CmpE300: ANALYSIS OF ALGORITHMS

Smoothing and Line Detecting with Parallel Programming

Student Name: Irmak Kavasoğlu

Student Id: 2013400090

Submitted to: Metehan Doyran

Project Type: Programming Project

Submission Date: 26 Dec 2016

Introduction

This project implements a simple smoothing and line detecting algorithm using C programming language with the help of MPI (Message Passing Interface) libraries.

The projects works on black and white images of size 200*200 pixels. Input is taken as a matrice of integers, where 0 represents 0 and 255 represents black. In between values are tones of grey.

To be able to see the result of the projects, a Python script is included in the project. With this script, we can see that the smoothing process results a 198*198 image, which is a blurry version of the original image and the line detecting process results a 196*196 image where the detected lines are white and the background is black. The treshold value for the line detecting algorithm is given by the user, so we can see that higher treshold gives us less lines and a stronger figure and a lower treshold gives a more complicated image with lots of lines.

Program Interface

This project is implemented in C, using MPI libraries. To be able to build and run this projects, user should have these installed. Also, the visualization script is written in Python, therefore the user should have Python installed to be able to see the results as images rather than 40.000 numbers.

User should have the *project.c* file and the input files in the same folder. To build the project, user should open a terminal in the folder which contains the project file, and enter the command *<mpicc -g project.c -o project>.* This command takes in the *project.c* file, compiles it and writes the executable in another file which is named *project.* User can change the executable's name however she wants.

To run the project, user should enter the following command to the terminal which is opened at the same folder:

mpiexec -n <Processors> <executable> <input> <output> <threshold>

Program Execution

As stated above, the program can be run by

mpiexec -n <Processors> <executable> <input> <output> <threshold>

command.

In this command, *<Processors>* is an integer, the total number of processors, n-1 slave processors and 1 boss processor. Since this is a parallel programming project that gives every slave processor equal jobs, user must choose the processor number so that the image size 200 should be divisible by the slave processor count (n-1). *<executable>* is the name of the executable file that user created in the command above. *<input>* and *<output>* are the names of the input and output files. Input file should contain 200 lines with each line containing 200 integers between 0 and 225. The *<treshold>* is an integer, which determines how strict is the line detection.

An example command would be:

<mpiexec -n 21 ./project input.txt tresholded10.txt 10>

which runs the executable with 1 boss and 20 slave processors, input is taken from *input.txt* and the output is written to *tresholded10.txt* with a treshold of 10.

To be able to see the results of the processes, user should use the *<python visualize.py>* command to run the visualize the files. User should manually change the python script to choose which file it will visualize.

Input and Output

The input and output files are made of integers. In our project, our program uses a 200*200 black and white image and gives a output of 196*196. Since it is a too big number to include in this document, we are going to use an example file of 5*12 pixels (numbers). If we run the visualization script on the file below, the image we would get would be this:  (The last line is white).

```
  0    0    0    0    0    0    0    0    0    0    0    0
 50   50   50   50   50   50   50   50   50   50   50   50
100  100  100  100  100  100  100  100  100  100  100  100
175  175  175  175  175  175  175  175  175  175  175  175
255  255  255  255  255  255  255  255  255  255  255  255
```

Program Structure

To make the structure easier to understand, the program is explained function by function. The program itself is given in the appendices, you can see the related functions from there. After the main function, the program will be explained in three main sections; helper functions, boss mode and slave mode.

main function:

The program starts from here. Since we are working with multiple processors, the first thing we do is to get the total number of processors and the rank of the current processor.

We will take the 0th processor as the boss processor and the others as the slaves. The program is divided into two modes; boss mode and slave mode. The main function sends the processors to the relevant mode function.

Since all the data read/write is going to happen in the boss mode, input and output file names are sent to boss and since the thresholding will be done by slaves, threshold number is sent to the slaves.

HELPER FUNCTIONS

readInput/writeOutput:

They read input from a given input file to a given 2d matrice of 200*200, and write output to a given file from a given matrice of 196*196.

allocate2DArray/deallocate2DArray:

The good part about these functions is, they do not allocate/deallocate each row in the matrice seperately. They allocate all the matrice contiguously and set the beginning of each row later; so that they can deallocate just by freeing the first element of the matrice and matrice itself.

smooth/threshold:

These two are used by slave processors. They both take a window of 3*3. The smooth function calculates the average and returns it, where the threshold checks for four dimentions line detection and depending on the given threshold, returns white(255) or black(0).

