SPARSE MATRIX - MATRIX MULTIPLICATION USING MPI + OPENMP

Varun Jain 21963986, Kieron Ho 20500057

Due Date: 2rd/Nov/2018

Assignment 3

CITS3402 High Performance Computing

Table of Contents

1. How To Compile The Program

1.1 Sequential Execution

1. To compile the sequential code:

   gcc -std=c99 -Wall -Werror -pedantic -o filename filename.c

2. Execute Sequential Code

   ./filename file_text.mtx file_text_2.mtx

1.2 openMP + MPI Execution

1. To compile the openMP code:

   mpicc -fopenmp -std=c99 -o <exeName> <filename.c>

2. Run syncCluster
3. To execute the openMp code:

mpirun --hostfile <hostFileName> <exeName> <matrixOne_data> <matrixOne_data> <openMPThreadCount>

Note: <hostfilename> user stores the number of nodes

2. Introduction

A matrix is a two-dimensional array consists of m rows and n columns, where each (m, n) cell is filled with a non-zero or zero element. Sparse matrices contain only a small-fraction of non-zero elements (around 10%), while the rest of the cells are filled with zero elements. There are many different forms of sparse matrices, the one we are using as input for the matrix multiplication computation is in the form of a market matrix. Market matrix is same as a sparse representation except that, it excludes all the 0 values from the matrix input file, reducing it to only non-zero values. By eliminating all the zero terms, there will be smaller number of entries in the matrix files which is both beneficial for the memory management and parallel computations.

Data stored within the sparse matrix is in a triplet format. Triplet format only stores non-zero entries which records the row index, column index and the value of the first, second and third entry, retrospectively.

3. Matrix Multiplication Implementation

The implementation for the matrix multiplication is broken down into several smaller parts, which will soon be explained in this section. The sequential and MPI + openMP implementation can be found in appendix A and appendix B, retrospectively. The implementation is written in C. Refer to either appendix A or B for more detailed explanation about the algorithm used in this project.

3.1 Sequential Implementation

Shown below are some segments of the sequential C program for matrix multiplication computation. Overall, the idea for this implementation is to compute the matrix multiplication, by giving two sparse matrices (market matrix representation) through the command line argument, as inputs. After computation, the program should store the output in a text-file called matrixMultiplication.txt, in a triplet form. The first, second and third will indicate the i^{th} row, j^{th} column and the value of the i^{th} and j^{th} index, retrospectively.

To compile the sequential code:

```
gcc -std=c99 -Wall -Werror -pedantic -o sequentialmatrix sequentialmatrix.c
```

To run the sequential code:

```
./sequentialmatrix 1138_bus.mtx 1138_bus_1.mtx
```

3.1.1. Matrix Multiplication Execution

During initial stages of the program's execution, the data contained inside the input files will be read line by line. Each string will be broken down into a series of small tokens. Each token will be stored in a separate array. In order words, the first, second and third column of the input file were stored in three separate arrays.

3.1.1 Matching Row and Column

The algorithm designed below iterates through the column index from 1138_bus.mtx and row index from 1138_bus_1.mtx. For each element in column index from 1138_bus.mtx, the program will iterate through row index from 1138_bus_1.mtx and checks whether column index matches the row index. If the two index's match then the algorithm will multiply those values of the column and row index and store the value in 'sum_of_the_values' variable.

```
void multiply()
{
    int resultantIndex = 0;
    for(int matOneIndex = 0; matOneIndex < file_one_number_of_lines
    ; matOneIndex++)
    {

        for(int matTwoIndex = 0; matTwoIndex <
    file_two_number_of_lines; matTwoIndex++)
        {
            float sum_of_the_values = 0;
            if(column1[matOneIndex] == row2[matTwoIndex])
            {
                sum_of_the_values += value1[matOneIndex]*value2[
    matTwoIndex];
            }
            if(sum_of_the_values != 0)
            {
                rowOutput[resultantIndex] =  row1[matOneIndex];
                columnOutput[resultantIndex] = column2[matTwoIndex
    ];
                valueOutput[resultantIndex] = sum_of_the_values;
                resultantIndex++;
            }
        }
    }
}
```

As long as "sum_of_the_values" is non-zero element, the element will be inserted into the valueOutput array. Alongside with that, the non-zero element from the row index from 1138_bus.mtx and column index from 1138_bus_1 will be stored in "rowOutput", and "columnOutput" array, retrospectively. As a result, there will be many duplicates of row and columns indexes stored in the rowOutput and columnOutput array, retrospectively. Each row and column index stored in the arrays will have its own value.

