CMPE 300
Analysis of Algorithms Programming Project

Line Detection with Parallel Processing
Project Report

Student Name: Harun Zengin

Student Number: 2012400081

Submitted to: Tunga Güngör, Metehan Doyran (TA)

Submission Date: 25 December 2016

1) Introduction

Parallel processing is a method of computation which the program divides the problem into subproblems which are solved then after by different CPU’s. The main advantage compared to sequential processing is the runtime efficiency. Since different processes solve different problems at the same time, ideally, the runtime will be divided by the number of processors.

Image processing may be an expensive process for large images since calculating with for instance a 2D array with 5000 x 5000 integers may have very long running times. We aim to solve this problem by running the image processing algorithms in parallel processes. To divide the algorithms, we use the MPI environment and implement our program in the C language.

Our Program takes a gray-scale square image written on a text file as input and produces a line-detected output. The output image is black-white, the pixels which were detected as a middle of a line, namely the lines are denoted as white and the other pixels as black. The processing is done via parallel processing and by this way, decreases the efficiency considerably.

2) Program Interface

To run the program, the user needs to install a C compiler and then the MPI environment in order to be able to compile and execute the program.

The program has to be compiled via the following command before running it:

```
mpicc -g main.c -o main
```

Then after it can be run via the command:

```
mpiexec -n <number of processes> ./main <input.txt>  <output.txt>
```

Where ,

<number of processes> is an integer denoting the number of processes to be run with. The process number must be a divider without remainder of the number of pixels given in the input image.

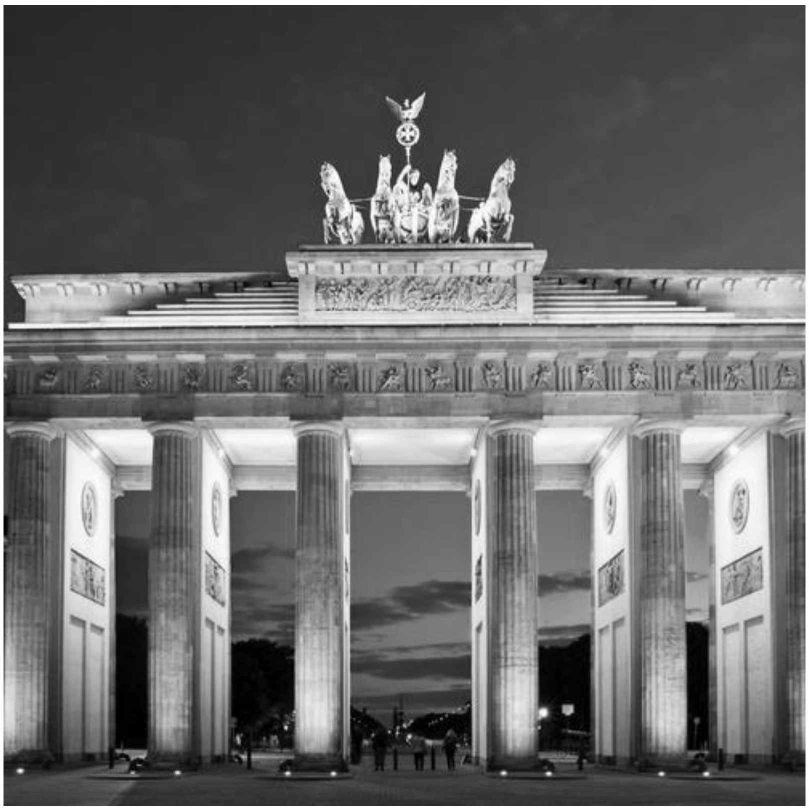
<input.txt> is the input file name of the file.

<output.txt> is the output file name the user wishes to enter.

