并行与分布式计算作业

大作业

姓名：郝裕玮

班级：计科 1 班

学号：18329015

一、 问题描述

实现一个并行的线性求解器，求解线性方程的解

二、 解决方案

求解线性方程组的通用方法为高斯消元法。具体步骤如下：

（1）将 $Ax=b$ 的矩阵写为增广矩阵（A,b）；

（2）选取每一行的主元并利用该主元消去所在列中的剩余元素，最终可将矩阵转换为上三角矩阵；

（3）利用回代法即可求出方程组的解向量 x

其中对于（2），注意事项如下：

上述过程称为**高斯消去法**(Gaussian elimination)，但必须对其进行改进，以适用于大多数情况。如果 $a_{kk}=0$，则不能使用第 k 行消除第 k 列的元素，而需要将第 k 行与对角线下的某行进行交换，以得到一个非零主元。如果不能找到非零主元，则线性方程组的系数矩阵是奇异的，因此线性方程组不存在惟一解。

同时，（2）的选取主元的方法具体为：

选主元策略的目的在于将元素中的最大绝对值移到主对角线上，然后用其消去列中的剩余元素。如果在第 p 列中存在多个非零元素，则要从中选择一个进行行交换。例 3.18 中的**偏序选主元**策略(partial pivoting strategy)是最常用的一个，而且用在程序 3.2 中。为了减少误差的传播，偏序选主元策略首先检查位于主对角线或主对角线下方第 p 列的所有元素，确定行 k，它的元素的绝对值最大，即

$$|a_{kp}| = \max\{|a_{pp}|, |a_{p+1\,p}|, \cdots, |a_{N-1\,p}|, |a_{Np}|\}$$

然后如果 $k>p$，则交换第 k 行和第 p 行。现在，每个倍数 m_{rp} 的绝对值，$r=p+1$，将小于或等于 1。这样就保证了定理 3.9 中的矩阵 $\boldsymbol{U}$ 与初始系数矩阵 $\boldsymbol{A}$ 的对应元素的相对大小一致。在偏序选主元策略中，通常选择更大的主元元素会导致更小的传播误差。

在 3.5 节中，可以看到求解 $N\times N$ 线性方程组需要总共 $(4N^3+9N^2-7N)/6$ 次算术操作。当 $N=20$ 时，总的算术操作次数为 5910，在计算过程中的误差传播将导致错误的结果。**按比例偏序选主元**（scaled partial pivoting）策略或**平衡**（equilibrating）策略可用来进一步减少误差传播。在按比例偏序选主元法中，搜索位于主对角线或主对角线下方第 p 列的元素，此元素满足在所在行中其绝对值相对最大。首先搜索第 p 行到第 N 行中绝对值最大的元素，称为 s_r：

$$s_r = \max\{|a_{rp}|, |a_{rp+1}|, \cdots, |a_{rN}|\} \quad \text{其中 } r=p,\ p+1,\ \cdots,\ N \tag{13}$$

通过求下式确定第 k 行：

$$\frac{|a_{kp}|}{s_k} = \max\left\{\frac{|a_{pp}|}{s_p}, \frac{|a_{p+1\,p}|}{s_{p+1}}, \cdots, \frac{|a_{Np}|}{s_N}\right\} \tag{14}$$

现在交换第 p 行和第 k 行，除非 $p=k$。这样也是为了保证定理 3.9 中的矩阵 $\boldsymbol{U}$ 与初始系数矩阵 $\boldsymbol{A}$ 的对应元素的相对大小一致。

由上述方法可知：在将增广矩阵化为上三角时，因为我们每一次只清除一列元素，而每一行的元素需要进行一次数字运算，且行与行之间没有数据依赖，所以该部分可以并行。同时对于回代法同样可以对该部分进行并行化处理。