3.1.2 Matrix Multiplication Computation

The algorithm designed below basically iterates through the rowOutput and coumnOutput to find matching row and column indexes, within the 1D array. The outer loop iterates through the rowOutput and columnOutput from index 0, while the inner loop iterates through the same array, but starting at index 1. This iteration will help us to locate the same row and column index values in the array and add up the values if the same row and column index is found.

```
void compute_matrix_multiplication(char filename[])
{
    FILE *fp = fopen(filename, "w+");
    int total_number_resultant_columns = sizeof(rowOutput)/sizeof(
    rowOutput[0]);

    for(int output_index = 0; output_index <
    total_number_resultant_columns; output_index++)
    {
        for(int resultant_index = output_index+1; resultant_index <
     total_number_resultant_columns; resultant_index++)
        {
            if(rowOutput[output_index] == rowOutput[resultant_index
    ] && columnOutput[output_index] == columnOutput[resultant_index
    ])
            {
                valueOutput[output_index] = valueOutput[
    output_index]+valueOutput[resultant_index];
                rowOutput[resultant_index] = 0;
                columnOutput[resultant_index] = 0;
                valueOutput[resultant_index] = 0;
            }
        }

        if(rowOutput[output_index] != 0 && columnOutput[
    output_index] != 0 && valueOutput[output_index] != 0)
        {
            printf("%d %d %f\n", rowOutput[output_index],
    columnOutput[output_index], valueOutput[output_index]);
            fprintf(fp, "%d %d %f\n", rowOutput[output_index],
    columnOutput[output_index], valueOutput[output_index]);
        }
    }
}
```

The addition of the individual output values will return the total value for a specific row and its corresponding column index and its value will be stored in the valueOutput array. Concurrently, the duplicate row and column indexes will be replaced with 0’s. This should compute the matrix multiplication – the three individual output arrays will consist of a unique row index, column index and the value consisting of no non-zero duplicated values.

The final output will be printed out on the terminal and will be stored in the matrixmultiplication.txt in the format of a triplet. So like, {row index, column index, value}. Only non-zero values will be computed into the text file and seen on the terminal screen.

3.2 openMP + MPI Implementation

Both OpenMP and MPI are parallelizing performance enhancing paradigms that function on two distinct levels. We have developed an algorithm to process sparse matrix multiplication compressed into market matrix form in conjunction with both OpenMP and MPI to improve the calculation times.

Step 1: Compile openMP Code

```
mpicc -fopenmp -std=c99 -o matrixMultiplication matrixMultiplicationAllParallel.c
```

Step 2: Run syncCluster

Step 3: Execute openMP code:

```
mpirun --hostfile host matrixMultiplication 1138_bus.mtx 1138_bus.mtx 2
```

3.2.1 OpenMP (Open Multi-Processing)

This parallelization technique functions within a single processing unit (computer) and utilizes the multi-threading capability of processors. With OpenMP tasks defined within a program can be split into multiple threads to be computed simultaneously on a single multi-thread enabled processor. This increases the processing speed as a single task broken down can be resolved more rapidly than if a sequential execution was performed.

3.2.2 MPI (Message Passing Interface)

This parallelization technique functions over multiple processing units (computers) and enables the distribution of data for execution and resolution over those computers simultaneously. The ability to break down and distribute a task to a network of computers results in faster execution of data processing.

3.2.3. Algorithm Implementation

Our algorithm has been designed to load into the master node two specified files that contain 1138 by 1138 matrix data stored in market matrix format. It will maintain the market matrix format throughout its process and will return the results as either a text file or onto the terminal.

3.2.3.1 Our Algorithm with Sequential Execution

The processing method of our algorithm compares each tuple of matrix one to each tuple of matrix two. This is done one matrix one tuple at a time, and so can be seen to have a complexity of matrix one size times matrix one size. Without sorting and improving the efficiency of the algorithm, this could take a long time on larger matrices.

