并行与分布式作业

“OpenMP、MPI 编程”
第五次作业

姓名：TRY
班级：18 级计科 7 班
学号：

一、 问题描述

1.OpenMP 计算稀疏矩阵和向量的乘法

1. Consider a sparse matrix stored in the compressed row format (you may find a description of this format on the web or any suitable text on sparse linear algebra). Write an OpenMP program for computing the product of this matrix with a vector. Download sample matrices from the Matrix Market (http://math.nist.gov/MatrixMarket/) and test the performance of your implementation as a function of matrix size and number of threads.

2.Open-MP 实现生产者-消费者模型

2. Implement a producer-consumer framework in OpenMP using sections to create a single producer task and a single consumer task. Ensure appropriate synchronization using locks. Test your program for a varying number of producers and consumers.

3.利用 MPI 通信程序测试本地进程以及远程进程之间的通信时延和带宽

二、解决方案

实验一：OpenMP 计算稀疏矩阵和向量的乘法

下面使用矩阵市场交易格式来表示稀疏矩阵。

在矩阵市场中，可以看到如下 MM 格式的稀疏矩阵。其中，左边的%括起来的是注释，右边的是下载下来的矩阵的实际内容。

所以，需要**设计循环语句以跳过注释**。

用MM坐标格式可以表示如下。

```
%% MatrixMarket矩阵坐标实数一般
%=============================================== ============================
%
%此ASCII文件表示L的稀疏MxN矩阵
以下矩阵市场格式的%非零值：
%
%+ ---------------------------------------------- +
%| %% MatrixMarket矩阵坐标实数一般 | <-标题行
%|%| <-+
%|%条评论 | |-0条或更多注释行
%|%| <-+
%| MNL | <-行，列，条目
%| I1 J1 A(I1, J1) | <-+
%| I2 J2 A(I2, J2) | |
%| IS J3 A(I3, J3) | |-L线
%| 。。。 | |
%| IL JL A(IL, JL) | <-+
%+ ---------------------------------------------- +
%
%索引基于1，即A(1,1)是第一个元素。
%
%=============================================== ============================
  5 5 8
    1 1 1.000e + 00
    2 2 1.050e + 01
    3 3 1.500e-02
    1 4 6.000e + 00
    4 2 2.505e + 02
    4 4 -2.800e + 02
    4 5 3.332e + 01
    5 5 1.200e + 01
```

1138_bus - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
%%MatrixMarket matrix coordinate real symmetric
1138 1138 2596
1 1  1.4747790000000e+03
5 1 -9.0171330000000e+00
563 1 -5.7306590000000e+00
2 2  9.1366540000000e+00
10 2 -3.4059950000000e+00
563 2 -5.7306590000000e+00
3 3  6.9614680000000e+01
11 3 -8.8105730000000e+00
34 3 -3.1152650000000e+01
35 3 -1.6066840000000e+01
104 3 -4.8692600000000e+00
475 3 -8.7153570000000e+00
```

```
ifstream fin("1138_bus.mtx");
if (!fin)
{
    cout << "打开文件失败！" << endl;
    exit(1);
}
while (fin.peek() == '%')
    while (fin.get() != '\n') ;//这样可以跳过前面的注释
```

本实验中，要求我们用**“行压缩储存格式 Compressed Row Storage(CRS)”**来表示稀疏矩阵。也就是说，需要将上面从矩阵市场下载的 MM 格式的稀疏矩阵转为行压缩储存格式的稀疏矩阵。

（其实，以上面的 MM 格式就可以构造出矩阵来与向量进行乘法，但是这样的空间并没有最大的节省；如果使用行压缩储存格式，则可使用**最少的空间**来储存稀疏矩阵，使空间复杂度最低）

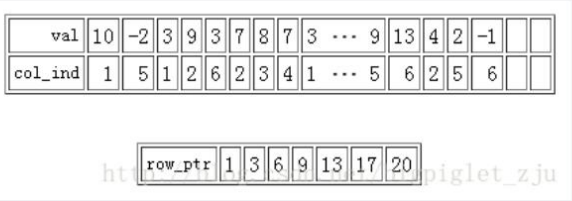
以下为**行压缩格式的矩阵表示：**

- **val 数组**，大小为矩阵 A 的非零元素的个数，保存矩阵 A 的非零元素（按从上往下，从左往右的行遍历方式访问元素）。
- **col_ind 数组**，和 val 数组一样，大小为矩阵 A 的非零元素的个数，保存 val 数组中元素的列索引。
- **row_ptr 数组**，大小为矩阵 A 的行数，保存矩阵 A 的每行第一个非零元素在 val 中的索引。
- **例子：**

矩阵A定义为

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{bmatrix}.$$

其CSR格式由三个数组定义为：



所以，本实验的**难点**在于如何用行压缩形式来表示稀疏矩阵。由于从矩阵市场上下载的矩阵**本来是先按照列排列，再按照行排列**的，也就是储存的第一优先级为列号，第二优先级为行号。

然而，这样不利于我们构造行压缩格式里面的 row_ptr 数组。（由 row_ptr 的定义可知，各单元的内容为该行的第一个非零值在 val 数组中对应的下标，所以，这要求我们按照**先行后列**的顺序排列整个数组中的非零值，与原始的顺序相反）

因此，鉴于本实验的目的在于用 OpenMP 实现行压缩格式储存的稀疏矩阵和向量的相乘，我将原矩阵的行和列调换，也就是**将原矩阵进行转置**，来模拟我们想要的顺序，以**更方便的构造**用行压缩格式存储的稀疏矩阵。这样，可以**避免**复杂的**重新排序**。