3) Program Execution

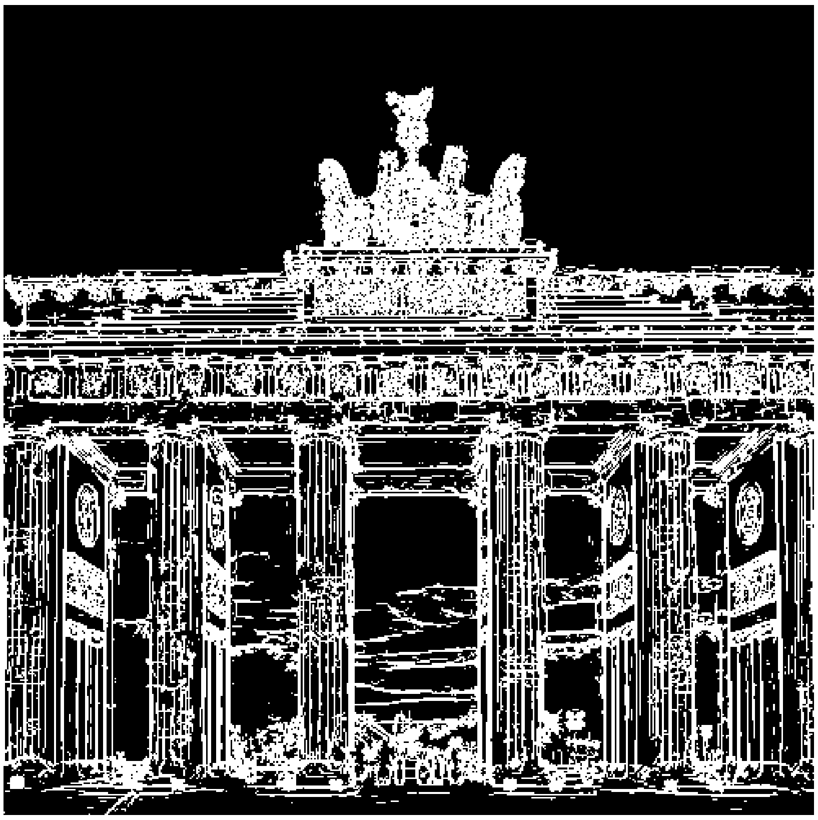
The program produces an image array as output. Its actual functionality is to detect the lines in the given input image and produce an output where it produces a black pixel for not detecting a line and a white pixel for detecting a line. For example consider the beautiful image of the Brandenburg Gate below:

This image needs first converted to a gray-scale image by the visualize.py script.

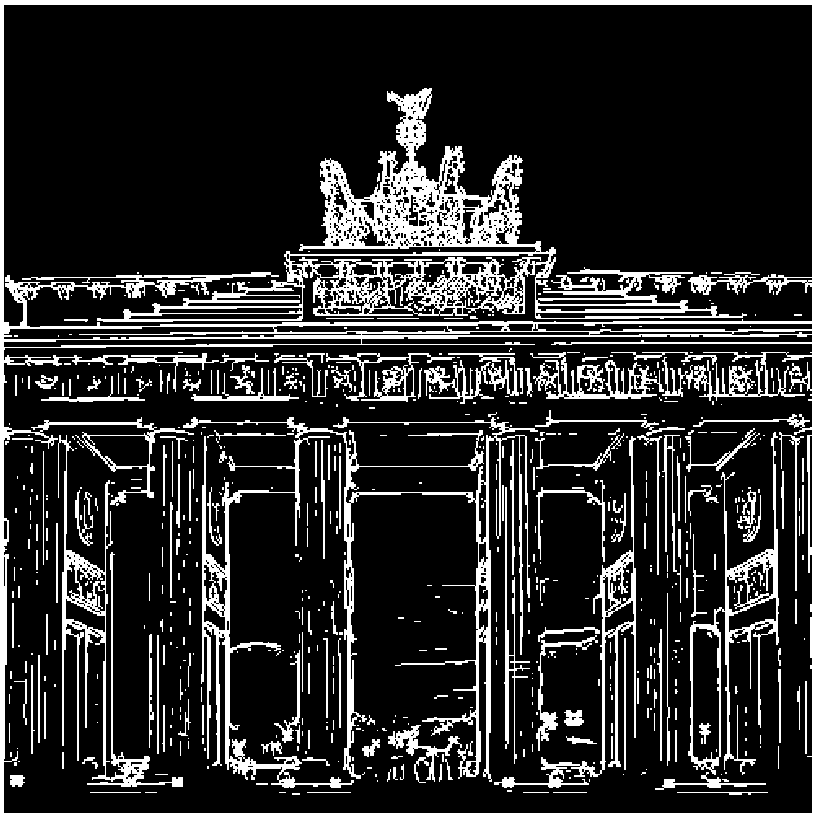


The converted grayscale image will look like this.

