

# 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 <stdio>
#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：

```
Active code page: 65001  
C:\Users\93508\Desktop\.vscode\.vscode>du  
static调度方式的用时为: 0.406743  
dynamic调度方式的用时为: 0.406167  
guided调度方式的用时为: 0.406163
```

（2）dummy 函数内部为 Sleep(100)时：

```
C:\Users\93508\Desktop\.vscode\.vscode>du  
static调度方式的用时为: 2.73434  
dynamic调度方式的用时为: 2.73371  
guided调度方式的用时为: 2.73376
```

（3）dummy 函数内部为 Sleep(1000)时：

```
C:\Users\93508\Desktop\.vscode\.vscode>du  
static调度方式的用时为: 25.364  
dynamic调度方式的用时为: 25.3847  
guided调度方式的用时为: 25.3696
```

(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

```
C:\Users\93508\Desktop\.vscode\.vscode>du
static调度方式的用时为: 2.74992
dynamic调度方式的用时为: 2.73285
guided调度方式的用时为: 2.76572
```

② size=2

```
C:\Users\93508\Desktop\.vscode\.vscode>du
static调度方式的用时为: 2.86805
dynamic调度方式的用时为: 2.7265
guided调度方式的用时为: 2.84301
```

③ size=4

```
C:\Users\93508\Desktop\.vscode\.vscode>du
static调度方式的用时为: 3.08537
dynamic调度方式的用时为: 2.76035
guided调度方式的用时为: 2.97253
```

④ size=8

```
C:\Users\93508\Desktop\.vscode\.vscode>du
static调度方式的用时为: 3.06235
dynamic调度方式的用时为: 2.73378
guided调度方式的用时为: 2.95836
```

④ size=16

```
C:\Users\93508\Desktop\.vscode\.vscode>du
static调度方式的用时为: 3.57112
dynamic调度方式的用时为: 2.78263
guided调度方式的用时为: 3.56605
```

综上所述，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 and 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> resource; //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(resource.empty()){
    element=1;//若当前队列为空，则加入值为 1 的元素
}
else{
    element=resource.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
resource.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=resource.front();//消费元素为当前队列的队头
        resource.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 and 多个消费者，代码如下所示（具体思路 and 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> resource;//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(resource.empty()){
        element=1;//若当前队列为空，则加入值为 1 的元素
    }
    else{
        element=resource.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
    resource.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=resource.front();//消费元素为当前队列的队头
        resource.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：

```
C:\WINDOWS\System32\cmd.exe
Active code page: 65001
C:\Users\93508\Desktop\.vscode\.vscode>21
No. 1已生产！当前队列资源数为：1
No. 2已生产！当前队列资源数为：2
No. 1已消费！当前队列资源数为：1
No. 3已生产！当前队列资源数为：2
No. 4已生产！当前队列资源数为：3
No. 5已生产！当前队列资源数为：4
No. 2已消费！当前队列资源数为：3
No. 6已生产！当前队列资源数为：4
No. 7已生产！当前队列资源数为：5
No. 8已生产！当前队列资源数为：6
No. 3已消费！当前队列资源数为：5
No. 9已生产！当前队列资源数为：6
No. 10已生产！当前队列资源数为：7
No. 11已生产！当前队列资源数为：8
No. 4已消费！当前队列资源数为：7
No. 12已生产！当前队列资源数为：8
No. 13已生产！当前队列资源数为：9
No. 14已生产！当前队列资源数为：10
No. 5已消费！当前队列资源数为：9
No. 15已生产！当前队列资源数为：10
No. 16已生产！当前队列资源数为：11
No. 6已消费！当前队列资源数为：10
No. 17已生产！当前队列资源数为：11
No. 18已生产！当前队列资源数为：12
No. 7已消费！当前队列资源数为：11
No. 19已生产！当前队列资源数为：12
No. 8已消费！当前队列资源数为：11
```