```
int x, y;//x=行,y=列
double t;//元素值
int former = -1;
for (int i = 0; i < num; i++)
{
    fin >> y >> x >> t;//读取每一个单元，且第一个数看成列，第二个数看成行
    if ((x - 1) != former)
    {
        row_ptr[x - 1] = i;//第x行的非零元素是i开始的
        former = x - 1;
    }
    val[i] = t;
    col_idx[i] = y-1;
}
```

而行压缩格式储存的稀疏矩阵和向量的乘法函数如下：使用了 omp parallel for 并行处理。

```
Vector mul(Vector& vec)
{
    Vector result(row_ptr.size(),0);
#pragma omp parallel for num_threads(thread_num)//并行！设置了并行线程数
    for (int i = 0; i < row_ptr.size(); i++)
    {
        int end;
        /* ... */
        end = i == row_ptr.size() - 1 ? num - 1 : row_ptr[i + 1] - 1;
        for (int m = row_ptr[i]; m <= end; m++)
        {
            result[i] += val[m] * vec[col_idx[m]];//该行的非零值和vec的对应单元值相乘，累加
        }
    }
    return result;
}
```

以下附上完整代码截图：

```
int row_size, col_size, num;
typedef vector<double> Vector;
//用行压缩方式储存矩阵：
vector<double> val;//大小为num，存值
vector<int> col_idx;//大小为num，存对应值的列值
vector<int> row_ptr;//大小为行数，存每一行第一个非零元素在val中的索引
int thread_num;

Vector mul(Vector& vec)
{
    Vector result(row_ptr.size(),0);
#pragma omp parallel for num_threads(thread_num)//并行！设置了并行线程数
    for (int i = 0; i < row_ptr.size(); i++)
    {
        int end;
        /* ... */
        end = i == row_ptr.size() - 1 ? num - 1 : row_ptr[i + 1] - 1;
        for (int m = row_ptr[i]; m <= end; m++)
        {
            result[i] += val[m] * vec[col_idx[m]];//该行的非零值和vec的对应单元值相乘，累加
        }
    }
    return result;
}

int main()
{
    ifstream fin("1138_bus.mtx");
    if (!fin)
    {
        cout << "打开文件失败！" << endl;
        exit(1);
    }
    while (fin.peek() == '%')
        while (fin.get() != '\n') ;//这样可以跳过前面的注释
    //读取行数、列数、非零值的大小
    fin >> row_size >> col_size >> num;
    val.resize(num);
    col_idx.resize(num);
    row_ptr.resize(row_size,0);

    Vector vec(col_size);
    int x, y;//x=行,y=列
    double t;//元素值
    int former = -1;
```

```
    for (int i = 0; i < num; i++)
    {
        fin >> y >> x >> t;//读取每一个单元，且第一个数看成列，第二个数看成行
        if ((x - 1) != former)
        {
            row_ptr[x - 1] = i;//第x行的非零元素是i开始的
            former = x - 1;
        }
        val[i] = t;
        col_idx[i] = y-1;
    }
    for (int i = 0; i < col_size; i++)
    {
        vec[i] = rand()%100+1;
    }
    cout << "请输入并行的线程数：";
    cin >> thread_num;
    clock_t start = clock();
    Vector result;
    for(int i=0;i<1e4;i++)
        result = mul(vec);
    clock_t end = clock();
    cout << thread_num<<"级线程并行时间(s)：" << (double)(end - start) / CLOCKS_PER_SEC << endl;
    for (auto i:result)
        cout << i << endl;
    system("pause");
    return 0;
}
```

实验二：Open-MP 实现生产者-消费者模型

这个实验是本次最难得一个设计。题目中说要“using sections to create a single producer task and a single consumer task”，又提到了“varying number of producers and consumers”，即要改变消费者和生产者的数量。

但经过思考与讨论，我发现 OpenMP 里面的 section 是不可以与多个生产者-消费者线程并存在同一个程序中的。因为 section 限定该代码是由一个线程执行的，而如果有多个生产者-消费者的话，就会用到 omp parallel for 来进行并行，此时，就不是该段代码只被一个线程执行一次了，存在矛盾。（其实感觉是题目交代不清，导致前后矛盾？）

因此，我写了两个版本的程序。其中**版本 1** 是使用了 section 的，此时，程序中只有一个生产者和一个消费者，两者并行执行。