3.2.3.2 Our Algorithm and MPI Data Flow

The matrix information is first read and stored by the master node. It then determines the number of worker nodes it can connect to and splits matrix one tuples into near equitable divisions. Each worker node then receives a fraction of matrix one and the full measure of matrix two. These worker nodes perform matrix multiplication on the two provided matrices before sending the results back to the master node, which collects and processes the sum of worker responses to produce a resultant matrix in market matrix form.

File 1 **File 2**

Parsed into master node

↓ ↓

Master Node

Split into near equitable divisions and sent to worker nodes

↓ ↓ ↓ ↓

Worker Nodes

Process the provided matrices and send back to master node

↓

Master Node

Receives and processes the results into an overall result

↓

Text file or Terminal

3.2.3.3 Our Algorithm and OpenMP Data Flow

The OpenMP implementation found in our algorithm only occurs in the worker nodes. After receiving the two matrices to multiply together, the OpenMP directive further divides matrix one into tuple groups that are multiplied through the tuples of matrix two. The results of this simultaneous evaluation are then combined together into an intermediate matrix and sent back to the master node.

3.2.3.4 Combining MPI and OpenMP

In order to combine MPI and OpenMP, the MPI_Init(int *argc, char ***argv) function must be replaced with MPI_Init_thread(int *argc, char ***argv, int required, int *provided). This enables the OpenMP function omp_set_num_threads() to change the number of threads the program uses. In the case of our algorithm the OpenMP multithreading will only take place in the worker nodes.

4. Results

In this section, we will compute a table and graph showing the run time execution of both the sequential code and the hybrid openMP + MPI computation. The results in the data will be recorded in milliseconds(ms).

4.1 MPI + openMP Runtime Computation

For the hybrid implementation, we have done nine experiments in total to find out the runtime for the number of different nodes and threads computed for the 1138 by 1138 matrix computation.

Nodes	Threads	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
Sequential Code		297	288	295	294	297	294
1	**2**	293	490	547	292	486	421.6
1	**4**	173	179	185	165	338	208
1	**6**	173	176	188	177	207	184.2
2	**2**	427	232	241	193	185	255.6
2	**4**	182	198	195	252	243	214
2	**6**	242	192	187	198	196	203
4	**2**	325	243	199	251	204	244.4
4	**4**	214	206	215	213	372	244
4	**6**	317	212	242	217	236	244.8
6	**2**	254	523	446	522	587	466.4
6	**4**	255	227	250	281	380	278.6
6	**6**	235	365	249	216	163	245.6

Figure 1. Runtime

4.2 Graphical Representation of Results

For each experiment, we have computed 5 trails in order to achieve a higher accuracy of results. Overall, there is a slight improvement in the runtime for most experiments over the sequential runtime, which remains relatively constant.