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：

```
C:\WINDOWS\System32\cmd.exe
Active code page: 65001
C:\Users\93508\Desktop\.vscode\.vscode>22
No. 1 已生产! 线程号为: 2, 当前队列资源数为: 1
No. 2 已生产! 线程号为: 6, 当前队列资源数为: 2
No. 2 已生产! 线程号为: 5, 当前队列资源数为: 3
No. 2 已生产! 线程号为: 1, 当前队列资源数为: 4
No. 2 已生产! 线程号为: 0, 当前队列资源数为: 5
No. 2 已生产! 线程号为: 7, 当前队列资源数为: 6
No. 2 已生产! 线程号为: 4, 当前队列资源数为: 7
No. 2 已生产! 线程号为: 3, 当前队列资源数为: 8
No. 3 已生产! 线程号为: 2, 当前队列资源数为: 9
No. 1 已消费! 线程号为: 11, 当前队列资源数为: 8
No. 2 已消费! 线程号为: 10, 当前队列资源数为: 7
No. 3 已生产! 线程号为: 6, 当前队列资源数为: 8
No. 2 已消费! 线程号为: 8, 当前队列资源数为: 7
No. 3 已生产! 线程号为: 5, 当前队列资源数为: 8
No. 2 已消费! 线程号为: 9, 当前队列资源数为: 7
No. 3 已生产! 线程号为: 1, 当前队列资源数为: 8
No. 3 已生产! 线程号为: 0, 当前队列资源数为: 9
No. 3 已生产! 线程号为: 7, 当前队列资源数为: 10
No. 3 已生产! 线程号为: 4, 当前队列资源数为: 11
No. 4 已生产! 线程号为: 3, 当前队列资源数为: 12
No. 4 已生产! 线程号为: 2, 当前队列资源数为: 13
No. 2 已消费! 线程号为: 11, 当前队列资源数为: 12
No. 4 已生产! 线程号为: 6, 当前队列资源数为: 13
No. 2 已消费! 线程号为: 10, 当前队列资源数为: 12
No. 4 已生产! 线程号为: 5, 当前队列资源数为: 13
No. 2 已消费! 线程号为: 8, 当前队列资源数为: 12
No. 4 已生产! 线程号为: 1, 当前队列资源数为: 13
```

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：

```
C:\WINDOWS\System32\cmd.exe  
Active code page: 65001  
C:\Users\93508\Desktop\.vscode\.vscode>21  
No. 1已生产！当前队列资源数为：1  
No. 2已生产！当前队列资源数为：2  
No. 1已消费！当前队列资源数为：1  
No. 3已生产！当前队列资源数为：2  
No. 2已消费！当前队列资源数为：1  
No. 4已生产！当前队列资源数为：2  
No. 3已消费！当前队列资源数为：1  
No. 5已生产！当前队列资源数为：2  
No. 4已消费！当前队列资源数为：1  
No. 6已生产！当前队列资源数为：2  
No. 5已消费！当前队列资源数为：1  
No. 7已生产！当前队列资源数为：2  
No. 6已消费！当前队列资源数为：1  
No. 8已生产！当前队列资源数为：2  
No. 7已消费！当前队列资源数为：1  
No. 9已生产！当前队列资源数为：2  
No. 8已消费！当前队列资源数为：1  
No. 10已生产！当前队列资源数为：2  
No. 9已消费！当前队列资源数为：1  
No. 11已生产！当前队列资源数为：2  
No. 10已消费！当前队列资源数为：1  
No. 12已生产！当前队列资源数为：2  
No. 11已消费！当前队列资源数为：1  
No. 13已生产！当前队列资源数为：2  
No. 12已消费！当前队列资源数为：1  
No. 14已生产！当前队列资源数为：2  
No. 13已消费！当前队列资源数为：1
```

## 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)

```

```

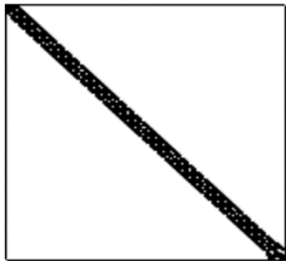
{
#pragma omp parallel for num_threads(thread_count)//由于内层循环不可拆分
(含累加)且外层各循环之间互不影响
//所以可将外层循环用 omp 进行并行化
    for(int i=0;i<=row-1;i++){
        for(int j=row_pointer[i];j<=row_pointer[i+1]-1;j++){//外层循环遍
历次数代表第 i 行的非零元素个数
            int pos=col_pointer[j];//定位第 i 行非零元素的列坐标
            ans[i]+=value[j]*vec[pos];//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
        }
    }
}

