并行与分布式计算作业

第5次作业

姓名：郝裕玮

班级：计科1班

学号：18329015

**对于问题1：**

一、问题描述

考虑一个简单的循环，它调用一个包含可编程延迟（睡眠）的函数 dummy。该函数的所有调用都独立于其他函数。使用static、dynamic和guided调度将这个循环划分为4个线程。对static和guided调度使用不同的参数。随着虚拟函数内的延迟变大，记录此实验的结果。

二、解决方案

dummy函数和其他C语言函数相同，有函数的返回值类型和形参定义，也有函数体，只是函数体内部没有任何执行语句。实际也称为空函数。

代码如下所示（具体思路和详细分析均已包含在代码注释中）：

#include <iostream>

#include <cstdlib>

#include <time.h>

#include <sys/time.h> //用于计时

#include <cstdio>

#include <cstring>

#include <windows.h> //调用dummy中的Sleep函数

#include <omp.h> //使用openmp时必须包含该库

using namespace std;

#define GET\_TIME(now) { \

   struct timeval t; \

   gettimeofday(&t, NULL); \

   now = t.tv\_sec + t.tv\_usec/1000000.0; \

}

//最后一行将us转换为s，统一单位

void dummy();//题目要求的延迟睡眠函数dummy

void Static();//调度方式为static

void Dynamic();//调度方式为dynamic

void Guided();//调度方式为guided

int main(){

    double start,end;//用于记录开始和结束的时间

    GET\_TIME(start);

    Static();

    GET\_TIME(end);

    cout<<"static调度方式的用时为："<<end-start<<endl;

    GET\_TIME(start);

    Dynamic();

    GET\_TIME(end);

    cout<<"dynamic调度方式的用时为："<<end-start<<endl;

    GET\_TIME(start);

    Guided();

    GET\_TIME(end);

    cout<<"guided调度方式的用时为："<<end-start<<endl;

}

void dummy(){

    Sleep(100);//睡眠/延迟时间可修改,单位为ms

}

void Static(){

#pragma omp parallel for num\_threads(4) schedule(static)

    for(int i=1;i<=100;i++){

        dummy();

    }

}

void Dynamic(){

#pragma omp parallel for num\_threads(4) schedule(dynamic)

    for(int i=1;i<=100;i++){

        dummy();

    }

}

void Guided(){

#pragma omp parallel for num\_threads(4) schedule(guided)

    for(int i=1;i<=100;i++){

        dummy();

    }

}

三、实验结果

程序编译命令为（包含-fopenmp参数即可）：

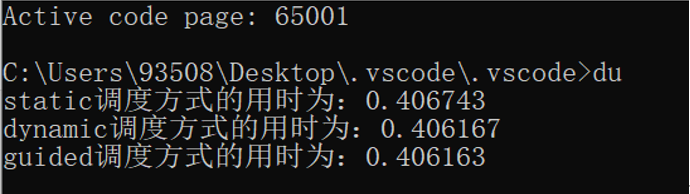
g++ dummy\_For\_1.cpp -fopenmp -o du

由于打印信息包含中文（且文件是utf-8编码），所以需要在cmd窗口下输入:

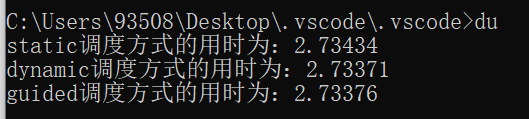
chcp 65001

以此来保证中文正常显示。

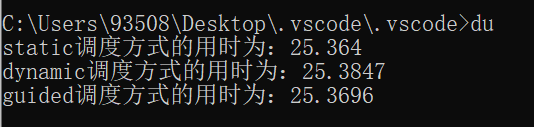
（1）dummy函数内部为Sleep(1)时：



（2）dummy函数内部为Sleep(100)时：



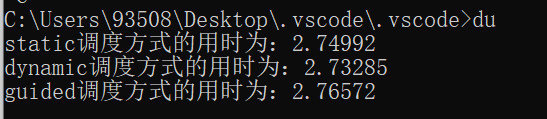
（3）dummy函数内部为Sleep(1000)时：



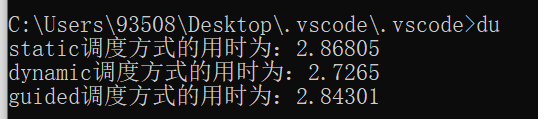
（4）对于static和guided的不同参数（以Sleep(100)为标准）：

对于schedule(static,size)和schedule(guided,size)中的size参数：使用size参数时，将分配给每个线程的size次连续的迭代计算。

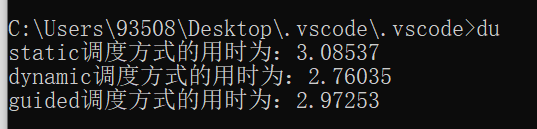
① size=1



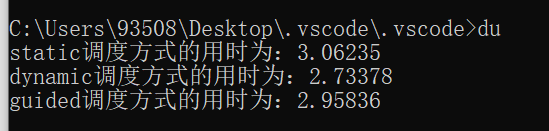
② size=2



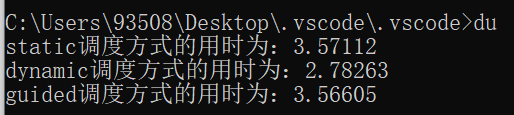
③ size=4



④ size=8



④ size=16



综上可知，当dummy函数内部延迟一定时，static和guided的size参数越大，运行时间越长（size=8相较于size=4略有降低，但size=16相较于size=8仍然是运行时间增加）