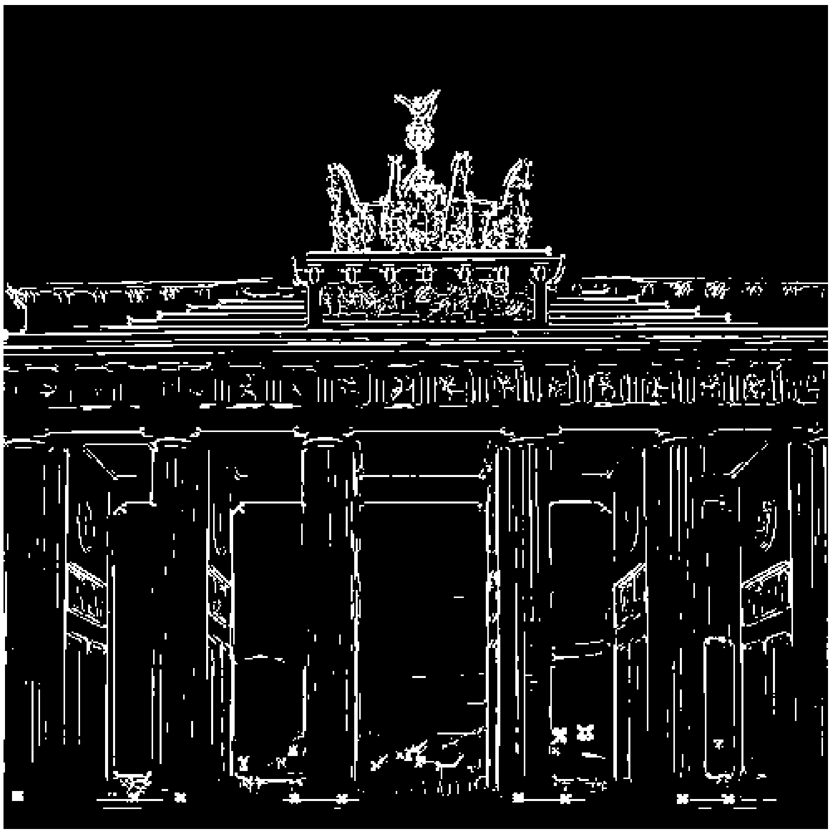
When we run our program given as input the image above, the program will detect the lines and convert the detected lines to white and the other pixels to black. It can do it with different thresholds calculated.



The output for threshold 10 will look like this. The threshold is simpler said a filter value, which determines the sensitivity of the lines. More simpler said, it is a value which decides to what is considered a line or not. it detects even smoother lines such as the clouds and details of the horses as a line.



The output set threshold to 25 will look like this. The details of the horses can be seen easier. The clouds have partly vanished and parts of the columns have vanished, because they resemble their surroundings more.



The output for threshold 40 will look like this. The horses are even better to distinguish but the columns are hardly recognized. This is because our program recognizes the pixel parts even harder as a line than threshold 25.

4) Input and Output

Our programs inputs and outputs are, as stated in Program Interface part, determined in the command line. Our input must be a text file consisting of a square array of integers, values 0 to 255. To skip a row in the array, a space character must be included between the two integers, and to skip a column, a newline character (/n) must be included. The input file must be a 200x200 array, although, the int pixel variable can be changed in order to adjust to a different square input.

The output format is similar to the input format, with the slightest change of the array size. The output arrays dimensions will decrease by four, for instance for a 200x200 input, we will have an output of 196x196 dimensioned 2D array. This decrease is due to boundary conditions of the smoothing and line-detection algorithms which will explained in the Program Structure part.

The inputs and outputs can be visualized by the provided visualize.py python script.

5) Program Structure

5.1) Reading The File and Scattering

The program is implemented in the MPI (Message Passing Interface) environment and is implemented using the C language. It firstly initializes the MPI environment via the MPI_Init function.

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
int size;
MPI_Comm_size(MPI_COMM_WORLD, &size);

int pixel = 200;    // can be changed for different images
int threshold = 10;  //the threshold value
int row_per_process = pixel/(size-1);
int mainImage[pixel + row_per_process][pixel];
int calculatedImage[pixel + row_per_process][pixel-4];
```

The code section above shows the fundamental definitions in the program. The rank and the size variables are the fundamental variables of the MPI environment. The pixel and threshold values here can be set for different values. The row_per_process variable is used quite frequently and denotes the pixel size divided by the number of slave processors, which will explained below. The mainImage array is the initial array which will be read from the input file and included into this array. The calculatedImage array is the final array, which will be at the end of the code, written to an output file. The mainImage and calculatedImage array sizes may be seem strange at first, but its logic is that we have one master and n slave processors. The MPI_Scatter function below will divide the array equally to ALL processors, including the master processor. In order not to lose any data by giving to the master processor, since the master processor will not do any line detection calculations, we fill the part of the mainImage array corresponding to the master processor with zeroes. The scattering section of the code is shown below.