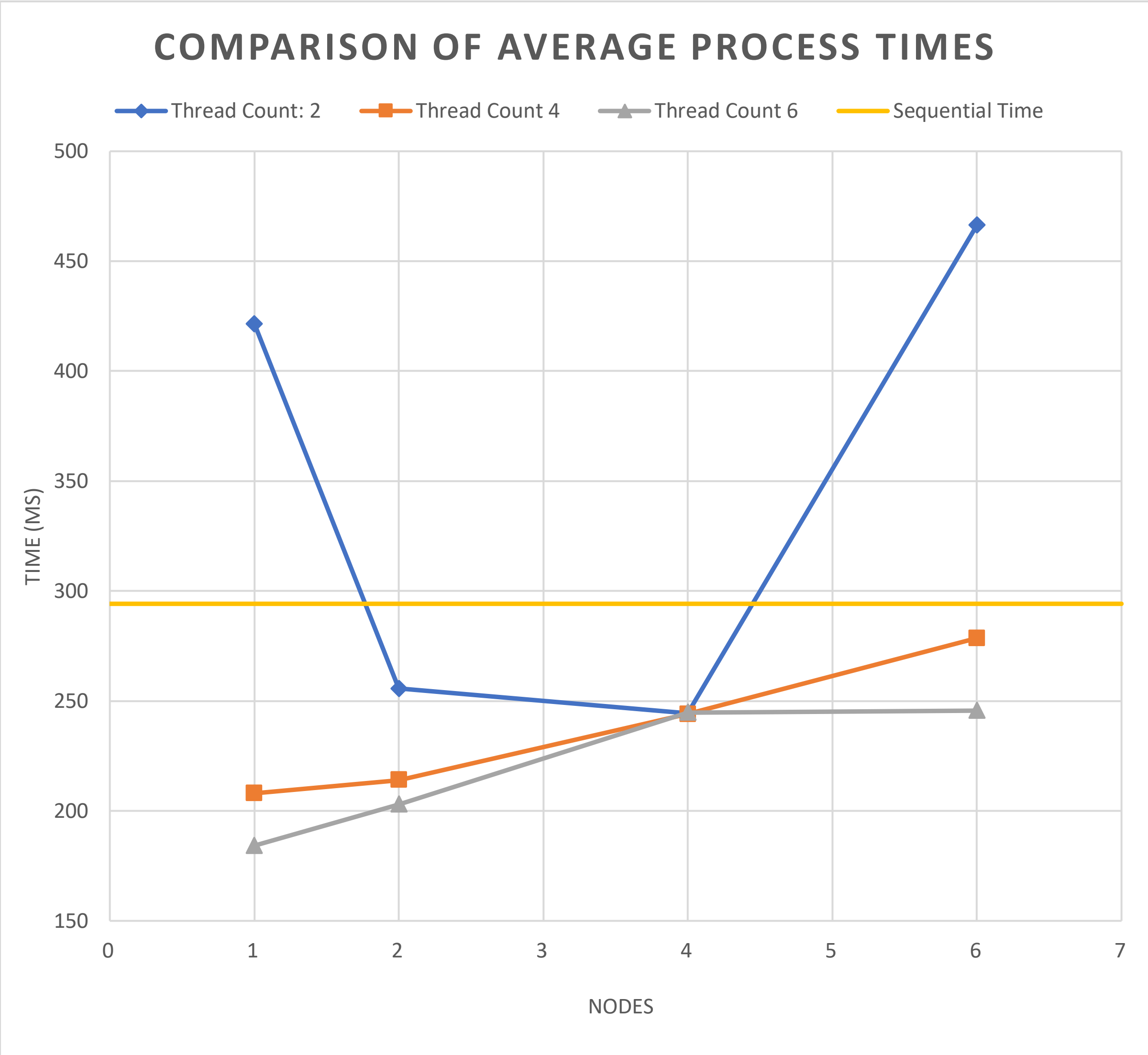

Figure 2. Graphical Representation of Runtime

4.3 Results Discussion

Our results show some expected improvements to computational times as well as some results that are unusual. For the thread counts of 4 and 6, the times were on average faster than the sequential for all tests performed, however there is a noticeable trend for both to have their times increasing as the number of nodes is increased. This disparity is at its greatest at the lower node counts, for one node and two. This phenomenon could be caused by the overhead computational time taken for the nodes to communicated between the master and the workers. While the matrix itself may seem significant in size, with an algorithm that only deals with the market matrix format the calculations may not be as robust as what the MPI paradigm was designed for.

Another interesting result is that both the fastest and slowest computational times were found while using 6 nodes. This could again be a result of the overhead of data transmission between the nodes, with the slowest time of 587ms during the use of only 2 threads, while the fastest time of 163ms was during the use of 6 threads.

The use of 2 threads created results that were not expected. The large jumps in times show improvements over the sequential average for node counts two and four, and slower computational times for 1 node and 6 nodes. This may be due to the problem of false sharing. In some cases, threads degrade the systems performance unwillingly when we modify the independent variable.

5. Conclusion

Overall the use of OpenMP for parallel analysis clearly shows an improvement over the sequential times, however the overhead of MPI sending times appears to have a detrimental effect on the computation times. It should be noted however that the fastest time was found using 6 nodes and 6 threads, so it may be possible that the workload of the node network during the experiment time affects the speed of the computations.

6. Appendix

Appendix A

Sequential Matrix Multiplication C Program

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*
    Matrix Multiplication of Two Sparse Matrix Input Files.