四、遇到的问题及解决方法

该题未遇到难以解决的问题，实验过程较为顺利。

**对于问题2：**

一、问题描述

使用section在OpenMP中实现生产者-消费者框架以创建单个生产者任务和单个消费者任务。使用锁来确保适当的同步。同时用不同数量的生产者和消费者来测试您的程序。

二、解决方案

（1）对于单个生产者和单个消费者，代码如下所示（具体思路和详细分析均已包含在代码注释中）：

#include <iostream>

#include <queue>

#include <math.h>

#include <omp.h>

#include <stdio.h>

#include <time.h>

#include <semaphore.h> //为使用信号量需要导入的库

using namespace std;

sem\_t num;//信号量num代表当前队列里有无资源，防止队列无资源时被消费者占用线程

sem\_t t;//信号量t代表当前线程是否被生产者或者消费者占用，防止二者发生冲突，类似于读写锁

int cnt;//用于记录当前队列内资源数

queue<int> resourse;//资源队列

//生产者

void produce(){

    while(true){

        int element;//加入队列的元素

        for(int i=1;i<=100000;i++){

            int j=1;

            j=2;

        }//用于延长函数执行时间，防止线程一直被生产者或消费者占用

        //生产消费交替执行更能体现程序正确性

        if(resourse.empty()){

            element=1;//若当前队列为空，则加入值为1的元素

        }

        else{

            element=resourse.back()+1;//不为空则加入值为 队尾+1 的元素

        }

        sem\_wait(&t);//等待信号量t释放再执行接下来的内容

        cnt++;//队列资源数量+1

        resourse.push(element);//将当前元素加入队列

        cout<<"No."<<element<<"已生产！当前队列资源数为："<<cnt<<endl;//打印信息

        sem\_post(&t);//释放信号量t,供生产者和消费者占用

        sem\_post(&num);//释放信号量num,告诉消费者此时资源队列不再为空

    }

}

//消费者

void consume(){

    while(true){

        int element;//加入队列的元素

        for(int i=0;i<700000;i++){

            int j=1;

            j=2;

        }//用于延长函数执行时间，防止线程一直被生产者或消费者占用

        //生产消费交替执行更能体现程序正确性

        sem\_wait(&num);//等待信号量num释放再执行接下来的内容(资源队列不为空才可以执行消费者函数)

        sem\_wait(&t);//等待信号量t释放再执行接下来的内容

        element=resourse.front();//消费元素为当前队列的队头

        resourse.pop();//弹出队头元素

        cnt--;//队列资源数量-1

        cout<<"No."<<element<<"已消费！当前队列资源数为："<<cnt<<endl;//打印信息

        sem\_post(&t);//释放信号量t,供生产者和消费者占用

    }

}

int main(){

    //初始化信号量

    sem\_init(&num,0,0);

    sem\_init(&t,0,1);

    #pragma omp parallel sections//使用section框架

    {

        #pragma omp section

        {

            produce();

        }

        #pragma omp section

        {

            consume();

        }

    }

}

（2）对于多个生产者和多个消费者，代码如下所示（具体思路和详细分析均已包含在代码注释中）：

#include <iostream>

#include <queue>

#include <math.h>

#include <omp.h>

#include <stdio.h>

#include <time.h>

#include <semaphore.h> //为使用信号量需要导入的库

using namespace std;

sem\_t num;//信号量num代表当前队列里有无资源，防止队列无资源时被消费者占用线程

sem\_t t;//信号量t代表当前线程是否被生产者或者消费者占用，防止二者发生冲突，类似于读写锁

int cnt;//用于记录当前队列内资源数

queue<int> resourse;//资源队列

//生产者

void produce(int id){

    while(true){

        int element;//加入队列的元素

        for(int i=1;i<=100000;i++){

            int j=1;

            j=2;

        }//用于延长函数执行时间，防止线程一直被生产者或消费者占用

        //生产消费交替执行更能体现程序正确性

        if(resourse.empty()){

            element=1;//若当前队列为空，则加入值为1的元素

        }

        else{

            element=resourse.back()+1;//不为空则加入值为 队尾+1 的元素

        }

        sem\_wait(&t);//等待信号量t释放再执行接下来的内容

        cnt++;//队列资源数量+1

        resourse.push(element);//将当前元素加入队列

        cout<<"No."<<element<<"已生产！线程号为："<<id<<",当前队列资源数为："<<cnt<<endl;//打印信息

        sem\_post(&t);//释放信号量t,供生产者和消费者占用

        sem\_post(&num);//释放信号量num,告诉消费者此时资源队列不再为空

    }

}

//消费者

void consume(int id){

    while(true){

        int element;//加入队列的元素

        for(int i=0;i<700000;i++){

            int j=1;

            j=2;

        }//用于延长函数执行时间，防止线程一直被生产者或消费者占用

        //生产消费交替执行更能体现程序正确性

        sem\_wait(&num);//等待信号量num释放再执行接下来的内容(资源队列不为空才可以执行消费者函数)

        sem\_wait(&t);//等待信号量t释放再执行接下来的内容

        element=resourse.front();//消费元素为当前队列的队头

        resourse.pop();//弹出队头元素

        cnt--;//队列资源数量-1

        cout<<"No."<<element<<"已消费！线程号为："<<id<<",当前队列资源数为："<<cnt<<endl;//打印信息

        sem\_post(&t);//释放信号量t,供生产者和消费者占用

    }

}

int main(){

    //设置生产者和消费者占用的线程数量(也即生产者和消费者的数量)

    int produce\_threads=8;//这里可修改，该线程数量代表生产者数量

    int consume\_threads=4;//这里可修改，该线程数量代表消费者数量

    //初始化信号量

    sem\_init(&num,0,0);

    sem\_init(&t,0,1);

    int total\_threads=produce\_threads+consume\_threads;//线程总数量，即生产者+消费者总数量

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

    {

        int id=omp\_get\_thread\_num();//获取线程id

        #pragma omp parallel sections//使用section框架

        {

            #pragma omp section

            {

                if(id<produce\_threads){//前produce\_threads个线程均用于生产者

                    produce(id);