BOSS MODE

bossMode function:

The first thing the boss does is to read the input. The input is saved into an 200*200 matrix.

Next, the boss distributes the slaves their part of the input. This is, 200/(size-1) lines for each slave processor. After distributing the matrix, boss no longer needs the data so it deallocates input matrix.

Then boss enters the *collectParts* function. With the tags, it indicates that it is going to collect parts of the smoothed image. This function does not return until it collects the results from all the slave processors and merges them into one smoothed image of 198*198 pixels.

It then tells each slave processor to stop waiting, since they will be waiting for a signal to continue after they have sent their smoothed parts to the boss. After all the slaves finish waiting and start thresholding process, the boss deallocates collected smooth image. Note that if we want the smoothed image as an output, we can print it here, before deallocation.

Next, the boss enters the *collectParts* function again, this time with a tag that indicates it is going to collect the threshold parts. This function again will not return until it collects all the parts from all the slaves.

After all the parts are collected, boss again tells everyone to stop waiting since they again will be waiting for a signal from the boss to continue.

The boss writes out the output and deallocates rest of the allocated arrays. All the processes finish properly.

collectParts function:

This function is the best friend of the boss mode. It is used to collect both the smoothed parts and the thresholded parts, then puts the result in the given matrix. It does not return until it collects all the parts.

It can be called with the tags COLLECT_SMOOTHED and COLLECT_THRESHOLDED to indicate for which purpose the function should work for. The main difference between two modes is the array sizes. To collect a smoothed image, it allocates result parts which has 198 columns and for thresholded, it allocates 196 columns. Row numbers are calculated from the total number of processors.

It simply has a for loop, and has an MPI_Recv for each processor. It does not care about the order the slaves turn their results in because it uses the rank from the status.MPI_SOURCE. After the for loop finishes, it can be sure that it received from everyone and returns the function.

SLAVE MODE

slaveMode function:

Each slave starts their image processing from here. They allocate space for their parts and wait for the boss to send their parts to them.

After receiving their parts, they call the *processPart* function with their part of the matrix and the SMOOTHING_PROCESS tag to indicate that this is an smoothing process. This function calculates the smoothed part and returns it to the slave.

Slaves receive the smoothed version of their part from this function, then they send this smoothed part to the boss. After sending the smoothed result to the boss, they call the *waitForBoss* function. This function waits until boss signals the slave to continue, and at the same time, until the signal arrives it can handle the pixel requests from other slaves.

When boss gives permission to continue, slaves call the *processPart* function again, this time with the smoothing part result they just calculated and THRESHOLDING_PROCESS tag to indicate that this is now the thresholding process. After the function returns, they send their thresholding results to the boss, and call the *waitForBoss* function again and wait for the signal from the boss to finish their program.

When boss says it received all the thresholded parts from everyone, slaves return the function and finish the program.

processPart function:

This function does the processing of the images. It can be called with two tags, SMOOTHING_PROCESS and THRESHOLDING_PROCESS. As the names give it away, is calculates the results for both smoothing and thresholding processes.

The first thing it does is to allocate space for the result matrice, that is a 198*198 for smoothing and 196*196 for thresholding.

It traverses each row and column of the given partial matrice. It constructs a window of 3*3 for the process. While constructing this window, if it needs some pixels that the slave does not have in its part, sends a request to the slave that has it.

The slave waits until the window is properly constructed with the needed elements from other slaves if needed. While waiting, it handles all the messages it receives. If the message it received is a request from it, it send the reply to the slave that wants some pixels from it; if the received message is the pixels it asked from others, it fills the window with them.