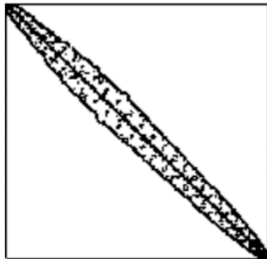
```

### 三、实验结果

测试文件的详细信息如下所示：

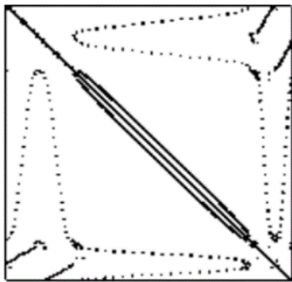


bcsstm27.rsa:

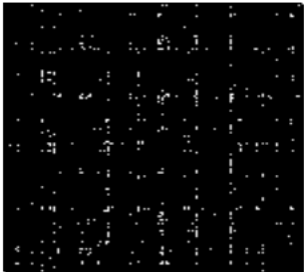


bcsstk15.rsa:

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：

| 线程数 | 串行时间 (s) | 并行时间 (s) | 加速比     |
|-----|----------|----------|---------|
| 2   | 1.20284  | 1.07787  | 1.11594 |
| 4   | 1.14036  | 0.71858  | 1.58696 |
| 8   | 1.14036  | 0.859172 | 1.32728 |
| 16  | 1.14036  | 1.07785  | 1.05802 |

对于 bcsstk15.rsa 在 2,4,8,16 线程下的运行结果:

|  |   |
|--|---|
| <pre>C:\WINDOWS\System32\cmd.exe - omp Active code page: 65001 C:\Users\93508\Desktop\, vscode\, vscode&gt;omp  矩阵规模: 3948*3948 矩阵非零元素个数: 60882 线程数: 2 循环次数: 1000次 串行时间: 0.265566 并行时间: 0.18746 加速比: 1.41666  Press any key to continue . . .  C:\Users\93508\Desktop\, vscode\, vscode&gt;omp  矩阵规模: 3948*3948 矩阵非零元素个数: 60882 线程数: 2 循环次数: 1000次 串行时间: 0.265579 并行时间: 0.187416 加速比: 1.41706  Press any key to continue . . .  C:\Users\93508\Desktop\, vscode\, vscode&gt;omp  矩阵规模: 3948*3948 矩阵非零元素个数: 60882 线程数: 2 循环次数: 1000次 串行时间: 0.265564 并行时间: 0.187456 加速比: 1.41667  Press any key to continue . . .</pre>  | <pre>C:\WINDOWS\System32\cmd.exe - omp Active code page: 65001 C:\Users\93508\Desktop\, vscode\, vscode&gt;omp  矩阵规模: 3948*3948 矩阵非零元素个数: 60882 线程数: 4 循环次数: 1000次 串行时间: 0.266322 并行时间: 0.122047 加速比: 2.18213  Press any key to continue . . .  C:\Users\93508\Desktop\, vscode\, vscode&gt;omp  矩阵规模: 3948*3948 矩阵非零元素个数: 60882 线程数: 4 循环次数: 1000次 串行时间: 0.268285 并行时间: 0.123197 加速比: 2.17769  Press any key to continue . . .  C:\Users\93508\Desktop\, vscode\, vscode&gt;omp  矩阵规模: 3948*3948 矩阵非零元素个数: 60882 线程数: 4 循环次数: 1000次 串行时间: 0.266287 并行时间: 0.121633 加速比: 2.18926  Press any key to continue . . .</pre>  |
| <pre>C:\WINDOWS\System32\cmd.exe - omp Active code page: 65001 C:\Users\93508\Desktop\, vscode\, vscode&gt;omp  矩阵规模: 3948*3948 矩阵非零元素个数: 60882 线程数: 8 循环次数: 1000次 串行时间: 0.265584 并行时间: 0.124944 加速比: 2.12562  Press any key to continue . . .  C:\Users\93508\Desktop\, vscode\, vscode&gt;omp  矩阵规模: 3948*3948 矩阵非零元素个数: 60882 线程数: 8 循环次数: 1000次 串行时间: 0.265562 并行时间: 0.124971 加速比: 2.12499  Press any key to continue . . .  C:\Users\93508\Desktop\, vscode\, vscode&gt;omp  矩阵规模: 3948*3948 矩阵非零元素个数: 60882 线程数: 8 循环次数: 1000次 串行时间: 0.265565 并行时间: 0.124968 加速比: 2.12506  Press any key to continue . . .</pre> | <pre>C:\WINDOWS\System32\cmd.exe - omp Active code page: 65001 C:\Users\93508\Desktop\, vscode\, vscode&gt;omp  矩阵规模: 3948*3948 矩阵非零元素个数: 60882 线程数: 16 循环次数: 1000次 串行时间: 0.265529 并行时间: 0.156212 加速比: 1.6998  Press any key to continue . . .  C:\Users\93508\Desktop\, vscode\, vscode&gt;omp  矩阵规模: 3948*3948 矩阵非零元素个数: 60882 线程数: 16 循环次数: 1000次 串行时间: 0.265527 并行时间: 0.156211 加速比: 1.6998  Press any key to continue . . .  C:\Users\93508\Desktop\, vscode\, vscode&gt;omp  矩阵规模: 3948*3948 矩阵非零元素个数: 60882 线程数: 16 循环次数: 1000次 串行时间: 0.265531 并行时间: 0.156211 加速比: 1.69982  Press any key to continue . . .</pre> |

对于 bcsstk24.rsa 在 2,4,8,16 线程下的运行结果:

```
C:\WINDOWS\System32\cmd.exe - omp
Active code page: 65001
C:\Users\93508\Desktop\, vscode\, vscode>omp
矩阵规模: 3562*3562
矩阵非零元素个数: 81736
线程数: 2
循环次数: 10000次
串行时间: 3.28048
并行时间: 2.32758
加速比: 1.4094
Press any key to continue . . .
C:\Users\93508\Desktop\, vscode\, vscode>omp
矩阵规模: 3562*3562
矩阵非零元素个数: 81736
线程数: 2
循环次数: 10000次
串行时间: 3.27276
并行时间: 2.3432
加速比: 1.39671
Press any key to continue . . .
C:\Users\93508\Desktop\, vscode\, vscode>omp
矩阵规模: 3562*3562
矩阵非零元素个数: 81736
线程数: 2
循环次数: 10000次
串行时间: 3.29191
并行时间: 2.35879
加速比: 1.39559
Press any key to continue . . .

C:\WINDOWS\System32\cmd.exe - omp
Active code page: 65001
C:\Users\93508\Desktop\, vscode\, vscode>omp
矩阵规模: 3562*3562
矩阵非零元素个数: 81736
线程数: 4
循环次数: 10000次
串行时间: 3.28048
并行时间: 1.45278
加速比: 2.25807
Press any key to continue . . .
C:\Users\93508\Desktop\, vscode\, vscode>omp
矩阵规模: 3562*3562
矩阵非零元素个数: 81736
线程数: 4
循环次数: 10000次
串行时间: 3.28048
并行时间: 1.45278
加速比: 2.25806
Press any key to continue . . .
C:\Users\93508\Desktop\, vscode\, vscode>omp
矩阵规模: 3562*3562
矩阵非零元素个数: 81736
线程数: 4
循环次数: 10000次
串行时间: 3.28044
并行时间: 1.46171
加速比: 2.24425
Press any key to continue . . .

C:\WINDOWS\System32\cmd.exe - omp
Active code page: 65001
C:\Users\93508\Desktop\, vscode\, vscode>omp
矩阵规模: 3562*3562
矩阵非零元素个数: 81736
线程数: 8
循环次数: 10000次
串行时间: 3.28047
并行时间: 1.4684
加速比: 2.23404
Press any key to continue . . .
C:\Users\93508\Desktop\, vscode\, vscode>omp
矩阵规模: 3562*3562
矩阵非零元素个数: 81736
线程数: 8
循环次数: 10000次
串行时间: 3.27278
并行时间: 1.4684
加速比: 2.2288
Press any key to continue . . .
C:\Users\93508\Desktop\, vscode\, vscode>omp
矩阵规模: 3562*3562
矩阵非零元素个数: 81736
线程数: 8
循环次数: 10000次
串行时间: 3.28048
并行时间: 1.48402
加速比: 2.21053
Press any key to continue . . .