                }

            }

            #pragma omp section

            {

                if(id>=produce\_threads && id<=total\_threads-1){

                //剩下的线程均用于消费者

                    consume(id);

                }

            }

        }

    }

}

三、实验结果

2个程序编译命令分别为（包含-fopenmp参数即可）：

g++ 2.1.cpp -fopenmp -o 21

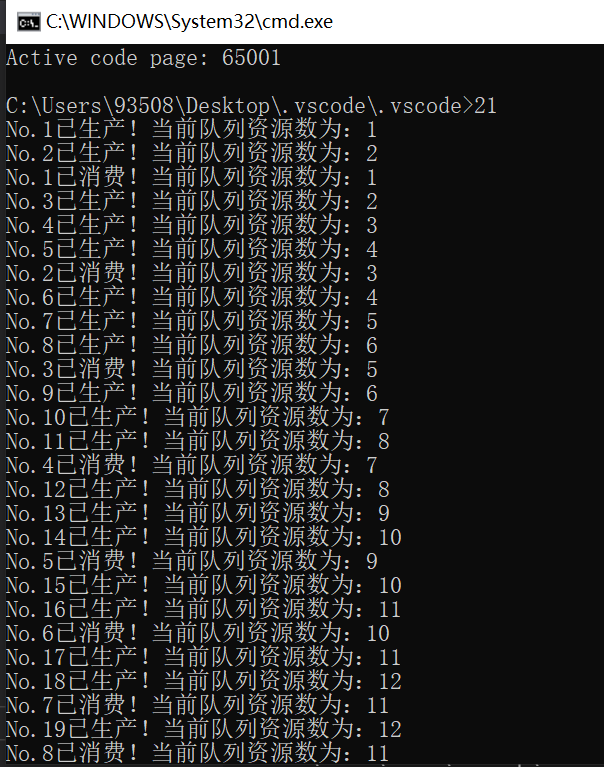
g++ 2.2.cpp -fopenmp -o 22

由于打印信息包含中文（且文件是utf-8编码），所以需要在cmd窗口下输入:

chcp 65001

以此来保证中文正常显示。

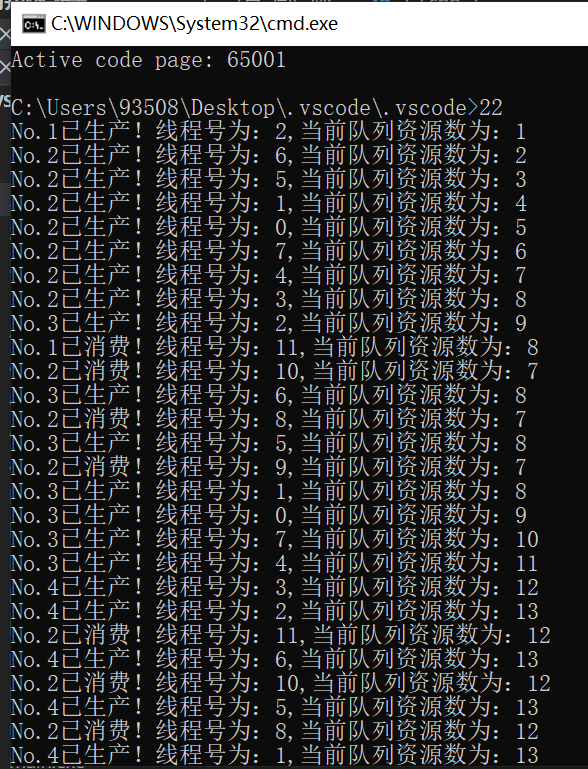
（1）对于单个生产者和单个消费者：



由上图可见，队列资源数和生产者与消费者的调用关系始终正确（生产使得资源+1，消费使得资源-1）。

（2）对于多个生产者和多个消费者：

以8个生产者，4个消费者为例：



由上图可见，队列资源数和生产者与消费者的调用关系始终正确（生产使得资源+1，消费使得资源-1），且前8个线程（0-7号线程）均用于生产者，后4个线程（8-11号线程）均用于消费者。

四、遇到的问题及解决方法

一开始我没有在生产者和消费者函数中添加延长运行时间的代码部分：

        /\*for(int i=0;i<700000;i++){

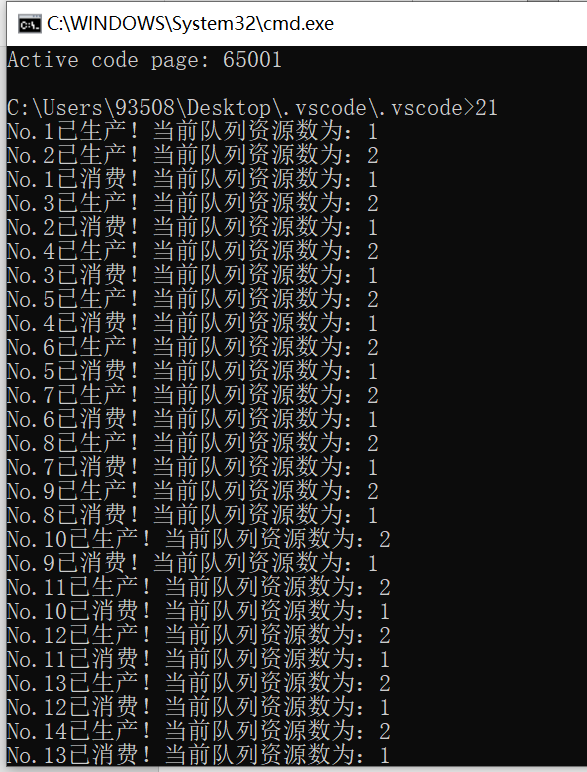
            int j=1;

            j=2;