```
int subImage [row_per_process][pixel];
//send (200/size-1) * 200 elements to each process
MPI_Scatter(mainImage,row_per_process*pixel, MPI_INT,
//receive same size in subImage
            subImage, row_per_process*pixel, MPI_INT,
            0,MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);
```

The subImage array is defined above for each processor, so that the each processor gets its own part of the input image. The division of the main image is illustrated below for 4 slave and one master processor. The subImage arrays are illustrated below for a sample 8x8 input.

0	0	0	0	0	0	0	0	Master Processor
0	0	0	0	0	0	0	0	Rank 0
15	155	155	131	56	2	15	123	Slave Processor
17	18	12	15	154	123	147	98	Rank 1
12	54	87	6	56	45	12	123	Slave Processor
54	78	79	78	45	65	32	65	Rank 2
155	155	48	78	45	32	125	15	Slave Processor
48	42	35	45	98	99	100	101	Rank 3
17	50	55	55	54	53	51	53	Slave Processor
152	111	121	112	111	110	154	148	Rank 4

5.2) Smoothing and Sending-Receiving

After scattering the arrays, what we firstly need to do is to smooth the image parts in order not to recognize everything in the image as a line. This is done via a simple convolution process. This is also illustrated below:

15	155	155
17	18	12
12	54	87

The smoothed value is the convolution of the 3x3 array part, and its value is calculated by averaging the 9 elements. So the smoothed value of this array part's centre denoted in red would be 58. Since only the middle elements can be calculated, we lose 2 columns and 2 processes.

During this process, a new problem arises. Considering the 3x3 array part above, the green elements of the array are members of the processor with rank 2, but the rest and center is in rank 1! So to calculate the smoothed value, we need the upper part of the array of process 2 and send it to process 1. How we implement this is shown below:

```
    int send_upper[pixel] ;      //upper part of the current subImage to be sent
    int send_lower[pixel] ;      //lower part of the current subImage to be sent
    for (i=0;i<pixel;i++){
        send_upper[i] = subImage[0][i];
        int lower = row_per_process-1;
        send_lower[i] = subImage[lower][i];
    }
    int receive_upper[pixel];               //upper part of lower rank to be received
    int receive_lower[pixel];               //lower part of upper rank to be received
    int smooIma [row_per_process][pixel-2];    //the smoothed subImage

    if (rank !=0 && rank != size -1){
        MPI_Send(
            &send_lower,        //data pointer
            pixel,                  //count
            MPI_INT,            //type
            rank+1,                  //receiver
            0,                  //tag
            MPI_COMM_WORLD);
    }
```

Every process sends its upper array and lower array to the upper and lower process with the MPI_Send function. And every process receives the necessary arrays by the code section below. The smooImage array is the one that holds the integer values after the smoothing process. This array is defined in every process.

```
if (rank >1){
  MPI_Recv(&receive_upper,pixel,MPI_INT,rank-1,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
}
if (rank >1){
    MPI_Send(&send_upper,pixel,MPI_INT,rank-1,1,MPI_COMM_WORLD);
}
if (rank !=0 && rank != size -1){