    Student Name/ID: Varun Jain 21963986
    Student Name/ID: Kieron Ho 20500057
 */

//------------------------------Global Variables------------------------------

int* row1;                          //row1 - stores all the column 1 index values for the row column from file input 1
int* column1;                       //column1 - stores all the column 2 index values from file input 1
float* value1;                      //row1 - stores all the column 3 values for the specified (row1, column1) from file input 1

int* row2;                          //row2 - stores all the column 1 index values for the row column from file input 2
int* column2;                       //column2 - stores all the column 2 index values from file input 2
float* value2;                      //row2 - stores all the column 3 values for the specified (row2, column2) from file input 2

int rowOutput[10000];               //rowOutput - store final unique row index value
int columnOutput[10000];            //columnOutput - store final unique column index value
float valueOutput[10000];           //valueOutput - store final value afer computation

int file_one_number_of_lines;       //stores the total number of lines in text file 1
int file_two_number_of_lines;       //stores the total number of lines in text file 2



//------------------------------Final Matrix Computation------------------------------


void compute_matrix_multiplication(char filename[])
{
    FILE *fp = fopen(filename, "w+");

    int total_number_resultant_columns = sizeof(rowOutput)/sizeof(rowOutput[0]);
    //used nested loops in order to find the duplicate values in the 1D arrays
    //iterate through one dimensional array starting at index 0
    for(int output_index = 0; output_index < total_number_resultant_columns; output_index++)
```

```
    {
        //iterate through one dimensional array starting at index 1
        for(int resultant_index = output_index+1; resultant_index < total_number_resultant_columns; resultant_index++)
        {
            //checks whether the ith position of the rowOutput and columnOutput is the same
            //as the jth position of the rowOutput and columnOutput.
            if(rowOutput[output_index] == rowOutput[resultant_index] && columnOutput[output_index] == columnOutput[resultant_index])
            {
                //if the row and column indexs match then add the values of the duplicate row and column indexes
                valueOutput[output_index] = valueOutput[output_index]+valueOutput[resultant_index];
                //replace all duplicate values for rowOutput, columnOutput and valueOutput with 0's
                //ensures there are only unique sets of row and column index pairs
                rowOutput[resultant_index] = 0;
                columnOutput[resultant_index] = 0;
                valueOutput[resultant_index] = 0;
            }
        }
        //results print out on terminal and stores the data in matrixmultiplication.txt in the form of a triplet
        //output should look like - rowOutput columnOutput valueOutput
        //returns only the non zero and unique values.
        if(rowOutput[output_index] != 0 && columnOutput[output_index] != 0 && valueOutput[output_index] != 0)
        {
            printf("%d %d %f\n", rowOutput[output_index], columnOutput[output_index], valueOutput[output_index]);
            fprintf(fp, "%d %d %f\n", rowOutput[output_index], columnOutput[output_index], valueOutput[output_index]);
        }
    }
}

//-------------------------------Matrix Multiplication-------------------------------

void multiply()
{
    int resultantIndex = 0;
    //iterate through file input 1 matrix
    for(int matOneIndex = 0; matOneIndex < file_one_number_of_lines; matOneIndex++)
    {
        //iterate through file input 2 matrix
        for(int matTwoIndex = 0; matTwoIndex < file_two_number_of_lines; matTwoIndex++)
        {
            float sum_of_the_values = 0;
```

```c
            //check whether the column index for file input 1 is the
            //same as the row index for file input 2
            if(column1[matOneIndex] == row2[matTwoIndex])
            {
                //if true, multiply the values for those column and row index values.
                sum_of_the_values += value1[matOneIndex]*value2[matTwoIndex];
            }
            //checks whether sum_of_the_values is a non-zero
            if(sum_of_the_values != 0)
            {
                //store the row index from file input 1
                rowOutput[resultantIndex] =  row1[matOneIndex];
                //store the column index from file input 2
                columnOutput[resultantIndex] = column2[matTwoIndex];
                //store non-zero value in valueOuput
                valueOutput[resultantIndex] = sum_of_the_values;
                resultantIndex++;
            }
        }

    }
}

//------------------------------Maximum Value------------------------------

int maximumValue(int length, int* row)
{
    int max = row[0];
    for(int i = 1; i < length; i++)
    {
        if(row[i] > max)
        {
            max = row[i];
        }
    }
    return max;
}

//------------------------------Total Number of Line------------------------------

/*
    input_file_number_of_lines() function returns the number of lines in a file
*/
int input_file_number_of_lines(char filename[])
{
    //intalise line counter
    int linecount = 0;
    char character;

    FILE *fp = fopen(filename, "r");
```

```c

    while((character = fgetc(fp)) != EOF)
    {
        if(character == '\n')   linecount++;
    }

    fclose(fp);
    //linecount will store the number o files in a a file
    return linecount;
}

//--------------------------------Readfile-----------------------------------

void readfile(char filename[], int *row, int *column, float *value)
{
    char line[BUFSIZ];