        }//用于延长函数执行时间，防止线程一直被生产者或消费者占用

        //生产消费交替执行更能体现程序正确性\*/

导致队列资源数量始终在0-2之间徘徊，线程竞争不明显：



**对于问题3：**

一、问题描述

考虑以压缩行格式存储的稀疏矩阵（您可以在网络上找到此格式的描述或有关稀疏线性代数的任何合适的文本）。 编写一个 OpenMP 程序来计算这个矩阵与向量的乘积。从Matrix Market (http://math.nist.gov/MatrixMarket/)下载示例矩阵，并根据矩阵大小和线程数测试您的实现性能。

二、解决方案

具体代码如下所示（具体分析和详细思路均已包含在代码注释中）：

#include <iostream>

#include <fstream> //用于文件读取

#include <cstdlib>

#include <sys/time.h> //用于计时

#include <cstdio>

#include <cstring>

#include <omp.h> //使用openmp时必须包含该库

using namespace std;

#define GET\_TIME(now) { \

   struct timeval t; \

   gettimeofday(&t, NULL); \

   now = t.tv\_sec + t.tv\_usec/1000000.0; \

}

//最后一行将us转换为s，统一单位

//OpenMP并行函数

void Parallel(int thread\_count,int row,int \*col\_pointer,int \*row\_pointer,double \*value,double \*vec,double \*ans);

//串行函数

void Serial(int thread\_count,int row,int \*col\_pointer,int \*row\_pointer,double \*value,double \*vec,double \*ans);

int main() {

    int thread\_count=16;//这里可修改线程数

    ifstream fin;

    fin.open("C:\\Users\\93508\\Desktop\\psmigr\_3.rua");//读取文件

    string text;//用于存储无关信息

    for(int i=1;i<=2;i++){

        getline(fin,text);//读取文件的前2行无关信息

    }

    int row,col,num;//row:矩阵行数;col:矩阵列数;num:矩阵非零元素个数

    fin>>text>>row>>col>>num>>text;//第3行中间3个数分别为row,col,num

    //打印矩阵信息

    cout<<endl;

    cout<<"矩阵规模："<<row<<"\*"<<col<<endl;

    cout<<"矩阵非零元素个数："<<num<<endl;

    for(int i=1;i<=2;i++){

        getline(fin,text);//读取第4行无关信息(需要读2次,因为第3行结尾有个回车/终止符,只读1次会导致读取数据时读入字符串)

    }

    int \*row\_pointer=(int\*)malloc(sizeof(int)\*(row+1));

    //row\_pointer数组:保存矩阵每行第1个非零元素在value中的索引,大小为row+1

    for(int i=0;i<=row;i++){

        fin>>row\_pointer[i];//根据空格依次读入每个值

        row\_pointer[i]--;//文件中的矩阵从(1,1)开始,所以需要对所有值-1

    }

    int \*col\_pointer=(int\*)malloc(sizeof(int)\*num);

    //col\_pointer数组:保存value数组中每个元素的列索引,大小为num

    for(int i=0;i<=num-1;i++){

        fin>>col\_pointer[i];//根据空格依次读入每个值

        col\_pointer[i]--;//文件中的矩阵从(1,1)开始,所以需要对所有值-1

    }

    double \*value=(double\*)malloc(sizeof(double)\*num);

    //value数组:按顺序保存矩阵所有的非零元素(按从上往下，从左往右的行遍历方式访问元素)

    for(int i=0;i<=num-1;i++){

        fin>>value[i];//根据空格依次读入每个值

    }

    double \*vec=(double\*)malloc(sizeof(double)\*col);

    //vec数组:记录用于和矩阵相乘的列向量,矩阵规模为col\*1

    for(int i=0;i<=row-1;i++){

        vec[i]=2;//为了计算方便,所有元素的值均设为2

    }

    double \*ans=(double\*)malloc(sizeof(double)\*row);

    //ans数组:用于存储矩阵和向量的相乘结果

    memset(ans,0,sizeof(double)\*row);//结果数组的元素全部初始化为0

    double start1,end1,start2,end2;

    int t=1000;//为了放大时间便于比较,设置串行和并行均循环运算t次(次数t视矩阵规模大小而定)

    cout<<"线程数："<<thread\_count<<endl;

    cout<<"循环次数："<<t<<"次"<<endl;

    GET\_TIME(start1);//得到串行计算运行的开始时间

    while(t--){

        Serial(thread\_count,row,col\_pointer,row\_pointer,value,vec,ans);//调用串行函数

    }

    GET\_TIME(end1);//得到串行计算运行的结束时间

    cout<<"串行时间："<<end1-start1<<endl;//输出串行时间

    t=1000;//将t重置

    GET\_TIME(start2);//得到并行计算运行的开始时间

    while(t--){

        Parallel(thread\_count,row,col\_pointer,row\_pointer,value,vec,ans);//调用并行函数

    }

    GET\_TIME(end2);//得到并行计算运行的结束时间

    cout<<"并行时间："<<end2-start2<<endl;//输出并行时间

    cout<<"加速比: "<<(end1-start1)/(end2-start2)<<endl<<endl;//输出加速比

    fin.close();//关闭文件

    system("pause");

    return 0;

}

void Parallel(int thread\_count,int row,int \*col\_pointer,int \*row\_pointer,double \*value,double \*vec,double \*ans)