C:\WINDOWS\System32\cmd.exe - omp
Active code page: 65001
C:\Users\93508\Desktop\, vscode\, vscode>omp
矩阵规模: 3562*3562
矩阵非零元素个数: 81736
线程数: 16
循环次数: 10000次
串行时间: 3.28048
并行时间: 1.77264
加速比: 1.85061
Press any key to continue . . .
C:\Users\93508\Desktop\, vscode\, vscode>omp
矩阵规模: 3562*3562
矩阵非零元素个数: 81736
线程数: 16
循环次数: 10000次
串行时间: 3.2961
并行时间: 1.70273
加速比: 1.93578
Press any key to continue . . .
C:\Users\93508\Desktop\, vscode\, vscode>omp
矩阵规模: 3562*3562
矩阵非零元素个数: 81736
线程数: 16
循环次数: 10000次
串行时间: 3.28223
并行时间: 1.69953
加速比: 1.93126
Press any key to continue . . .
```



对于 psmigr\_3.rua 在 2,4,8,16 线程下的运行结果:

C:\WINDOWS\System32\cmd.exe - omp

Active code page: 65001

C:\Users\93508\Desktop\.vscode\.vscode>omp

矩阵规模: 3140\*3140  
矩阵非零元素个数: 543162  
线程数: 2  
循环次数: 1000次  
串行时间: 2.12447  
并行时间: 1.31219  
加速比: 1.61902

Press any key to continue . . .

C:\Users\93508\Desktop\.vscode\.vscode>omp

矩阵规模: 3140\*3140  
矩阵非零元素个数: 543162  
线程数: 2  
循环次数: 1000次  
串行时间: 2.13253  
并行时间: 1.26532  
加速比: 1.68536

Press any key to continue . . .

C:\Users\93508\Desktop\.vscode\.vscode>omp

矩阵规模: 3140\*3140  
矩阵非零元素个数: 543162  
线程数: 2  
循环次数: 1000次  
串行时间: 2.1245  
并行时间: 1.2497  
加速比: 1.7

Press any key to continue . . .

C:\WINDOWS\System32\cmd.exe - omp

Active code page: 65001

C:\Users\93508\Desktop\.vscode\.vscode>omp

矩阵规模: 3140\*3140  
矩阵非零元素个数: 543162  
线程数: 4  
循环次数: 1000次  
串行时间: 2.12448  
并行时间: 0.781064  
加速比: 2.71998

Press any key to continue . . .

C:\Users\93508\Desktop\.vscode\.vscode>omp

矩阵规模: 3140\*3140  
矩阵非零元素个数: 543162  
线程数: 4  
循环次数: 1000次  
串行时间: 2.12452  
并行时间: 0.781066  
加速比: 2.72002