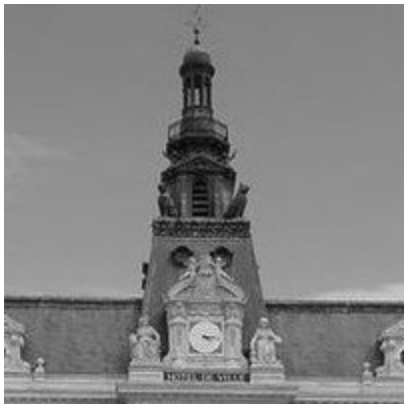
When the window is ready, it calculates the result value for it which is given by *smooth* function for smoothing process and *threshold* function for thresholding process. It puts the result value to the related part of the matrice that is going to be the result of this process, then returns it.

waitForBoss function:

It waits for a signal from the boss. This signal is be customized by the tag that the function is called with. So if this function is called with the tag BOSS_RECEIVED_SMOOTH, it will wait until it receives a signal from the boss with the same tag; same thing applies to BOSS_RECEIVED_THRESHOLDED. If it receives other messages, it also handles them and send the requested pixels to the other slaves that have done the requests.

Examples

On the top, you see the original image that is 200*200. Below, there are the thresholded images with threshold values 10, 25 and 40 respectively.



We can see that as the threshold increases, the number of white pixels decrease, the program detects less lines and gives us a clearer image. However, it the threshold is too high, we might not be able to see the lines that we were supposed to see, which is also not so good.

Improvements and Extensions

In the project description, we were asked to request and send the needed pixels 3 pixels at a time. This resulted that some pixels were requested and sent 3 times between the same slaves. This could be avoided by sending the whole line by one request and not needing anything again.

What could be done more is that since we already know that the processors 2, 3, 4, .., n will request the line from the previous processor and similarly, processors 1, 2, 3, …, n-1 will request a line from the next processor; we could have sent those lines to the related processors in the beginning. This could save us from always keeping an eye on the requests so that if somebody makes a request we could respond.

This project can be taken to the next level by adding support to RGB images or letting the user decide what would be the process to be done (line detecting, smoothing, maybe enchanting colors etc.).

Difficulties Encountered

This was a very good project to practice parallel programming in my opinion. I was a little bit unsure about the memory leaks and I still am. Since we use memory allocation a lot in this project, it could be a good place to learn how to detect and avoid memory leaks, but since there were no recommendations in the description, I didn't try the tools that help detecting leaks but I followed alternative ways to keep track of what I allocate and deallocate (yes I did logging).

Conclusion

This project was a good exersize for parallel programming and it has taught us about how to use MPI technologies.

