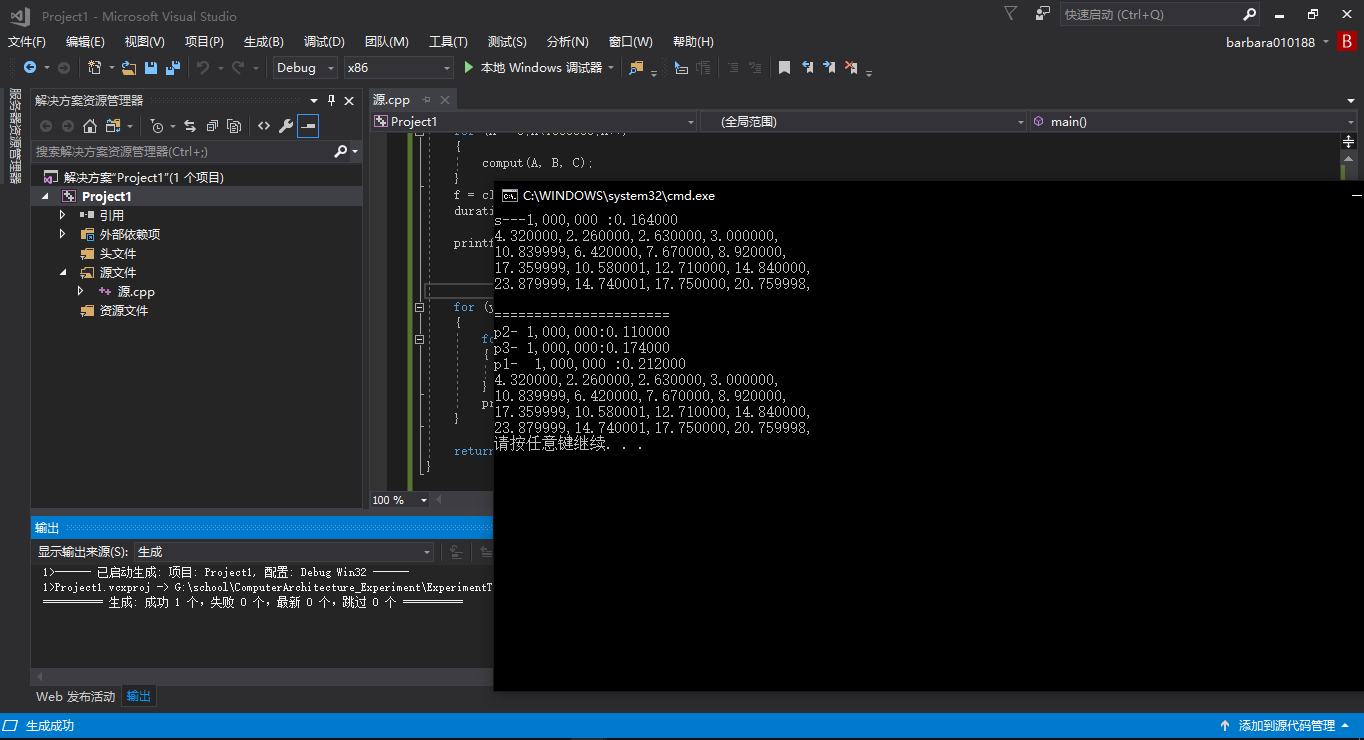
1. 求π：Linux：
   1. 讲四个文件分别放入”/home/ubuntu/exp/”，编译后运行结果。
2. 实验结果及数据处理
   1. 实验一windows：