Press any key to continue . . .

C:\Users\93508\Desktop\.vscode\.vscode>omp

矩阵规模: 3140\*3140  
矩阵非零元素个数: 543162  
线程数: 4  
循环次数: 1000次  
串行时间: 2.1245  
并行时间: 0.781031  
加速比: 2.72012

Press any key to continue . . .

C:\WINDOWS\System32\cmd.exe - omp

Active code page: 65001

C:\Users\93508\Desktop\.vscode\.vscode>omp

矩阵规模: 3140\*3140  
矩阵非零元素个数: 543162  
线程数: 8  
循环次数: 1000次  
串行时间: 2.16366  
并行时间: 0.781064  
加速比: 2.77014

Press any key to continue . . .

C:\Users\93508\Desktop\.vscode\.vscode>omp

矩阵规模: 3140\*3140  
矩阵非零元素个数: 543162  
线程数: 8  
循环次数: 1000次  
串行时间: 2.12131  
并行时间: 0.781045  
加速比: 2.71598

Press any key to continue . . .

C:\Users\93508\Desktop\.vscode\.vscode>omp

矩阵规模: 3140\*3140  
矩阵非零元素个数: 543162  
线程数: 8  
循环次数: 1000次  
串行时间: 2.1245  
并行时间: 0.796684  
加速比: 2.66668

Press any key to continue . . .

C:\WINDOWS\System32\cmd.exe - omp

Active code page: 65001

C:\Users\93508\Desktop\.vscode\.vscode>omp

矩阵规模: 3140\*3140  
矩阵非零元素个数: 543162  
线程数: 16  
循环次数: 1000次  
串行时间: 2.15571  
并行时间: 0.749823  
加速比: 2.87496

Press any key to continue . . .

C:\Users\93508\Desktop\.vscode\.vscode>omp

矩阵规模: 3140\*3140  
矩阵非零元素个数: 543162  
线程数: 16  
循环次数: 1000次  
串行时间: 2.12455  
并行时间: 0.749771  
加速比: 2.8336

Press any key to continue . . .

C:\Users\93508\Desktop\.vscode\.vscode>omp

矩阵规模: 3140\*3140  
矩阵非零元素个数: 543162  
线程数: 16  
循环次数: 1000次  
串行时间: 2.1245  
并行时间: 0.74982  
加速比: 2.83335

Press any key to continue . . .

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：

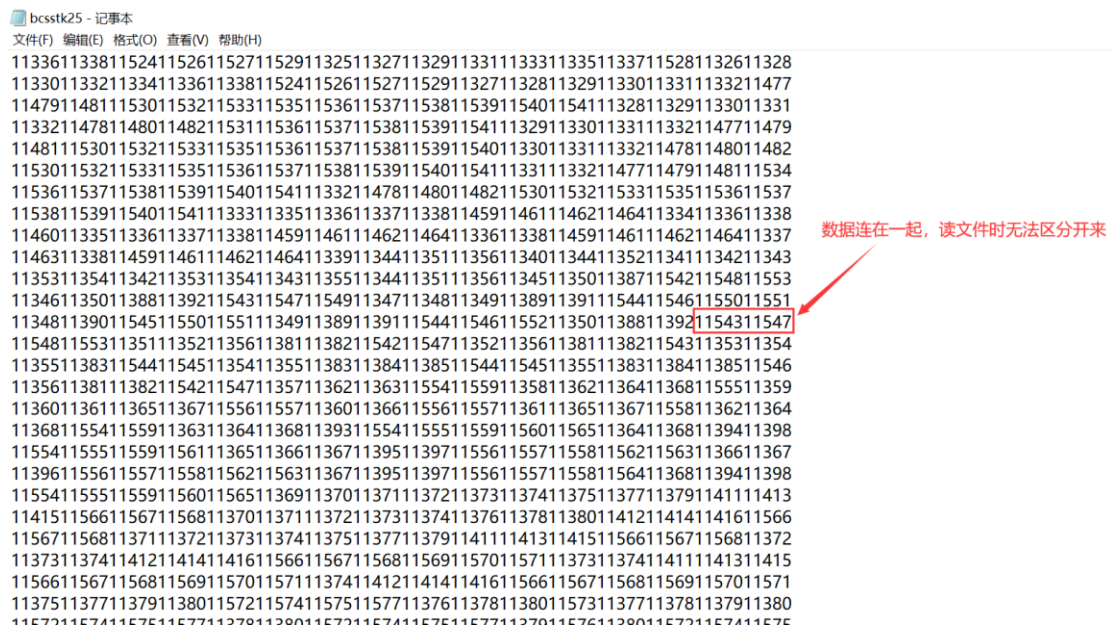




## 四、遇到的问题及解决方法

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：