    FILE *fp = fopen(filename, "r");

    int i = 0;

    while(fgets(line, sizeof line, fp) != NULL)
    {
        //stores column 1 - the row indexes
        row[i] = atoi(strtok(line, " "));
        //stores column 2 - the column indexes
        column[i] = atoi(strtok(NULL, " "));
        //stores column 3 - the value of the row and column index
        value[i] = atof(strtok(NULL, " "));
        i++;
    }
    fclose(fp);
}


//--------------------sparse file 1 input-----------------------------

void allocate_space_file1(char filename[])
{
    //returns the number of lines in file input 1
    file_one_number_of_lines = input_file_number_of_lines(filename);
    //creates dynamic memory for row1, column1 and value1 array
    row1 = (int*)malloc(file_one_number_of_lines*sizeof(int));
    column1 = (int*)malloc(file_one_number_of_lines*sizeof(int));
    value1 = (float*)malloc(file_one_number_of_lines*sizeof(float));
    //call readfile() to store values in row1, column1, value1 array
    readfile(filename, row1, column1, value1);
}
```

```c
//----------------------sparse file 2 input----------------------

void allocate_space_file2(char filename[])
{
    //return the total number of lines in file input 2
    file_two_number_of_lines = input_file_number_of_lines(filename);
    //creates dynamic memory for row1, column1 and value1 array
    row2 = (int*)malloc(file_two_number_of_lines*sizeof(int));
    column2 = (int*)malloc(file_two_number_of_lines*sizeof(int));
    value2 = (float*)malloc(file_two_number_of_lines*sizeof(float));
    //call readfile() to store values in row2, column2, value2 array
    readfile(filename, row2, column2, value2);
}

//----------------------Main() Executation----------------------

int main(int argc, char* argv[])
{
    //fileinput one
    allocate_space_file1(argv[1]);
    //fileinput two
    allocate_space_file2(argv[2]);
    char matrixMultiplicationComputation[] = "matrixmultiplication.txt";
    multiply();
    compute_matrix_multiplication(matrixMultiplicationComputation);
    return 0;
}
```

Appendix B

openMP + MPI Multiplication C Program

```c
#include "mpi.h"
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

/*
    Matrix Multiplication of Two Sparse Matrix Input Files.

    Student Name/ID: Varun Jain 21963986
    Student Name/ID: Kieron Ho 20500057
 */

#define MASTER 0
#define FROM_MASTER 1
#define FROM_WORKER 2
#define TEMP_ARRAY_SIZE 10000
#define FINAL_ARRAY_SIZE 10000

int row1[TEMP_ARRAY_SIZE];                //row1 - stores all the column 1 index values for the row column from file input 1
int column1[TEMP_ARRAY_SIZE];             //column1 - stores all the column 2 index values from file input 1
float value1[TEMP_ARRAY_SIZE];            //row1 - stores all the column 3 values for the specified (row1, column1) from file input 1

int row2[TEMP_ARRAY_SIZE];                //row2 - stores all the column 1 index values for the row column from file input 2
int column2[TEMP_ARRAY_SIZE];             //column2 - stores all the column 2 index values from file input 2
float value2[TEMP_ARRAY_SIZE];            //row2 - stores all the column 3 values for the specified (row2, column2) from file input 2

int rowOutput[TEMP_ARRAY_SIZE];      //rowOutput - stores the temporary rows received from workers
int columnOutput[TEMP_ARRAY_SIZE]; //columnOutput - stores the temporary columns received from workers
float valueOutput[TEMP_ARRAY_SIZE];//valueOutput - stores the temporary values received from workers

int file_one_number_of_lines;
int file_two_number_of_lines;
int resultSize;

void printMatrix(int row[], int column[], float values[], int elementsInMatrix){//Working
int i;
for(i = 0 ; i < elementsInMatrix ; i++){
        if(values[i] != 0){
    printf("%d %d %.2f\n", row[i], column[i], values[i]);
        }
    }
}
```

```
int maximumValue(int length, int* row)
{
    int max = row[0];
    for(int i = 1; i < length; i++)
    {
        if(row[i] > max)
        {
            max = row[i];
        }
    }
    return max;
}

/*
    input_file_number_of_lines() function returns the number of
    lines in a file
*/
int input_file_number_of_lines(char filename[])
{
    int linecount = 0;
    char character;