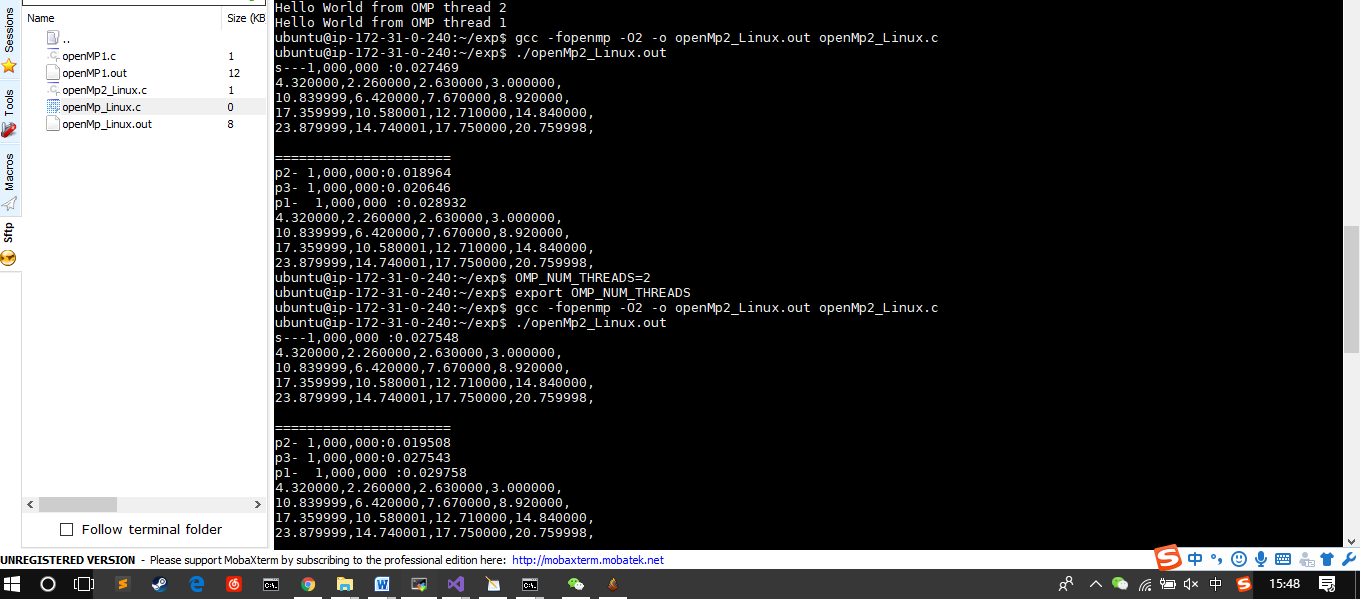
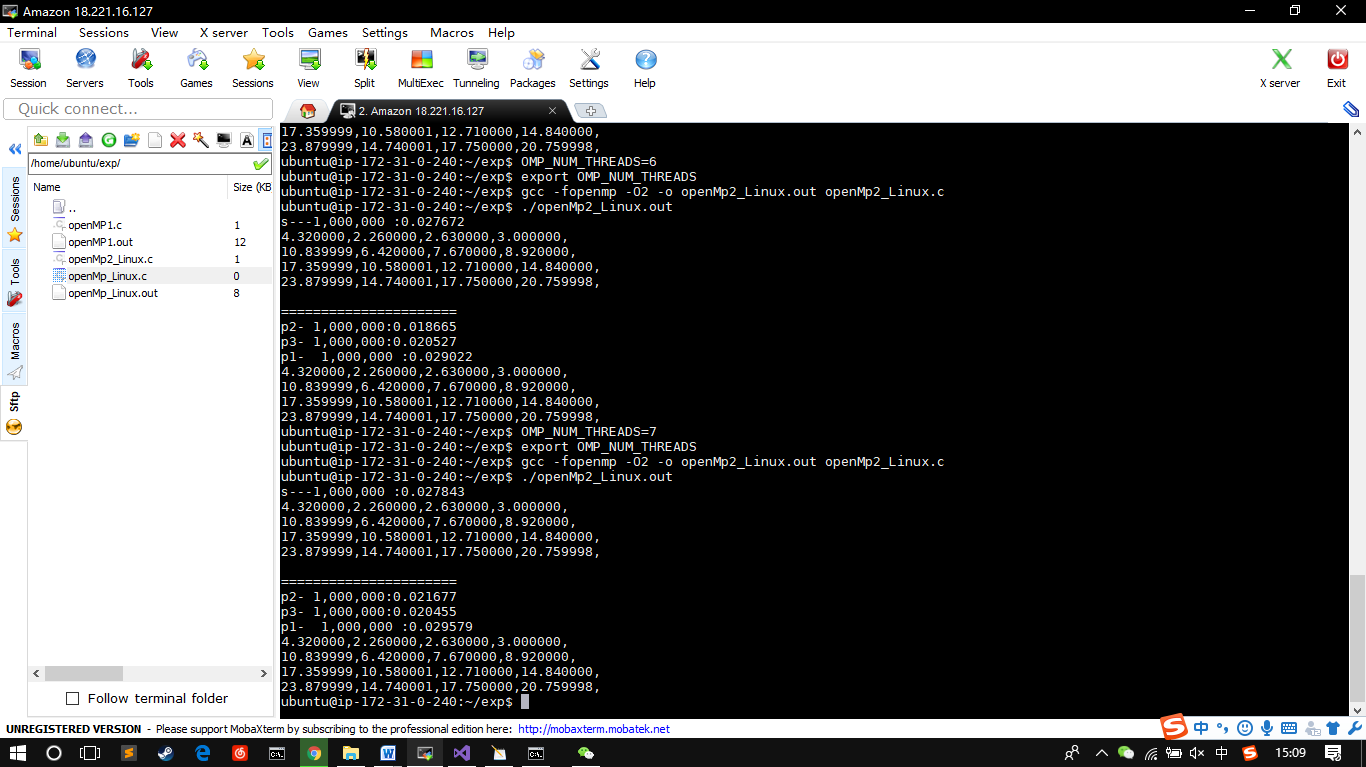
* 1. 实验一Linux:



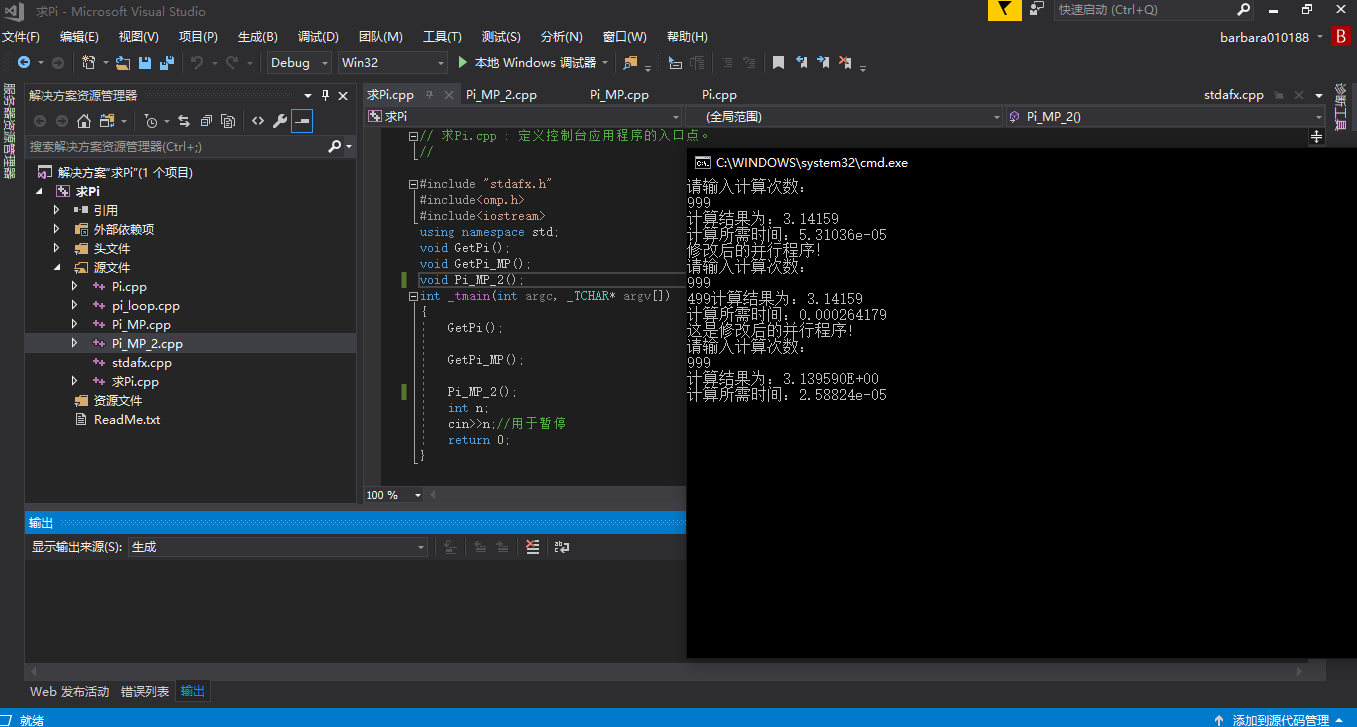
* 1. 实验二windows



* 1. 实验二linux:（实验中分别设置了2,4,6,7,8,10五种线程数，其中线程数为7时运行时间最短，性能最好。）



* 1. 实验三windows:



1. 分析与讨论

通过实验二矩阵计算，我们可以看出：

可以把你的程序运行时间T分成三个部分：并行计算的部分, 假设这部分时间用单个线程时是t1。线程同步的部分，比如说异步锁，假设这部分时间是t2。串行部分，假设这部分时间是t3。使用n个线程的时间可以粗略表达为 T=t1/n+n\*t2+t3.