{

#pragma omp parallel for num\_threads(thread\_count)//由于内层循环不可拆分(含累加)且外层各循环之间互不影响

//所以可将外层循环用omp进行并行化

    for(int i=0;i<=row-1;i++){

        for(int j=row\_pointer[i];j<=row\_pointer[i+1]-1;j++){//外层循环遍历次数代表第i行的非零元素个数

            int pos=col\_pointer[j];//定位第i行非零元素的列坐标

            ans[i]+=value[j]\*vec[pos];//将该元素与列向量的对应位置相乘，并累加到结果向量的对应位置上

        }

    }

}

void Serial(int thread\_count,int row,int \*col\_pointer,int \*row\_pointer,double \*value,double \*vec,double \*ans)

{

    for(int i=0;i<=row-1;i++){

        for(int j=row\_pointer[i];j<=row\_pointer[i+1]-1;j++){//外层循环遍历次数代表第i行的非零元素个数

            int pos=col\_pointer[j];//定位第i行非零元素的列坐标

            ans[i]+=value[j]\*vec[pos];//将该元素与列向量的对应位置相乘，并累加到结果向量的对应位置上

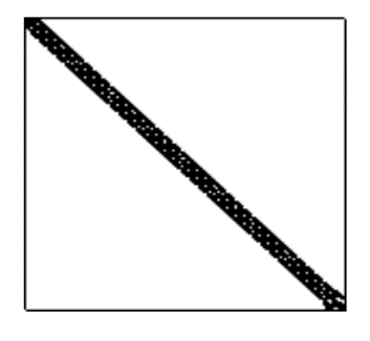
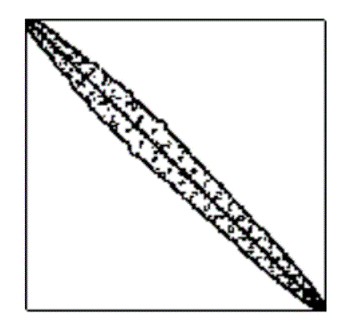
        }

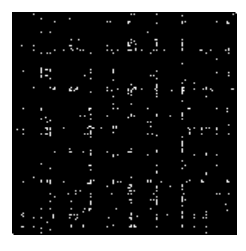
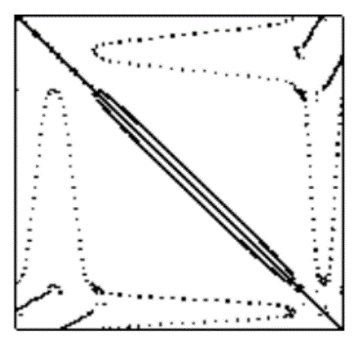
    }

}

三、实验结果

测试文件的详细信息如下所示：

 bcsstm27.rsa： bcsstk15.rsa：

bcsstk24.rsa： psmigr\_3.rua（非稀疏矩阵，但为了提高数据量展示加速比区别，还是选用了该数据文件）：

|  |  |  |
| --- | --- | --- |
| 矩阵名 | 矩阵规模 | 非零元素个数 |
| bcsstm27 | 1224\*1224 | 28675 |
| bcsstk15 | 3948\*3948 | 60882 |
| bcsstk24 | 3562\*3562 | 81736 |
| psmigr\_3 | 3140\*3140 | 543162 |

程序编译命令为（包含-fopenmp参数即可）：

g++ omp.cpp -fopenmp -o omp

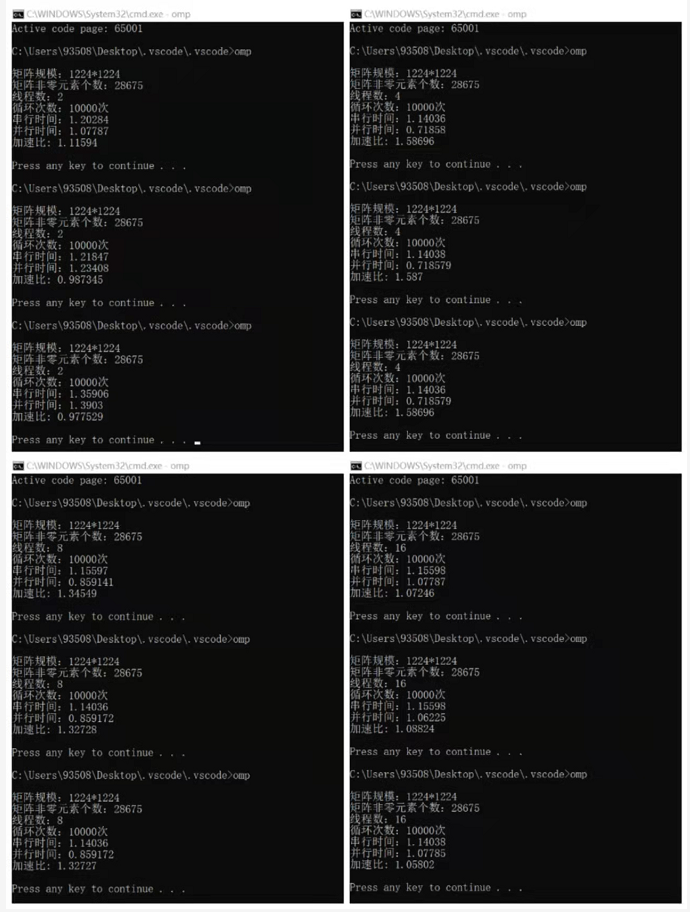
各矩阵具体运行结果见下页：

由于打印信息包含中文（且文件是utf-8编码），所以需要在cmd窗口下输入:

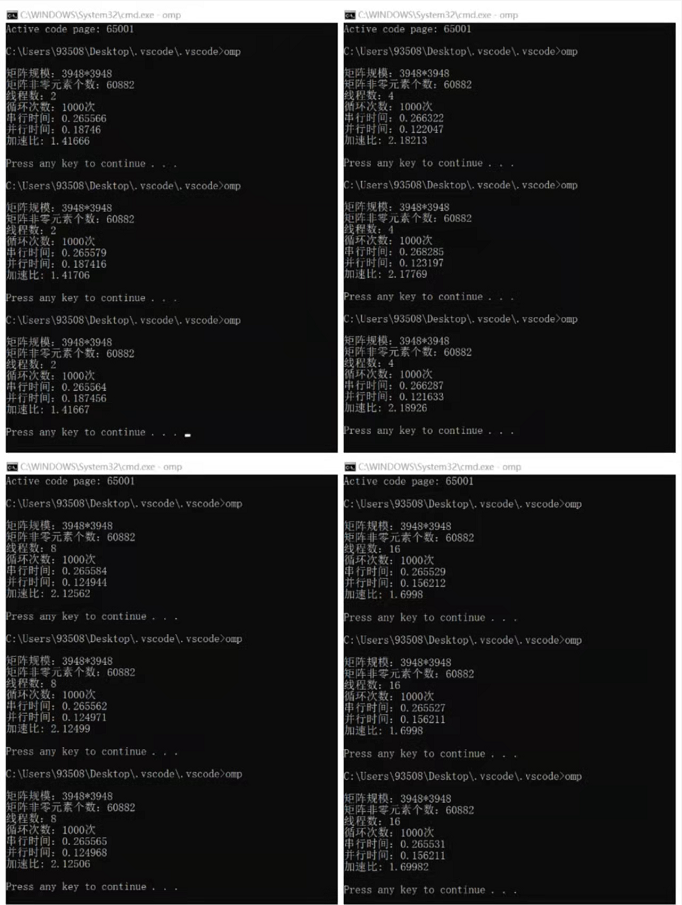
chcp 65001

以此来保证中文正常显示。

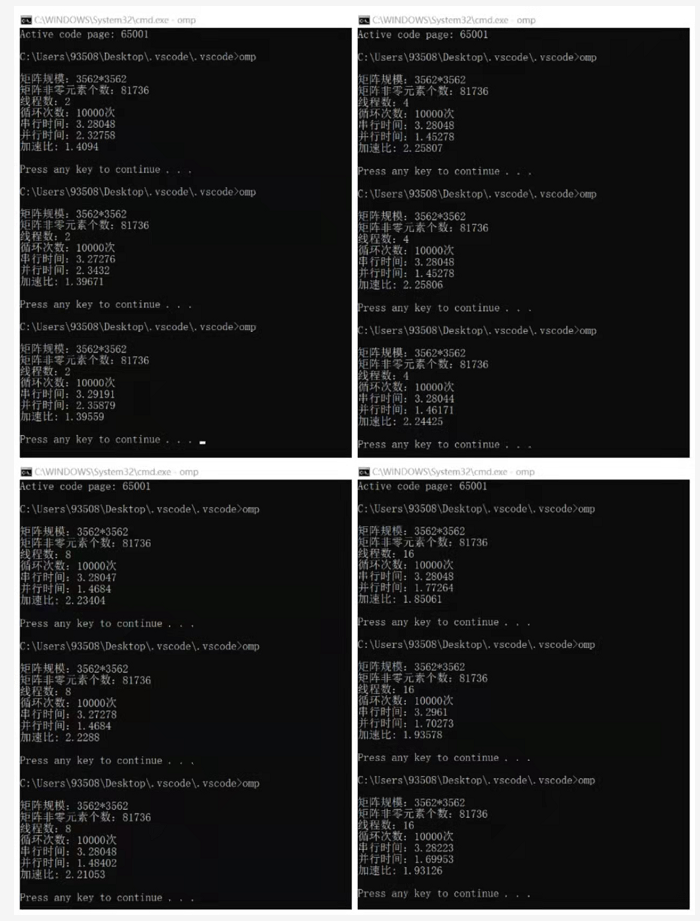
对于bcsstm27.rsa在2,4,8,16线程下的运行结果：



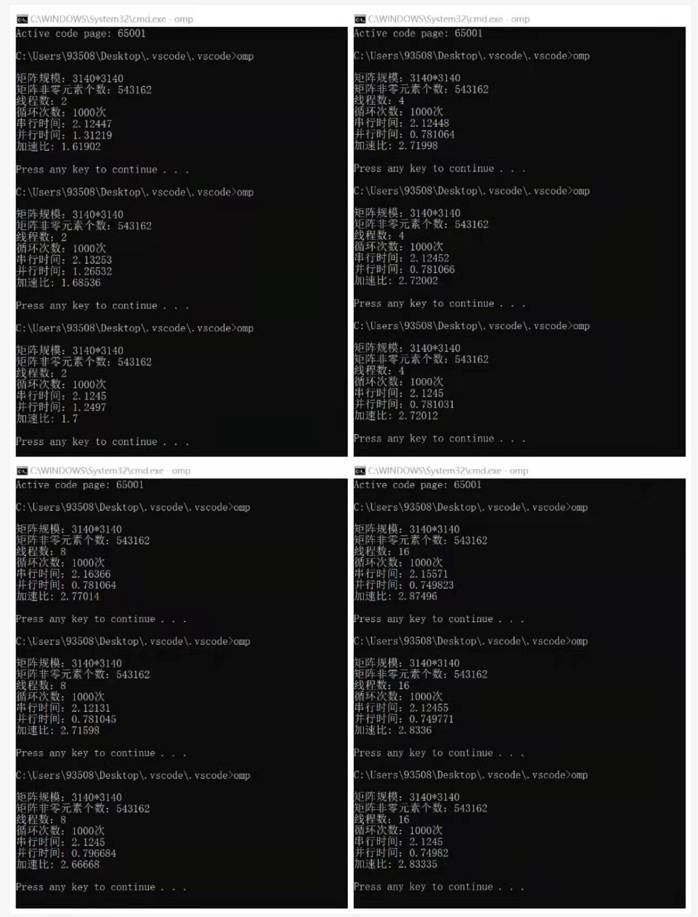
对于bcsstk15.rsa在2,4,8,16线程下的运行结果：



对于bcsstk24.rsa在2,4,8,16线程下的运行结果：



对于psmigr\_3.rua在2,4,8,16线程下的运行结果：



最终结果整合为如下表格：

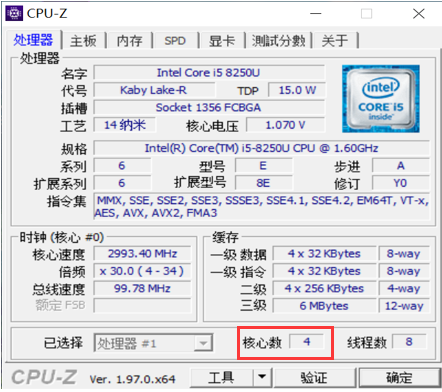
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 矩阵名 | 矩阵规模 | 非零元素个数 | 2线程加速比 | 4线程加速比 | 8线程加速比 | 16线程加速比 |
| bcsstm27 | 1224\*1224 | 28675 | 1.0269 | 1.5870 | 1.3333 | 1.0709 |
| bcsstk15 | 3948\*3948 | 60882 | 1.4168 | 2.1830 | 2.1252 | 1.6998 |
| bcsstk24 | 3562\*3562 | 81736 | 1.4006 | 2.2535 | 2.2245 | 1.9059 |
| psmigr\_3 | 3140\*3140 | 543162 | 1.6681 | 2.7200 | 2.7176 | 2.8473 |

综上结果可分析得知：

（1）对于不同大小的矩阵，在并行线程数相同的情况下，矩阵规模和非零元素数量越大，并行加速比越高；

（2）对于相同大小的矩阵，在并行线程数不同的情况下，加速比随着线程数的增加先升高后降低，并都在线程数为4时达到最高加速比（psmigr\_3除外，该矩阵在线程数为16时达到最高加速比）。