```
bcstk25 - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
11336113381152411526115271152911325113271132911331113331133511337115281132611328
11330113321133411336113381152411526115271152911327113281132911330113311133211477
11479114811153011532115331153511536115371153811539115401154111328113291133011331
11332114781148011482115311153611537115381153911541113291133011331113321147711479
11481115301153211533115351153611537115381153911540113301133111332114781148011482
11530115321153311535115361153711538115391154011541113311133211477114791148111534
11536115371153811539115401154111332114781148011482115301153211533115351153611537
11538115391154011541113331133511336113371133811459114611146211464113341133611338
11460113351133611337113381145911461114621146411336113381145911461114621146411337
11463113381145911461114621146411339113441135111356113401134411352113411134211343
11353113541134211353113541134311355113441135111356113451135011387115421154811553
11346113501138811392115431154711549113471134811349113891139111544115461155011551
11348113901154511550115511134911389113911154411546115521135011388113921154311547
11548115531135111352113561138111382115421154711352113561138111382115431135311354
11355113831154411545113541135511383113841138511544115451135511383113841138511546
11356113811138211542115471135711362113631155411559113581136211364113681155511359
11360113611136511367115561155711360113661155611557113611136511367115581136211364
11368115541155911363113641136811393115541155511559115601156511364113681139411398
11554115551155911561113651136611367113951139711556115571155811562115631136611367
11396115561155711558115621156311367113951139711556115571155811564113681139411398
1155411555115591156011565113691137011371113721137311374113751137711379114111413
11415115661156711568113701137111372113731137411376113781138011412114141141611566
11567115681137111372113731137411375113771137911411114131141511566115671156811372
11373113741141211414114161156611567115681156911570115711137311374114111141311415
11566115671156811569115701157111374114121141411416115661156711568115691157011571
11375113771137911380115721157411575115771137611378113801157311377113781137911380
115731157411575115771157811579115801158115821158311584115851158611587115881158911590115911592115931159411595115961159711598115991160011601160211603116041160511606116071160811609116101161116121161311614116151161611617116181161911620116211622116231162411625116261162711628116291163011631163211633116341163511636116371163811639116401164116421164311644116451164611647116481164911650116511652116531165411655116561165711658116591166011661166211663116641166511666116671166811669116701167116721167311674116751167611677116781167911680116811682116831168411685116861168711688116891169011691169211693116941169511696116971169811699117001170117021170311704117051170611707117081170911710117111712117131171411715117161171711718117191172011721172211723117241172511726117271172811729117301173117321173311734117351173611737117381173911740117411742117431174411745117461174711748117491175011751175211753117541175511756117571175811759117601176117621176311764117651176611767117681176911770117711772117731177411775117761177711778117791178011781178211783117841178511786117871178811789117901179117921179311794117951179611797117981179911800118011802118031180411805118061180711808118091181011811181211813118141181511816118171181811819118201182118221182311824118251182611827118281182911830118311832118331183411835118361183711838118391184011841184211843118441184511846118471184811849118501185118521185311854118551185611857118581185911860118611862118631186411865118661186711868118691187011871187211873118741187511876118771187811879118801188118821188311884118851188611887118881188911890118911892118931189411895118961189711898118991190011901190211903119041190511906119071190811909119101191119121191311914119151191611917119181191911920119211922119231192411925119261192711928119291193011931193211933119341193511936119371193811939119401194119421194311944119451194611947119481194911950119511952119531195411955119561195711958119591196011961196211963119641196511966119671196811969119701197119721197311974119751197611977119781197911980119811982119831198411985119861198711988119891199011991199211993119941199511996119971199811999200020012002200320042005200620072008200920102011201220132014201520162017201820192020202120222023202420252026202720282029203020312032203320342035203620372038203920402041204220432044204520462047204820492050205120522053205420552056205720582059206020612062206320642065206620672068206920702071207220732074207520762077207820792080208120822083208420852086208720882089209020912092209320942095209620972098209921002101210221032104210521062107210821092110211121122113211421152116211721182119212021212122212321242125212621272128212921302131213221332134213521362137213821392140214121422143214421452146214721482149215021512152215321542155215621572158215921602161216221632164216521662167216821692170217121722173217421752176217721782179218021812182218321842185218621872188218921902191219221932194219521962197219821992200220122022203220422052206220722082209221022112212221322142215221622172218221922202221222222232224222522262227222822292230223122322233223422352236223722382239224022412242224322442245224622472248224922502251225222532254225522562257225822592260226122622263226422652266226722682269227022712272227322742275227622772278227922802281228222832284228522862287228822892290229122922293229422952296229722982299300030013002300330043005300630073008300930103011301230133014301530163017301830193020302130223023302430253026302730283029303030313032303330343035303630373038303930403041304230433044304530463047304830493050305130523053305430553056305730583059306030613062306330643065306630673068306930703071307230733074307530763077307830793080308130823083308430853086308730883089309030913092309330943095309630973098309931003101310231033104310531063107310831093110311131123113311431153116311731183119312031213122312331243125312631273128312931303131313231333134313531363137313831393140314131423143314431453146314731483149315031513152315331543155315631573158315931603161316231633164316531663167316831693170317131723173317431753176317731783179318031813182318331843185318631873188318931903191319231933194319531963197319831993200320132023203320432053206320732083209321032113212321332143215321632