```
int main()
{
    int num = 0;
    omp_init_lock(&(q.back_mutex));
    omp_init_lock(&(q.front_mutex));
    clock_t start = clock();

#pragma omp parallel num_threads(2)
{
    #pragma omp sections
    {
        #pragma omp section
        {
            producer(250);
        }
        #pragma omp section
        {
            consumer(250);
        }
    }
}
    clock_t end = clock();
    cout << "1个生产者-消费者的时间为：(s)" << (double)(end - start) / CLOCKS_PER_SEC << endl;
    omp_destroy_lock(&(q.front_mutex));
    omp_destroy_lock(&(q.back_mutex));
    return 0;
}
```

版本 2 是可以输入生产者-消费者的数目的，并且用前一半线程（4 个）来执行生产者函数，后一半线程（4 个）来执行消费者函数。且生产者和消费者并行执行。最后输出时间。

```
int main()
{
    int num = 0;
    cout << "请输入生产者-消费者的数量：";
    cin >> num;
    omp_init_lock(&(q.back_mutex));
    omp_init_lock(&(q.front_mutex));
    clock_t start = clock();
    omp_set_num_threads(omp_get_num_procs());//设置线程数
#pragma omp parallel for
    for (int i = 0; i < num; ++i)//多个生产者消费者并行
    {
        int thread_id = omp_get_thread_num();
        if (thread_id < omp_get_num_threads() / 2)
            producer(250);
        else
            consumer(250);
    }
    clock_t end = clock();
    //cout << q.cnt << endl;
    cout <<num<<"个生产者-消费者的时间为：(s)"<< (double)(end - start) / CLOCKS_PER_SEC << endl;
    omp_destroy_lock(&(q.front_mutex));
    omp_destroy_lock(&(q.back_mutex));
    return 0;
}
```

而在文件“MultiAccessQueue.h”中，我设计了一个类“MultiAccessQueue”，继承了基类 queue<int>。即运用了**队列来模拟缓冲区**。这样，就只用处理“缓冲区不空才能消费”的限制，而不用特别处理“缓冲区不满才能生产”的限制（因

为 queue 会自己长空间）。当然用循环数组来实现是最严谨的，但由于此处考察的重点是锁的应用，所以使用了 queue 来处理更方便。所以，只需要处理 pop 时的判断。

并且，此时在派生类 MultiAccessQueue 中重定义 push 和 pop 操作，其中添加了**锁的操作**，并引用了基类 queue 中的 push 和 pop。

```
class MultiAccessQueue :queue<T>//继承queue
{
public:
    omp_lock_t back_mutex;//push用锁
    omp_lock_t front_mutex;//pop用锁
    int cnt = 0;
    void push(T val)
    {
        omp_set_lock(&back_mutex);//获得锁
        queue<T>::push(val);
        omp_unset_lock(&back_mutex);
        return;
    }
    void pop()
    {
        omp_set_lock(&front_mutex); // ...
        if (!queue<T>::empty())
        {
            queue<T>::pop();
            //cnt++;
        }

        omp_unset_lock(&front_mutex);
        return;
    }
};
```

而生产者、消费者函数设计如下。其中，每个消费者和生产者每次消费、生产的个数都是参数 cnt（也就是 250），即每个消费者、生产者每一次会对应消费、生产 250 个产品。而由于每一次消费、生产时间较长，所以不在此处设锁（否则，会导致一段时间内只能由一个生产者、消费者来生产或消费，其他人不能使用）。而是**在真正 push 和 pop 之前加锁**。这样就可以使得一段时间内，不同的消费者都可以消费，不同的生产者都可以生产。（实际上，在每一个时刻，至多只有 1 个生产者在生产，至多只有 1 个消费者在消费。但是，在宏观上一段时间，不同的生产者都可以生产，不同的消费者也都可以消费，只要不同时操作就好）

```
void producer(int cnt)//每个生产者cnt个数，
{
    for (int i = 0; i < cnt; ++i)
        q.push(i);
}
void consumer(int cnt)
{
    for (int i = 0; i < cnt; ++i)
        q.pop();
}
```

实验三：利用 MPI 通信程序测试本地进程以及远程进程之间的通信时延和带宽

编程环境是使用“超算习堂”的 MPI 编程环境。

通信时延用的是在进程 0 中发送数据之前的时间 timesend，和在进程 1 接受完数据之后的时间 timerecv，并将 timerecv 发回到进程 0 中，计算差值，即为通信时延。

带宽用的是 Mbps 作为单位，用“所发送的数据的大小/通信时延”得到带宽。

代码如下：