    FILE *fp = fopen(filename, "r");

    while((character = fgetc(fp)) != EOF)
    {
        if(character == '\n')    linecount++;
    }

    fclose(fp);

    return linecount;
}


void readfile(char filename[], int *row, int *column, float *value)
{
    char line[BUFSIZ];

    FILE *fp = fopen(filename, "r");

    int i = 0;

    while(fgets(line, sizeof line, fp) != NULL)
    {
        row[i] = atoi(strtok(line, " "));
        column[i] = atoi(strtok(NULL, " "));
        value[i] = atof(strtok(NULL, " "));
        i++;
    }
    fclose(fp);
}
```

```
void allocate_space_file1(char filename[])
{
    file_one_number_of_lines = input_file_number_of_lines(filename);
    readfile(filename, row1, column1, value1);
}

void allocate_space_file2(char filename[])
{
    file_two_number_of_lines = input_file_number_of_lines(filename);
    readfile(filename, row2, column2, value2);
}


int main(int argc, char* argv[])
{
    //indices replaces rows, aveindex replaces averow
    int numtasks, taskid, numworkers, source, dest, mtype, indices, aveindex,
    extra, offset, i, j, k, rc, threadsProvided;
    int startTime = clock()*1000/CLOCKS_PER_SEC;

    if(argc > 3)
    {
        int threads = atoi(argv[3]);
        omp_set_num_threads(threads);
    }else omp_set_num_threads(1);
    MPI_Status status;
    MPI_Init_thread(&argc,&argv, MPI_THREAD_FUNNELED, &threadsProvided);
    MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
    MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
    if(numtasks < 2){
        printf("Need at least two MPI tasks. Quitting...\n");
        MPI_Abort(MPI_COMM_WORLD, rc);
        exit(1);
    }
    numworkers = numtasks-1;
    /**************************** master task ********************************/
    if(taskid == MASTER)
    {
        allocate_space_file1(argv[1]);
        allocate_space_file2(argv[2]);
        aveindex = file_one_number_of_lines/numworkers;
        extra = file_one_number_of_lines%numworkers;
        offset = 0;
        mtype = FROM_MASTER;

    /*Should initialise master-only data structures here*/

        int rowFinal[FINAL_ARRAY_SIZE];
        int columnFinal[FINAL_ARRAY_SIZE];
        float valueFinal[FINAL_ARRAY_SIZE];

        for(dest=1; dest<=numworkers; dest++)
```

```
        {
            indices = (dest <= extra) ? aveindex+1 : aveindex;//number of tuples to send from matrix
            MPI_Send(&indices, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);//the number of tuples
            MPI_Send(&file_two_number_of_lines, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);//send the amount of tuples in the second matrix

            //send the trimmed matrix A data
            MPI_Send(&row1[offset], indices, MPI_INT, dest, mtype, MPI_COMM_WORLD);//trimmed matrix one rows
            MPI_Send(&column1[offset], indices, MPI_INT, dest, mtype, MPI_COMM_WORLD);//trimmed matrix one columns
            MPI_Send(&value1[offset], indices, MPI_FLOAT, dest, mtype, MPI_COMM_WORLD);//trimmed matrix one value

            //send the matrix B data
            MPI_Send(&row2, file_two_number_of_lines, MPI_INT, dest, mtype, MPI_COMM_WORLD);//all matrix two rows
            MPI_Send(&column2, file_two_number_of_lines, MPI_INT, dest, mtype, MPI_COMM_WORLD);//all matrix two columns
            MPI_Send(&value2, file_two_number_of_lines, MPI_FLOAT, dest, mtype, MPI_COMM_WORLD);//all matrix two value
        //prepare the next message
            offset = offset + indices;
        }
        /*Receive results from worker tasks*/
        int finalArrayDataSize = 0;
        mtype = FROM_WORKER;
        for(i = 1 ; i <= numworkers; i++)
        {
            source = i;
            MPI_Recv(&resultSize, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
            MPI_Recv(&rowOutput, TEMP_ARRAY_SIZE, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);//resultant row
            MPI_Recv(&columnOutput, TEMP_ARRAY_SIZE, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);//resultant column
            MPI_Recv(&valueOutput, TEMP_ARRAY_SIZE, MPI_FLOAT, source, mtype, MPI_COMM_WORLD, &status);//resultant values