具体代码实现如下所示（具体分析已包含在注释中）：

```
#include <iostream>
#include <omp.h>
#include <cstdlib>
#include <ctime>
#include <cmath>
#include <stack>
#include <sys/time.h>
#include <time.h>
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
//该结构体用于计算并行计算和串行计算的运行时间

//命令行编译: g++ -fopenmp gauss.cpp
//求解 Ax=b

//A 矩阵生成
void matGene(double *A, int size){
    for(int i = 0; i <= size-1; i++){
        for(int j = 0; j <= size-1; j++){
            //将 A[i][j]存储到一维数组 A[]中
            A[i * size + j] = rand() % 10;
        }
    }
}

//b 向量生成
void vecGene(double *b, int size){
    for (int i = 0; i <= size-1; i++){
```

```
        b[i] = rand() % 5; //A[i]
    }
}

//展示矩阵 A
void matShow(double *A, int size){
    //按行打印矩阵
    for (int i = 0; i <= size-1; i++){
        for (int j = 0; j <= size-1; j++){
            cout << A[i * size + j] << " ";
        }
        cout << endl;
    }
}

//展示向量 b
void vecShow(double *b, int size){
    for (int i = 0; i < size; i++){
        cout << b[i] << endl;
    }
}

int main() {
    //设置矩阵(n*n)和向量规模(n*1)
    int n = 1000;
    //初始化
    double* A = new double[n * n + 1];
    double* b = new double[n + 1];
    int* serial_num = new int[n + 1];

    //随机生成矩阵 A 和向量 b
    srand(time(NULL));
    matGene(A, n);
    vecGene(b, n);

    //用于展示初始化矩阵 A 和向量 b
    //cout << "A = " << endl;
    //matShow(A, n);
    //cout << "b = " << endl;
    //vecShow(b, n);


    for(int i = 0; i < n; i++){
        serial_num[i] = -1;
    }
```

```
    stack<int> s1;
    double start,end;

    //记录计算开始时间
    GET_TIME(start);
    //设置 omp 并行线程数量
    int num_threads = 4;

    //将增广矩阵转化为上三角矩阵
    for(int j = 0; j < n; j++){
        double max_coff = 0;
        int max_index;
        for(int i = 0; i < n; i++){
            if(serial_num[i] == -1 && abs(A[i * n + j]) >
abs(max_coff)){
                max_coff = A[i * n + j];
                max_index = i;
            }
        }
        serial_num[max_index] = j;
        s1.push(max_index);
        #pragma omp parallel for num_threads(num_threads)
        for(int i = 0; i < n; i++){
            if(serial_num[i] == -1){
                double tmp_coff = A[i * n + j] / A[max_index * n + j];
                A[i * n + j] = 0;
                for(int k = j + 1; k < n; k++){
                    A[i * n + k] -= tmp_coff * A[max_index * n + k];
                }
                b[i] -= tmp_coff * b[max_index];
            }
        }
    }

    //对上三角矩阵进行回代求解
    double* result = new double[n + 2];
    for(int j = n - 1; j >= 0; j--){
        int this_index = s1.top();
        s1.pop();
        serial_num[this_index] = -1;
        #pragma omp parallel for num_threads(num_threads)
        for(int i = 0; i < n; i++){
            if(serial_num[i] != -1){
```

```cpp
                double tmp_coff = A[i * n + j] / A[this_index * n + j];
                A[i * n + j] = 0;
                b[i] -= tmp_coff * b[this_index];
            }
        }
        b[this_index] /= A[this_index * n + j];
        A[this_index * n + j] = 1;
        result[j] = b[this_index];
    }
    GET_TIME(end);
    //可展示解向量 x
    //cout << "x = " << endl;
    //vecShow(result, n);

    //输出并行运算时间
    cout << "Time: " << end-start << endl;

    return 0;
}
```

三、 实验结果

以下结果均在超算习堂上运行得出。

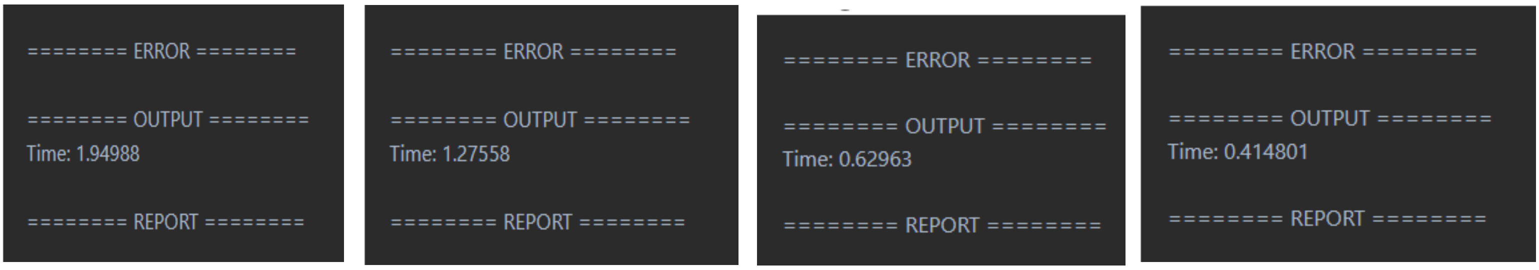
我们只需要将代码中的同样的 2 行代码

```cpp
#pragma omp parallel for num_threads(num_threads)
```