MPI_Recv(&receive_lower,pixel,MPI_INT,rank+1,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
}
```

The smoothing process carefully detects which array misses and needs to be evaluated from an upper or lower row missing. Even if every processor has only one image, the algorithm will calculate it from the upper and lower processes.

5.3) Line Detection

Before we start the line detection, the lower and upper rows need to be sent again, since the smoothed values of edge rows do not exist in the neighbor processes.

The detection of lines is done by convoluting 3x3 matrices with 3x3 matrices. An illustration for a horizontal line is shown below. Every element gets multiplied with the element in the same position.

87	6	56
79	78	45
48	78	45

X

-1	-1	-1
2	2	2
-1	-1	-1

After convolution, the value would be 86

The convolution process is done for 4 different line aligning possibilities. The horizontal one is shown above, the other 3 are:

-1	2	-1
-1	2	-1
-1	2	-1

-1	-1	2
-1	2	-1
2	-1	-1

2	-1	-1
-1	2	-1
-1	-1	2

Our threshold value defined in 5.1 is compared with every output of these convolutions. If any one of the outputs is greater than or equal the threshold, we assume that we have detected a line and write a 155 to the corresponding line_detected array that we defined. Again, we lose two rows and two columns here.

Then, we invoke the MPI_Gather function which is the inverse of the scatter function which collects our line_detected subarrays to the master processor, by collecting them to the calculatedImage array.

Then, we print the array to the output file.

6) Examples

To visualizing the explanations above, we will give a sample 6x6 image, and then run the algorithm on it with two processors. Given the input below,

55	58	69	40	35	45
78	87	45	46	48	35
48	65	154	78	88	55
68	84	121	45	35	45
72	76	78	41	38	15
78	77	79	45	37	12

Since we have 2 processors, we need 3 rows of zeroes sent to processor 0, the master processor.

0	0	0	0	0	0	Master 0
0	0	0	0	0	0	
0	0	0	0	0	0	
55	58	69	40	35	45	Slave 1
78	87	45	46	48	35	
48	65	154	78	88	55	
68	84	121	45	35	45	Slave 2
72	76	78	41	38	15	
78	77	79	45	37	12	

The subImage array of Slave 1 is shown with blue, The send_Lower Image is shown with green border, the receive_Lower is shown with yellow.

55	58	69	40	35	45
78	87	45	46	48	35
48	65	154	78	88	55
68	84	121	45	35	45

The smooIma array of processor 1 would then be

73	71	67	52
83	80	73	52

The overall array would then be:

73	71	67	52
83	81	73	53
85	82	75	49
81	72	58	35

After convoluting it with the horizontal, vertical, and / and \ matrices, we will get these matrices respectively. Again, before doing these, we need to send and receive the edge rows of the array.

20	16
38	42

11	43
13	41

6	20
21	22

-4	-23
-21	-6

If applied a threshold of 25, we will get the following output image.

0	155
155	155

7) Improvements and Extensions

Although our program can create a lot of processes, the ideal process number is the one close to the real physical processor number. The running time may be still faster for process numbers slightly higher than the physical processor numbers due to multiprogramming in computers, but the increase of the process number does not always yield to shorter running times. So one improvement in the program would be to advice the user to enter a process number slightly higher of her machines' physically processor numbers.

There are too many barriers in the program, which means a process waits for other processes to finish. This could result for a longer running time if the Process Scheduler of the computer has too many processes waiting in the process queue.

Also, a GUI can be designed for simplicity, and more user friendliness.

8) Difficulties Encountered

I was not very familiar with the C language before starting this project. So I lost a lot of time trying to allocate dynamic arrays with the malloc function. But the problem here was that the MPI scatter function could not scatter 2D dynamic array because these arrays were pointers within pointers and MPI could not follow the pointer value for another process. I guess every process is allocated for a different memory, so when I used MPI Scatter, it would give me unreasonable arrays.

Another problem I encountered was to distinguish between the master process and the other processes. The MPI Scatter function did scatter a part also to the master process which I did not want. So I lost a lot of time doing researched about how I could resolve this but the best and not the smartest solution was to fill the part of the master process' rows with zeroes.

9) Conclusion

To conclude, Image processing can be speeded up with parallel processing and it is a smart idea to divide the parts of the images to different processes in order to do the work for each part at the same time. The line detection is also a simple and smart idea, to use convolution and different thresholds to detect a line is the implementation of science to arts. I had a lot of fun doing the project, and it did not bother me to write 10 pages of report at all.

10) References

Experimental and Theoretical Speedup Prediction of MPI-Based Applications, Alaa Elnashar and Sultan Aljahdali

http://www.doiserbia.nb.rs/img/doi/1820-0214/2013/1820-02141300047E.pdf

Introduction to Parallel Computing Slides, Frank Willmore

https://portal.tacc.utexas.edu/c/document_library/get_file?uuid=e05d457a-0fbf-424b-87ce-c96fc0077099&groupId=13601

11) Appendix

```
/* Student Name: Harun Zengin
* Student Number: 2012400081
* Compile Status: Compiling
* Program Status: Working
* Notes: The program works as wanted in the project description.
*        But the arrays are not dynamically allocated because for a 2D array that doesn't
work.
*        There are code sections which are commented out. I didn't delete them to show you
my tries and testing behaviour.
*        The int calculated variable is only created for testing issues.
*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char* argv[])
{
    if(argc != 3){
        printf("Arguments not supplied!");
        return 0;
    }
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);
    // Find out rank, size
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int pixel = 200;    // can be changed for different images
    int threshold = 10;  //the threshold value
    int row_per_process = pixel/(size-1);
    int mainImage[pixel + row_per_process][pixel];
    int calculatedImage[pixel + row_per_process][pixel-4];
    int i,j;                                                //for all 2D array iterations
    /* Dynamically allocation doesn2t work
    for (i=0; i<pixel + (row_per_process; i++)
         mainImage[i] = (int *)malloc(pixel * sizeof(int));*/

    if (rank == 0) {
        FILE *myFile;
        myFile = fopen(argv[1], "r");

        //read file into array mainImage

        for (i = 0; i < row_per_process; i++){
            for (j = 0; j < pixel; j++){
                mainImage[i][j] = 0;
            }
        }

        for (i = row_per_process; i < pixel + row_per_process; i++){
            for(j = 0; j<pixel; j++){
                fscanf(myFile, "%d", &mainImage[i][j]);
            }
        }
        /*
        //Only to print if current file is read rigth or not!
        FILE *fa = fopen("input read","w");

           for (i = 0; i < pixel + row_per_process; i++){
                for(j = 0; j<pixel; j++){
                    fprintf(fa, "%d ", mainImage[i][j]);
                }
                fprintf(fa, "\n" );
            }
        fclose(fa);
        */