173218321932203221322232233224322532263227322832293230323132323233323432353236323732383239324032413242324332443245324632473248324932503251325232533254325532563257325832593260326132623263326432653266326732683269327032713272327332743275327632773278327932803281328232833284328532863287328832893290329132923293329432953296329732983299330033013302330333043305330633073308330933103311331233133314331533163317331833193320332133223323332433253326332733283329333033313332333333343335333633373338333933403341334233433344334533463347334833493350335133523353335433553356335733583359336033613362336333643365336633673368336933703371337233733374337533763377337833793380338133823383338433853386338733883389339033913392339333943395339633973398339934003401340234033404340534063407340834093410341134123413341434153416341734183419342034213422342334243425342634273428342934303431343234333434343534363437343834393440344134423443344434453446344734483449345034513452345334543455345634573458345934603461346234633464346534663467346834693470347134723473347434753476347734783479348034813482348334843485348634873488348934903491349234933494349534963497349834993500350135023503350435053506350735083509351035113512351335143515351635173518351935203521352235233524352535263527352835293530353135323533353435353536353735383539354035413542354335443545354635473548354935503551355235533554355535563557355835593560356135623563356435653566356735683569357035713572357335743575357635773578357935803581358235833584358535863587358835893590359135923593359435953596359735983599360036013602360336043605360636073608360936103611361236133614361536163617361836193620362136223623362436253626362736283629363036313632363336343635363636373638363936403641364236433644364536463647364836493650365136523653365436553656365736583659366036613662366336643665366636673668366936703671367236733674367536763677367836793680368136823683368436853686368736883689369036913692369336943695369636973698369937003701370237033704370537063707370837093710371137123713371437153716371737183719372037213722372337243725372637273728372937303731373237333734373537363737373837393740374137423743374437453746374737483749375037513752375337543755375637573758375937603761376237633764376537663767376837693770377137723773377437753776377737783779378037813782378337843785378637873788378937903791379237933794379537963797379837993800380138023803380438053806380738083809381038113812381338143815381638173818381938203821382238233824382538263827382838293830383138323833383438353836383738383839384038413842384338443845384638473848384938503851385238533854385538563857385838593860386138623863386438653866386738683869387038713872387338743875387638773878387938803881388238833884388538863887388838893890389138923893389438953896389738983899390039013902390339043905390639073908390939103911391239133914391539163917391839193920392139223923392439253926392739283929393039313932393339343935393639373938393939403941394239433944394539463947394839493950395139523953395439553956395739583959396039613962396339643965396639673968396939703971397239733974397539763977397839793980398139823983398439853986398739883989399039913992399339943995399639973998399940004001400240034004400540064007400840094010401140124013401440154016401740184019402040214022402340244025402640274028402940304031403240334034403540364037403840394040404140424043404440454046404740484049405040514052405340544055405640574058405940604061406240634064406540664067406840694070407140724073407440754076407740784079408040814082408340844085408640874088408940904091409240934094409540964097409840994100410141024103410441054106410741084109411041114112411341144115411641174118411941204121412241234124412541264127412841294130413141324133413441354136413741384139414041414142414341444145414641474148414941504151415241534154415541564157415841594160416141624163416441654166416741684169417041714172417341744175417641774178417941804181418241834184418541864187418841894190419141924193419441954196419741984199420042014202420342044205420642074208420942104211421242134214421542164217421842194220422142224223422442254226422742284229423042314232423342344235423642374238423942404241424242434244424542464247424842494250425142524253425442554256425742584259426042614262426342644265426642674268426942704271427242734274427542764277427842794280428142824283428442854286428742884289429042914292429342944295429642974298429943004301430243034304430543064307430843094310431143124313431443154316431743184319432043214322432343244325432643274328432943304331433243334334433543364337433843394340434143424343434443454346434743484349435043514352435343544355435643574358435943604361436243634364436543664367436843694370437143724373437443754376437743784379438043814382438343844385438643874388438943904391439243934394439543964397439843994400440144024403440444054406440744084409441044114412441344144415441644174418441944204421442244234424442544264427442844294430443144324433443444354436443744384439444044414442444344444445444644474448444944504451445244534454445544564457445844594460446144624463446444654466446744684469447044714472447344744475447644774478447944804481448244834484448544864487448844894490449144924493449444954496449744984499450045014502450345044505450645074508450945104511451245134514451545164517451845194520452145224523452445254526452745284529453045314532453345344535453645374538453945404541454245434544454545464547454845494550455145524553455445554556455745584559456045614562456345644565456645674568456945704571457245734574457545764577457845794580458145824583458445854586458745884589459045914592459345944595459645974598459946004601460246034604460546064607460846094610461146124613461446154616461746184619462046214622462346244625462646274628462946304631463246334634463546364637463846394640464146424643464446454646464746484649465046514652465346544655465646574658465946604661466246634664466546664667466846694670467146
```

(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：

```
C:\WINDOWS\System32\cmd.exe - omp
Active code page: 65001

C:\Users\93508\Desktop\.vscode\.vscode>omp

矩阵规模: 3948*3948
矩阵非零元素个数: 60882
线程数: 4
循环次数: 1000次
串行时间: 0.267288
```

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）：

```
C:\Users\93508\Desktop\.vscode\.vscode>omp  
矩阵规模：3948*3948  
矩阵非零元素个数：60882  
线程数：4  
循环次数：1000次  
串行时间：0.262302  
并行时间：0.642798  
加速比：0.408063
```

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。