```
#include <mpi.h>
#include <stdio.h>
#include<memory.h>
#include <stdlib.h>
#include<time.h>
#define MAX 100000000

int main(int argc,char* argv[]) {
    int my_rank, comm_sz;
    MPI_Comm comm;
    MPI_Status status;
    double timesend = 0;
    double timerecv = 0;
    int* buffsend = (int*)malloc(MAX*sizeof(int));
    int* buffrecv = (int*)malloc(MAX*sizeof(int));
    memset(buffsend, 5, MAX);
    memset(buffrecv, 0, MAX);
```

```
    MPI_Init(&argc, &argv);//MPI初始化
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);//获取进程编号
    if (my_rank == 0) {
        double timerecv_in_1 = 0;
        double duration1 = 0;// = timerecv_in_1 - timesend;
        timesend = MPI_Wtime();/*获取时间*/
        MPI_Send(buffsend, MAX, MPI_INT, 1, 0, MPI_COMM_WORLD);

        MPI_Recv(&timerecv_in_1, 1, MPI_DOUBLE, 1, 1, MPI_COMM_WORLD, &status);
        duration1 = timerecv_in_1 - timesend;
        printf("4百万字节的通信时延 :  %lf s\n", duration1);
        printf("带宽 :  %lf Mbps\n", 8*100 * sizeof(MPI_INT) / duration1);
    }
    else if (my_rank == 1) {
        MPI_Recv(buffrecv, MAX, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
        timerecv = MPI_Wtime();
        MPI_Send(&timerecv, 1, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);/*把结束时间传回去*/
    }

    MPI_Finalize();
    return 0;
}
```

三、实验结果

实验一：OpenMP 计算稀疏矩阵和向量的乘法

并行线程数量	时间(s)
1	42.406
2	23.173
4	18.45
8	17.974
16	13.827
32	17.019

可以看出，在线程数=16 的时候，加速比最大。且当线程数<16 的时候，加速比越来越大；当线程数多于 16 的时候，加速比减小。

实验二：Open-MP 实现生产者-消费者模型

生产者-消费者数量	时间（s）

32	0.017
64	0.028
512	0.141
4096	1.082
32768	8.04

可以看出，时间随着生产者-消费者的数量的增加而增加，且因变量的相差的倍数大概等于自变量的增加的倍数。

实验三：利用 MPI 通信程序测试本地进程以及远程进程之间的通信时延和带宽

四、遇到的问题及解决方法

本次实验是有关于 OpenMP 和 MPI 的编程实验，我在其中遇到了很多的困难，其中一部分原因在于无法正确理解老师题目的意思，感觉要求有一些不清晰，也希望下次老师或者 TA 可以具体解释一下题目的含义。

首先来理解omp_set_num_threads()：

作用上来说，我们知道它是用于覆盖环境变量OMP_NUM_THREDS的设置的，使用上来说，要注意的是，omp_set_num_threads只能用于并行区域之外，如果用于并行区域之内，在Debug下运行时会输出“User Error 1001: omp_set_num_threads should only be called in serial regions”到控制台，如果是Release模式不会输出，理论上应该是被忽略了。总之，**在串行代码区调用omp_set_num_threads来设置线程数量**。

然后分析omp_get_num_threads()：

用于获取当前线程组（team）的线程数量，如果不在并行区调用，返回1.

这句话就清楚了描述了get的作用了，获取的是当前线程组的线程数量，所以一般会在并行区域调用，其返回的是实际的parallel区域内由上面几大因素决定之后的实际的运行的线程数量，并不是set的值，所以也很容易理解，在串行区调用它会返回1（所以一般也不会在串行区去调用）。

总结：omp_set_num_threads在串行区域调用才会有效，omp_get_num_threads获取当前线程组的线程数量，一般在并行区域调用，在串行区域调用返回为1。两个函数没有本质上的数量关系！

第一个问题，在于“omp_get_num_threads”的使用。一开始，我在串行区使用了这个命令，尝试获取线程数。但却发现线程数一直是 1。后来，经过查询资料，才发现线程数只有在并行区才会正确返回，在串行区只会返回 1。


1138_bus.mtx

第二个问题，在于“1138_bus.mtx”文件的读取。由于是 1138_bus.mtx.gz 是一个压缩文件，要解压才会变成我们需要的 1138_bus.mtx，这才是可读文件。

第三个问题，在于如何用行压缩格式来储存稀疏矩阵。一开始，以为只有重新排序才可以构造出行压缩格式储存的稀疏矩阵。后来，经过与同学的讨论，才发现可以通过转置来实现。

第四个问题，在于如何计算远程通信和本地通信之间的时延。由于实在无法成功搭建远程通信的平台，最终只完成了本地通信之间的时延的计算。