            /*Process new data here. designed to add to the existing results arrays marked "final"*/
            //needs to be run sequentially
            if(resultSize>0){
                for(j = 0 ; j < resultSize ; j++){//for each received output tuple, non zeroes
                        for(k = 0 ; k < FINAL_ARRAY_SIZE ; k++){//go through the final tuples
                if(rowOutput[j] == rowFinal[k] && columnOutput[j] == columnFinal[k]){//if element result exists, update
                    valueFinal[k] += valueOutput[j];//the added multiplication value
                    break;//if you have found a place to put it, move along
                }
```

```
                else if(columnFinal[k] <= 0 && rowFinal[k] <= 0 && valueFinal[k] == 0){//if element didn't exist, add it(provided it isn't 0, 0)
                    rowFinal[k] = rowOutput[j];
                    columnFinal[k] = columnOutput[j];
                    valueFinal[k] = valueOutput[j];
                    finalArrayDataSize++;
                    break;
                    }
                }
            }
        }
    }
        /*Print results here*/
        printMatrix(rowFinal, columnFinal, valueFinal, finalArrayDataSize);
        int endTime = clock()*1000/CLOCKS_PER_SEC;
        printf("Time taken: %dms\n", endTime - startTime);
    }

    /************************ Worker Tasks ****************************************/
    if(taskid > MASTER)
    {
        mtype = FROM_MASTER;
        //Receive the parameters
        MPI_Recv(&indices, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
        MPI_Recv(&file_two_number_of_lines, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
        //Receive the data for matrix one trimmed
        MPI_Recv(&row1, indices, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
        MPI_Recv(&column1, indices, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
        MPI_Recv(&value1, indices, MPI_FLOAT, MASTER, mtype, MPI_COMM_WORLD, &status);

        //Receive all of data for matrix two
        MPI_Recv(&row2, file_two_number_of_lines, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
        MPI_Recv(&column2, file_two_number_of_lines, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
        MPI_Recv(&value2, file_two_number_of_lines, MPI_FLOAT, MASTER, mtype, MPI_COMM_WORLD, &status);

        int resultantIndex = 0;
        int matTwoIndex;
        file_one_number_of_lines = indices;

        #pragma omp parallel for private(matTwoIndex)
        for(int matOneIndex = 0; matOneIndex < file_one_number_of_lines; matOneIndex++)//for each line in file one,
        {
        for(matTwoIndex = 0; matTwoIndex < file_two_number_of_lines; matTwoIndex++)//for each line in file two,
```

```
                {
                    float sum_of_the_values = 0;
                    if(column1[matOneIndex] == row2[matTwoIndex])
                    {
                        sum_of_the_values += value1[matOneIndex]*value2[matTwoIndex];
                    }
                    if(sum_of_the_values != 0)
                    {
                        rowOutput[resultantIndex] =  row1[matOneIndex];
                        columnOutput[resultantIndex] = column2[matTwoIndex];
                        valueOutput[resultantIndex] = sum_of_the_values;
                        resultantIndex++;//what does this do? How to mimic this in paralel?
                    }
                }
            }
            int total_number_resultant_columns = sizeof(rowOutput)/sizeof(rowOutput[0]);
            resultSize = total_number_resultant_columns;

            for(int output_index = 0; output_index < total_number_resultant_columns; output_index++)
            {
                for(int resultant_index = output_index+1; resultant_index < total_number_resultant_columns; resultant_index++)
                {
                    if(rowOutput[output_index] == rowOutput[resultant_index] && columnOutput[output_index] == columnOutput[resultant_index])
                    {
                        valueOutput[output_index] = valueOutput[output_index]+valueOutput[resultant_index];
                        rowOutput[resultant_index] = 0;
                        columnOutput[resultant_index] = 0;
                        valueOutput[resultant_index] = 0;
                    }
                }
            }
            if(indices>0)
            {
                mtype = FROM_WORKER;
                MPI_Send(&resultSize, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
                MPI_Send(&rowOutput, TEMP_ARRAY_SIZE, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
                MPI_Send(&columnOutput, TEMP_ARRAY_SIZE, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
                MPI_Send(&valueOutput, TEMP_ARRAY_SIZE, MPI_FLOAT, MASTER, mtype, MPI_COMM_WORLD);
            }
        MPI_Finalize();
        return 0;
}
```