分析这个公式可以看到t1/n部分随n增加而降低，但是t2会随着n的增加而增加。如果t1比较小，线程同步的地方比较多而导致随着线程数增加t2占据主导，就会出现线程增多，总时间反而变大的情况。

也就是说：在一定范围内：随着节点数的增加，程序的执行时间减少，当节点数到了一定数目后，执行时间反而增加。

通过实验三求π可以对比发现现象：

（运算次数999，线程数8）

|  |  |  |  |
| --- | --- | --- | --- |
|  | 方法一 | 方法二 | 方法三 |
| 时间 | 5.7566e-05 | 0.00029497 | 2.58824e-05 |

方法三（优化并行）时间 < 方法一（传统串行）< 方法二（并行）。即方法三好于方法一好于方法二。那么这里出现两个问题，一，为什么并行后总时间反而增长？二，并行是如何优化以达到最好的效果的？

那么我们先来分析问题一。首先，我们通过分析求π函数的代码可以看出，在这个计算过程中存在依赖的。这里的依赖包括了数据依赖和循环依赖。如果不解决这两个依赖问题首先会导致结果错误。缺省情况下，任何在循环前声明的变量，在线程间都是共享的。为了消除这种循环依赖关系之外，还需要保证每个线程有它自己的factor副本，OpenMP的语句为我们考虑了，通过添加private子句到parallel指令中来实现这一目标。所以当添加 **#pragma omp parallel for private(tempTotal)** 后可解决以上问题。但当使用这种复制副本的方式解决问题后，我发现程序运行时间反而变长了，感觉非常疑惑，所以我去搜索资料，查到了以下论点：

|  |
| --- |
| OpenMP的parallel region结束时，线程之间需要同步：即主线程需要等待所有其他线程完成工作之后才能继续，这个过程可以称做barrier。一个简单的barrier的实现如下\_Atomic int finished = 0;  int num\_threads = N;  void barrier()  {  atomic\_add (&finished, 1);  while (atomic\_read(&finished) < num\_threads);  return;  }  在支持compare-and-swap指令的硬件平台上，atomic\_add可以用cas指令实现。下面这个表格，给出了cas指令的延迟数据：  在上面的barrier代码中，假定threads被不同的CPU core执行，那么一定会出现CAS cache miss。假设只有两个threads，分别被CPU1和CPU2执行。thread1执行atomic\_add时，假设正好满足best-case CAS，即存放finished的cache line正好在CPU1的local cache里。那么在thread2执行atomic\_add时就会发生CAS cache miss，即CPU2要从CPU1的cache中先获得finished的cache line数据，然后再写入自己的结果。根据上面的表格，CAS cache miss需要500个CPU cycles，现代体系结构的CPU每个cycle可以执行不止1条指令，所以500个cycle理想情况下可以执行500条以上的指令。所以，就算只有两个threads，并且threads之间完全没有其他通信，如果parallel region单线程执行时需要不到500个cycles，那么由于barrier的开销，OpenMP多线程就会比单线程还慢。上述barrier只是一个简化了很多的示例，OpenMP一个parallel region还有很多其他额外开销。另外，如果threads之间还要共享其他数据，并且访问共享数据时需要“锁”的保护，一次锁操作也需要至少一条cas指令，这又增加了额外的开销。所以，最好的OpenMP使用场景是线程之间没有很多需要锁保护的共享访问，另外parallel region应该尽可能大，以抵消OpenMP多线程带来的额外同步开销。 |

根据以上资料，据我的思考，我这样写代码反而会大大增大上述T公式里提到的线程同步的部分时间t2，反而会导致效率变低，速度变慢。

所以我们就需要优化该并行方法，根据以上分析，tempTotal部分的操作是导致时间变长的主要原因，所以在优化的代码中，需要把并行部分的for循环任务量细化，“分担”到function03里去，这样就成功解耦，本来的并行除了增大开销外并没有发挥作用，而这样一来，根据内存块的并行，function03不会再阻碍并行，反而会大大减小求tempTotal的总时间，进而减少了整个程序的运行时间。

1. 实验心得

OMP是一种比较简单的让程序获得并行化效果的方法。相对于其他并行操作如：MPI，Intel IPP，Intel TBB，MapReduce，OpenMP实现非常简单，门槛低易上手是它最大的优点。它还可以共享存储模型，而且，它主要面向循环级的并行开发，可以容易地实现增量性的并行化。但是，我个人感觉，虽说OpenMP只要在程序适当位置#pragma就好了，编译器以及动态库会帮我并行处理。然而风险在于如果#pragma没有写好，很容易产生数据竞跑，而且编译器根本不会提示，程序还是照样能跑，只不过跑出来的结果可能会错很多。而且，哪怕很不容易的跑通了，答案也是正确的，也很容易发现整个项目效率反而变低了，还需要进一步的优化。而这种优化对程序员的能力是有要求的，需要分析代码内部，整个调试和优化过程还是很复杂的。