    }
    int subImage [row_per_process][pixel];
       /*Commented out because dynamic array doesnt work for 2D Array
        for (i=0; i< row_per_process; i++)
            subImage[i] = (int *)malloc(pixel * sizeof(int));*/


    MPI_Scatter(mainImage,row_per_process*pixel, MPI_INT,     //send (200/size-1) * 200 ele-
ments to each process
                subImage, row_per_process*pixel, MPI_INT,      //receive same size in
subImage
                0, MPI_COMM_WORLD);

    MPI_Barrier(MPI_COMM_WORLD);
/*  Commented out because dynamic array doesnt work for 2D Array
    if(rank != 0){
        for (i=0; i<200 + (200/(size-1)); i++){
            free(mainImage[i]);
        }
    }*/
    int calculated = 0; //for testing
/*
    To send and receive the upper and lower parts of the array, these arrays are defined in
each process and
    are maintaining the synchronization

*/

    int send_upper[pixel] ;                 //upper part of the current subImage to be sent
    int send_lower[pixel] ;                 //lower part of the current subImage to be sent
    for (i=0;i<pixel;i++){
        send_upper[i] = subImage[0][i];
        int lower = row_per_process-1;
        send_lower[i] = subImage[lower][i];
    }
    int receive_upper[pixel];               //upper part of lower rank to be received
    int receive_lower[pixel];               //lower part of upper rank to be received
    int smooIma [row_per_process][pixel-2];    //the smoothed subImage

    if (rank !=0 && rank != size -1){
        MPI_Send(
            &send_lower,        //data pointer
            pixel,                  //count
            MPI_INT,            //type
            rank+1,                  //receiver
            0,                  //tag
            MPI_COMM_WORLD);
    }
    if (rank >1){
        MPI_Recv(&receive_upper,pixel,MPI_INT,rank-1,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }
    if (rank >1){
        MPI_Send(&send_upper,pixel,MPI_INT,rank-1,1,MPI_COMM_WORLD);
    }
    if (rank !=0 && rank != size -1){
        MPI_Recv(&receive_lower,pixel,MPI_INT,rank+1,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }
    MPI_Barrier(MPI_COMM_WORLD);
    //enter smoothing part
    if(rank != 0 ){
        for (i=0;i<row_per_process;i++){
            if(!((rank == 1 && i == 0)||(rank == size -1 && i == row_per_process-1))) {
                for (j=1; j<pixel-1; j++){
                    calculated = 16;
                    if(i==0 && pixel ==  size-1){
                        smooIma[i][j-1] = ( receive_upper[j-1] + receive_upper[j] + rece-
ive_upper[j+1] +
                                            subImage[i][j-1] + subImage[i][j] +
subImage[i][j+1] +                  //Smoothes image if every process has one row
                                            receive_lower[j-1]  + receive_lower[j]  + rece-
ive_lower[j+1])/9;
                    } else if(i == 0 && !(rank ==1 && size -1== pixel)){
                        smooIma[i][j-1] = ( receive_upper[j-1] + receive_upper[j] + rece-
ive_upper[j+1] +
                                            subImage[i][j-1] + subImage[i][j] +
subImage[i][j+1] +                  //if the upper array is needed
                                            subImage[i+1][j-1] + subImage[i+1][j] +
subImage[i+1][j+1])/9;
                        calculated =2;
                    } else if(i == row_per_process-1&& !(rank ==size-1 && size-1 == pixel)){
                        smooIma[i][j-1] = ( subImage[i-1][j-1] + subImage[i-1][j] +
subImage[i-1][j+1] +
                                            subImage[i][j-1] + subImage[i][j] +
subImage[i][j+1] +                  //if the lower array is needed
                                            receive_lower[j-1]  + receive_lower[j]  + rece-
ive_lower[j+1])/9;
                    } else {
                        smooIma[i][j-1] = ( subImage[i-1][j-1] + subImage[i-1][j] +
subImage[i-1][j+1] +
                                            subImage[i][j-1] + subImage[i][j] +
subImage[i][j+1] +                  //if everything is present
                                            subImage[i+1][j-1] + subImage[i+1][j] +
subImage[i+1][j+1])/9;
                        calculated = 5;
                    }
                }
            }
        }
    }