注释掉，即可将该程序变为串行程序。

（1）不同线程数量下的运行时间如下图所示：

矩阵规模为 1000*1000，线程数量依次为 1（串行）、2、4、8

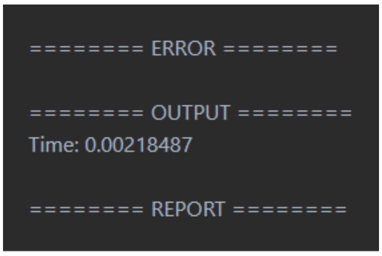


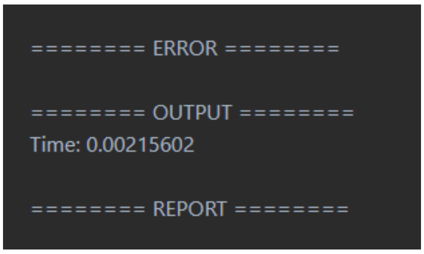
易计算得 2,4,8 线程相对于串行算法的加速比分别为：1.529，3.097，4.701，即加速比随着线程数量的增加越来越高。

（2）不同矩阵规模下的运行时间如下图所示：

线程数为 4，矩阵规模为 100*100，500*500,1000*1000

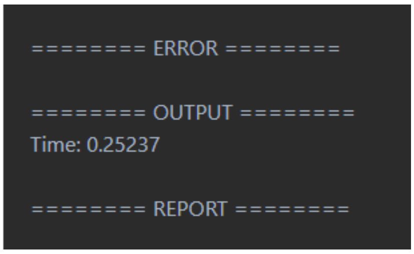
对于 100*100，左图为串行，右图为并行：

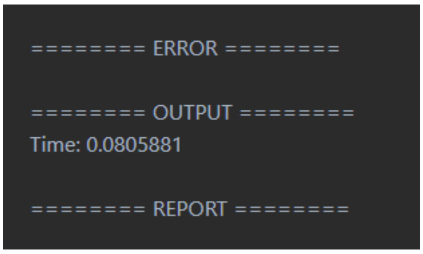




所以加速比为：1.013

对于 500*500，左图为串行，右图为并行：



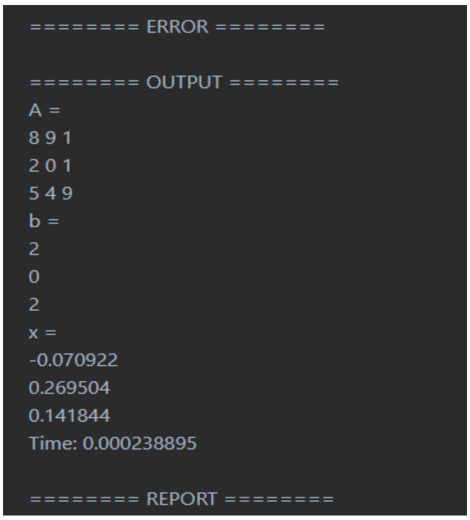


所以加速比为：3.132

对于 1000*1000，（1）中已计算过，加速比为 3.097。

由上图结果可知，并行加速比随着矩阵规模的增大而增加。

为证明方程组的解向量 x 计算正确，我们取如下方程组进行验证（矩阵规模为 3*3）：



易证计算结果正确（右图网址为：http://www.ab126.com/shuxue/2693.html）

四、遇到的问题及解决方法

本次实验由于之前有在数值计算上学习过高斯消元法的相关知识，所以只需对着书本即可实现串行算法而在此基础上的 OpenMP 并行算法也并不难，只需要在两个 for 循环部分加上 OpenMP 的并行语句即可。