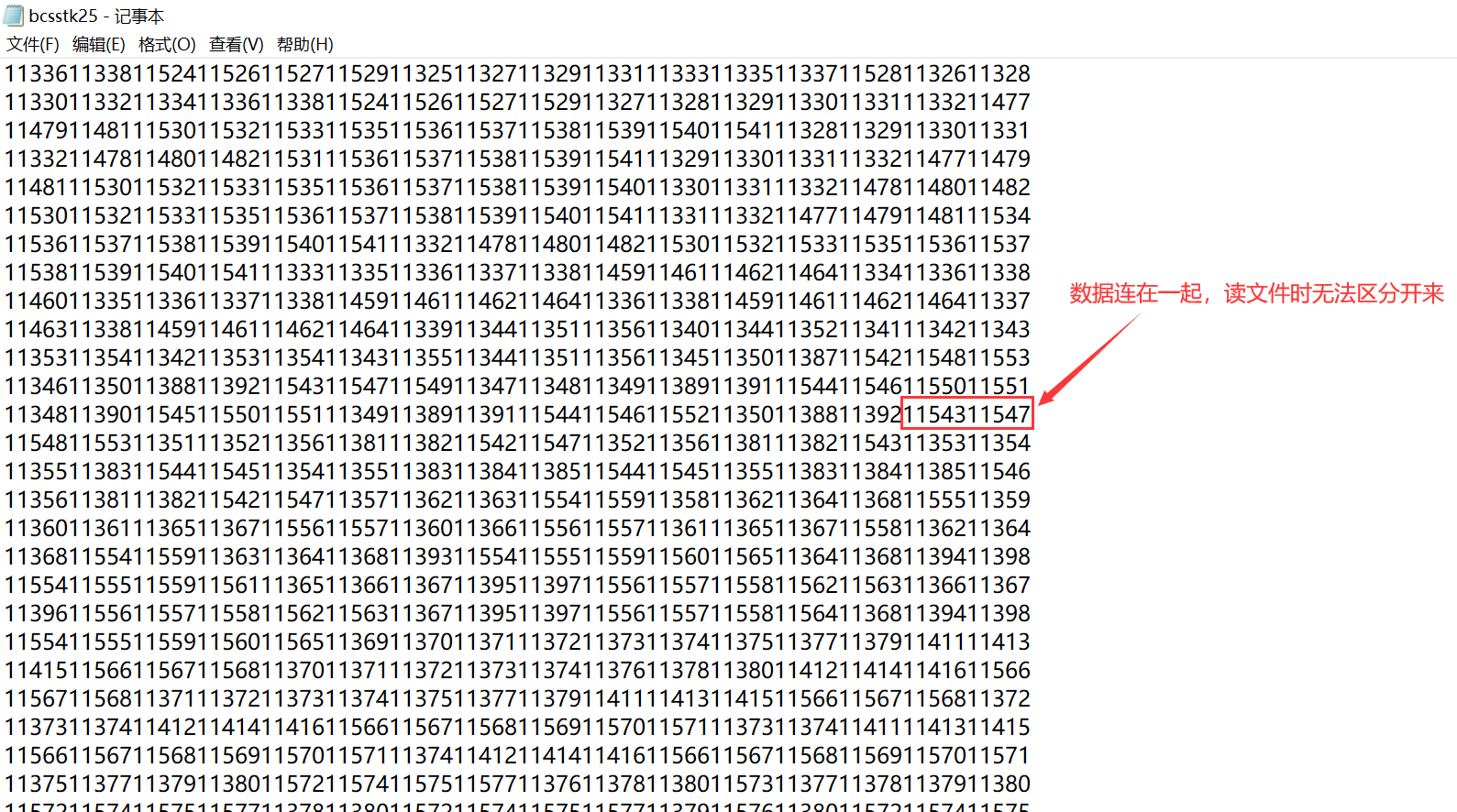
在线程数为4时达到最优加速比的原因是我的电脑为4核的，如下图所示：



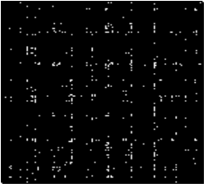
四、遇到的问题及解决方法

该题在写代码时不太顺利，主要体现在两方面：

（1）数据筛选困难，因为Matrix Market上的Harwell-Boeing Collection里的部分大规模的数据集里的数据有些是无法读取的，如下图所示：



所以在根据矩阵规模和非零元素个数大小来挑选矩阵时花费了一部分时间去审核数据。（也正因为需要一个特大规模的矩阵，最后我被迫选择了psmigr\_3.rua这样的非稀疏矩阵，选择它的主要原因还是因为非零元素个数多达54万，便于和其他矩阵比较加速比差异）。



psmigr\_3.rua

（2）一开始我把并行函数写成了下列形式：

int i,j;

#pragma omp parallel num\_threads(thread\_count) default(none) \

      private(i,j) shared(row,col\_pointer,row\_pointer,value,vec,ans)

    for(i=0;i<=row-1;i++){

        #pragma omp for

        for(j=row\_pointer[i];j<=row\_pointer[i+1]-1;j++){//外层循环遍历次数代表第i行的非零元素个数

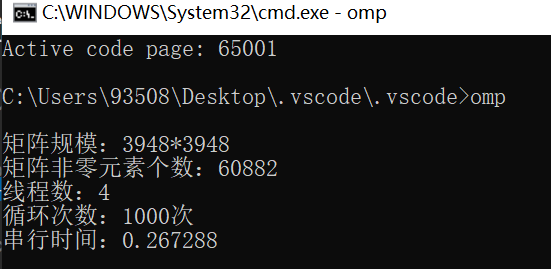
            int pos=col\_pointer[j];//定位第i行非零元素的列坐标

            ans[i]+=value[j]\*vec[pos];//将该元素与列向量的对应位置相乘，并累加到结果向量的对应位置上

        }

    }

发现并行跑不出结果：



经过分析发现内部循环需要累加，不能直接用omp for将其拆开，而是应该对外层循环进行并行化分配到各个线程，于是我又修改成了以下形式：

    int i,j;

#pragma omp parallel for num\_threads(thread\_count)//由于内层循环不可拆分(含累加)且外层各循环之间互不影响

    //所以可将外层循环用omp进行并行化

    for(i=0;i<=row-1;i++){

        for(j=row\_pointer[i];j<=row\_pointer[i+1]-1;j++){//外层循环遍历次数代表第i行的非零元素个数

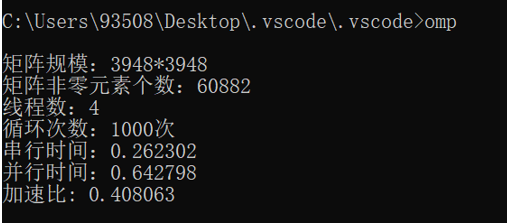