    MPI_Barrier(MPI_COMM_WORLD);            //synchronize all processes before sending and
receiving smoothed edges
    /*
        In order to save memory, the sharing of edges is the same as the previous code sec-
tion.
        There is no extra array created to share edges.
    */
    for (i=0;i<pixel-2;i++){
        send_upper[i] = smooIma[0][i];
        int lower = row_per_process-1;
        send_lower[i] = smooIma[lower][i];
    }

    if (rank !=0 && rank != size -1){
        MPI_Send(
            &send_lower,        //data pointer
            pixel,                  //count
            MPI_INT,            //type
            rank+1,                  //receiver
            0,                  //tag
            MPI_COMM_WORLD);
    }
    if (rank >1){
        MPI_Recv(&receive_upper,pixel,MPI_INT,rank-1,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }
    if (rank >1){
        MPI_Send(&send_upper,pixel,MPI_INT,rank-1,1,MPI_COMM_WORLD);
    }
    if (rank !=0 && rank != size -1){
        MPI_Recv(&receive_lower,pixel,MPI_INT,rank+1,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }
    MPI_Barrier(MPI_COMM_WORLD);            //Synchronize before starting line detecting
process
    int line_detected [row_per_process][pixel-4];    //the line detected subImage


    if(rank != 0 ){
        int k;  // we check 4 matrices
        int line_matrix[3][3];  // line matrices to be convoluted with
        int checker;            // computed convolution value
        for(k=0;k<4;k++){
            if(k==0){
                line_matrix[0][0] = -1; line_matrix[0][1] = -1; line_matrix[0][2] = -1;
                line_matrix[1][0] =  2; line_matrix[1][1] =  2; line_matrix[1][2] =  2;
                line_matrix[2][0] = -1; line_matrix[2][1] = -1; line_matrix[2][2] = -1;
            }
            if(k==1){
                line_matrix[0][0] = -1; line_matrix[0][1] =  2; line_matrix[0][2] = -1;
                line_matrix[1][0] = -1; line_matrix[1][1] =  2; line_matrix[1][2] = -1;
                line_matrix[2][0] = -1; line_matrix[2][1] =  2; line_matrix[2][2] = -1;
            }
            if(k==2){
                line_matrix[0][0] =  2; line_matrix[0][1] = -1; line_matrix[0][2] = -1;
                line_matrix[1][0] = -1; line_matrix[1][1] =  2; line_matrix[1][2] = -1;
                line_matrix[2][0] = -1; line_matrix[2][1] = -1; line_matrix[2][2] =  2;
            }
            if(k==3){
                line_matrix[0][0] = -1; line_matrix[0][1] = -1; line_matrix[0][2] =  2;
                line_matrix[1][0] = -1; line_matrix[1][1] =  2; line_matrix[1][2] = -1;
                line_matrix[2][0] =  2; line_matrix[2][1] = -1; line_matrix[2][2] = -1;
            }
            for (i=0;i<row_per_process;i++){
                if(!((rank == 1 && i < 2)||(rank == size -1 && i > row_per_process-3))) {
                    for (j=1; j<pixel-3; j++){
                        calculated = 16;
                        //Detects image if every process has one row
                        if(pixel ==  size-1 && !(rank ==1 || rank ==2 || rank == size-1 ||
rank == size-2)) {
                            if(line_detected[i][j-1] != 255){
                                checker =               receive_upper[j-1] *  line_mat-
rix[0][0] + receive_upper[j]  *  line_matrix[0][1]+ receive_upper[j+1]    *  line_mat-
rix[0][2]+
                                                        smooIma[i][j-1]    *  line_mat-
rix[1][0] + smooIma[i][j]     *  line_matrix[1][1]+ smooIma[i][j+1]       *  line_mat-
rix[1][2]+
                                                        receive_lower[j-1] *  line_mat-
rix[2][0] + receive_lower[j]  *  line_matrix[2][1]+ receive_lower[j+1]    *  line_mat-
rix[2][2];
                                if (checker > threshold){
                                    line_detected[i][j-1] = 255;
                                } else {
                                    line_detected[i][j-1] = 0;
                                }

                            }
                        //if the upper array is needed
                        } else if(i ==0 && !((rank ==1 || rank == 2 )&& size -1== pixel)){
```