            int pos=col\_pointer[j];//定位第i行非零元素的列坐标

            ans[i]+=value[j]\*vec[pos];//将该元素与列向量的对应位置相乘，并累加到结果向量的对应位置上

        }

    }

但运行后发现并行时间远大于串行时间（以bcsstk15.rsa为例）：



加速比从原来的2.1830降至0.4081。

这次的问题原因我寻找了很久，在询问了同学后得知我应该把i, j设置为for循环内部的局部变量（不要将i,j的定义放在#pragma omp外面），避免线程间频繁通信带来的并行开销。

在将代码修改成以下形式后，我得到了理想的实验结果：

#pragma omp parallel for num\_threads(thread\_count)//由于内层循环不可拆分

(含累加)且外层各循环之间互不影响

    //所以可将外层循环用omp进行并行化

    for(int i=0;i<=row-1;i++){

        for(int j=row\_pointer[i];j<=row\_pointer[i+1]-1;j++){//外层循环遍历次数代表第i行的非零元素个数

            int pos=col\_pointer[j];//定位第i行非零元素的列坐标

            ans[i]+=value[j]\*vec[pos];//将该元素与列向量的对应位置相乘，并累加到结果向量的对应位置上

        }

    }

或者将i,j变量设置为private也可以，代码如下：

    int i,j;

#pragma omp parallel for num\_threads(thread\_count) private(i,j)//由于内层循环不可拆分(含累加)且外层各循环之间互不影响

    //所以可将外层循环用omp进行并行化

    for(i=0;i<=row-1;i++){

        for(j=row\_pointer[i];j<=row\_pointer[i+1]-1;j++){//外层循环遍历次数代表第i行的非零元素个数

            int pos=col\_pointer[j];//定位第i行非零元素的列坐标

            ans[i]+=value[j]\*vec[pos];//将该元素与列向量的对应位置相乘，并累加到结果向量的对应位置上

        }

    }

这个问题我在之前编写的OpenMP程序中从未发现过，然而这次由于操作的对象是稀疏矩阵，所以将这一问题放大了。

（3）一开始在本地VSCode上运行时忘记加编译参数-fopenmp，导致并行串行时间几乎一致。