```
                        if(line_detected[i][j-1] != 255){
                        checker  =                  receive_upper[j-1] *  line_mat-
rix[0][0]+ receive_upper[j]   *  line_matrix[0][1]+ receive_upper[j+1]    *  line_mat-
rix[0][2]+
                                                    smooIma[i][j-1]    *  line_mat-
rix[1][0]+ smooIma[i][j]      *  line_matrix[1][1]+ smooIma[i][j+1]       *  line_mat-
rix[1][2]+
                                                    smooIma[i+1][j-1]  *  line_mat-
rix[2][0]+ smooIma[i+1][j]    *  line_matrix[2][1]+ smooIma[i+1][j+1]     *  line_mat-
rix[2][2];
                            if (checker > threshold){
                                line_detected[i][j-1] = 255;
                            } else {
                                line_detected[i][j-1] = 0;
                            }
                        }
                    //if the lower array is needed
                    } else if(i == row_per_process -1 && !((rank ==size-1 ||rank == size
-2)&& size-1 == pixel)){
                        if(line_detected[i][j-1] != 255){
                            checker =            smooIma[i-1][j-1]     *  line_mat-
rix[0][0] + smooIma[i-1][j]   *  line_matrix[0][1]+ smooIma[i-1][j+1]     *  line_mat-
rix[0][2]+
                                                 smooIma[i][j-1]       *  line_mat-
rix[1][0]+ smooIma[i][j]      *  line_matrix[1][1]+ smooIma[i][j+1]       *  line_mat-
rix[1][2]+
                                                 receive_lower[j-1]    *  line_mat-
rix[2][0] + receive_lower[j]  *  line_matrix[2][1]+ receive_lower[j+1]    *  line_mat-
rix[2][2];
                            if (checker > threshold){
                                line_detected[i][j-1] = 255;
                            } else {
                                line_detected[i][j-1] = 0;
                            }
                        }
                    //if everything is present
                    } else {
                        if(line_detected[i][j-1] != 255){
                        checker =                   smooIma[i-1][j-1]  *  line_mat-
rix[0][0]+ smooIma[i-1][j]    *  line_matrix[0][1]+ smooIma[i-1][j+1]     *  line_mat-
rix[0][2]+
                                                    smooIma[i][j-1]    *  line_mat-
rix[1][0]+ smooIma[i][j]      *  line_matrix[1][1]+ smooIma[i][j+1]       *  line_mat-
rix[1][2]+
                                                    smooIma[i+1][j-1]  *  line_mat-
rix[2][0]+ smooIma[i+1][j]    *  line_matrix[2][1]+ smooIma[i+1][j+1]     *  line_mat-
rix[2][2];
                        calculated = 5;
                            if (checker > threshold){
                                line_detected[i][j-1] = 255;
                            } else {
                                line_detected[i][j-1] = 0;
                            }
                        }
                    }
                }
            }
        }
    }
}
    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Gather(line_detected, (pixel-4)*(row_per_process), MPI_INT, calculatedImage, (pixel-
4)*row_per_process, MPI_INT, 0, MPI_COMM_WORLD);
```

```
    /*
        The gathered arrays are gathered in the calcualted Image array and in this section
it is written to the output file.
        The first for loop starts from row_per_process +2 because the array of the process 0
is not filled with data but with zerooes.
    */

    if (rank==0){
        FILE *for_output = fopen(argv[2],"w");
        for (i = row_per_process +2; i < pixel + row_per_process -2; i++){
            for(j = 0; j<pixel-4; j++){
                fprintf(for_output, "%d ", calculatedImage[i][j]);
                //printf("%d ",calculatedImage[i][j]);
            }
            fprintf(for_output, "\n" );
            //  printf("\n");
        }
        fclose(for_output);
    }

/*
    //This code section is to check every process ingredients of arrays.
    //Creates a seperate output file for each process.
    char out[20] = "I";

    for(i=0;i<rank;i++){
        strcat(out,"I");
    }
        //Test output
    FILE *f = fopen(out,"w");
        fprintf(f,"Rank %d The subImage array: \n", rank);
       for (i = 0; i < row_per_process; i++){
            for(j = 0; j<pixel; j++){
                fprintf(f, "%d ", subImage[i][j]);
            }
            fprintf(f, "\n" );
        }
       fprintf(f,"The smooImage array(calculated = %d):\n",calculated);

        for (i = 0; i < row_per_process; i++){
            for(j = 0; j<pixel-2; j++){
                fprintf(f, "%d ", smooIma[i][j]);
            }
            fprintf(f, "\n" );
        }

        fprintf(f,"The line_detected array(calculated = %d):\n",calculated);

        for (i = 0; i < row_per_process; i++){
            for(j = 0; j<pixel-4; j++){
                fprintf(f, "%d ", line_detected[i][j]);
            }
            fprintf(f, "\n" );
        }



        fprintf(f,"Received Lower:\n");

        for (i=0; i<pixel;i++){
            fprintf(f,"%d ",receive_lower[i]);
        }
            fprintf(f, "\n" );
        fprintf(f,"Received Upper:\n");
        for (i=0; i<pixel;i++){
            fprintf(f,"%d ",receive_upper[i]);
        }
            fprintf(f, "\n" );
        fprintf(f,"Send Lower:\n");

        for (i=0; i<pixel;i++){
            fprintf(f,"%d ",send_lower[i]);
        }
            fprintf(f, "\n" );
        fprintf(f,"Send Upper:\n");

        for (i=0; i<pixel;i++){
            fprintf(f,"%d ",send_upper[i]);
        }
            fprintf(f, "\n" );
    fclose(f);


*/

    MPI_Finalize();


    return 0;
}
```