Appendices: The Source Code

```
/* Student Name: Irmak Kavasoglu
 * Student Number: 2013400090
 * Compile Status: Compiling
 * Program Status: Working
 */
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

#define DEFAULT_TAG 0
#define REQUEST_ABOVE_TAG 1
#define REQUEST_BELOW_TAG 2
#define INCOMING_LINE_TAG 3
#define SMOOTHING_RESULT_TAG 4
#define THRESHOLDING_RESULT_TAG 5
#define BOSS_RECEIVED_SMOOTH 6
#define BOSS_RECEIVED_THRESHOLDED 7
#define SMOOTHING_PROCESS 8
#define THRESHOLDING_PROCESS 9
#define COLLECT_SMOOTHED 10
#define COLLECT_THRESHOLDED 11
/**
 * Allocates a 2D integer array of given size.
 */
int **allocate2DArray(int rows, int cols) {
    int *data = (int *) malloc(rows*cols*sizeof(int));
    int **array= (int **) malloc(rows*sizeof(int*));
    for (int i = 0; i < rows; i++) {
        array[i] = &(data[cols*i]);
    }
    return array;
}

/**
 * Deallocates the allocated array.
 */
void deallocate2DArray(int **array) {
    free(*array);
    free(array);
}
```

```
/**
 * Calculates the average of a 3*3 matrix.
 */
int smooth(int window[3][3]){
    double sum = 0.0;
    int j=0, i=0;
    for(i=0; i<3; i++){
        for(j=0; j<3; j++){
            sum += window[i][j];
        }
    }
    return sum/9.0;
}

/**
 * Returns 0 if the window does not exceed given threshold, 255 otherwise.
 */
int threshold(int window[3][3], int thresholdNumber) {
    int calc[4];
    // Calculate horizontal
    calc[0] = 0;
    for (int i = 0; i < 3; i++) {
        calc[0] -= window[0][i];
        calc[0] += 2*window[1][i];
        calc[0] -= window[2][i];
    }
    // Calculate vertical
    calc[1] = 0;
    for (int i = 0; i < 3; i++) {
        calc[1] -= window[i][0];
        calc[1] += 2*window[i][1];
        calc[1] -= window[i][2];
    }
    // Calculate +45
    calc[2] = 0;
    for (int i = 0; i < 3; i++) {
        calc[2] -= window[(i+1)%3][2-i];
        calc[2] += 2*window[i][2-i];
        calc[2] -= window[i][(4-i)%3];
    }
    // Calculate -45
    calc[3] = 0;
    for (int i = 0; i < 3; i++) {
        calc[3] -= window[(i+1)%3][i];
        calc[3] += 2*window[i][i];
        calc[3] -= window[i][(i+1)%3];
    }
```

```
    //threshold
    int result = 0;
    for (int i = 0; i < 4; i++) {
        if (calc[i] > thresholdNumber) {
            result = 255;
            break;
        }
    }
    return result;
}

/**
 * Reads 200*200 input from file input.txt to the given int **.
 */
void readInput(char *input, int **matrix) {
    // Prepare the reader.
    FILE *reader = fopen(input, "r");

    // Read data.
    for (int i = 0; i < 200; i++) {
        for (int j = 0; j < 200; j++) {
            fscanf(reader, "%d", &matrix[i][j]);
        }
    }
    // Close the reader.
    fclose(reader);
    printf("Data read from %s.\n", input);
}

/**
 * Writes out output.
 */
void writeOutput(int **matrix, int rows, int columns, char *name) {
    // Prepare the writer.
    FILE *writer = fopen(name, "w");

    // Write data.
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < columns; j++) {
            fprintf(writer, "%d ", matrix[i][j]);
        }
        fprintf(writer, "\n");
    }
    // Close the writer.
    fclose(writer);
    printf("Data written to %s.\n", name);
}
```

```
/**
 * Can be function in two ways depending on the tag:
 * - Collector of the smoothed parts if tag is COLLECT_SMOOTHED
 * - Collector of the thresholded parts if tag is COLLECT_THRESHOLDED
 *
 * Does not return until it collects all the parts from every slave processor.
 * Puts the result in the given result matrice.
 */
void collectParts(int size, int tag, int **result) {
   MPI_Status status;

   int slaveRows = 200/(size-1);
   int columns;
   int **resultPart;
   // First and last processor's row number is less. Lack keeps number of
missing lines.
   int lack;
   // Fill the variables depending on the tag.
   if (tag == COLLECT_SMOOTHED) {
      resultPart = allocate2DArray(slaveRows, 198);
      columns = 198;
      lack = 1;
   } else if (tag == COLLECT_THRESHOLDED) {
      resultPart = allocate2DArray(slaveRows, 196);
      columns = 196;
      lack = 2;
   }

   int partSize = slaveRows*columns;

   for (int i = 0; i < size-1; i++) {
      int resultTag = (tag == COLLECT_SMOOTHED) ?
SMOOTHING_RESULT_TAG : THRESHOLDING_RESULT_TAG;
      MPI_Recv(&resultPart[0][0], partSize, MPI_INT, MPI_ANY_SOURCE,
resultTag, MPI_COMM_WORLD, &status);
      //printf("here %d receiving %d.\n", i, status.MPI_SOURCE);
      // if it is the first or last one, take slaveRows-lack line.
      if (status.MPI_SOURCE == 1) {
         for (int j = 0; j < slaveRows-lack; j++) {
            for (int k = 0; k < columns; k++) {
               result[j][k] = resultPart[j][k];
            }
         }
      } else if (status.MPI_SOURCE == size-1) {
         for (int j = 0; j < slaveRows-lack; j++) {
            for (int k = 0; k < columns; k++) {
               int resultRowStart = (status.MPI_SOURCE-1)*slaveRows - lack;
```

```
                result[resultRowStart + j][k] = resultPart[j][k];
            }
        }
    } else {
        for (int j = 0; j < slaveRows; j++) {
            for (int k = 0; k < columns; k++) {
                int resultRowStart = (status.MPI_SOURCE-1)*slaveRows - lack;
                // if there are 200 processors this is possible for 2nd and 199th
processors.
                if (resultRowStart < 0 || resultRowStart >= columns) {
                    continue;
                }
                result[resultRowStart + j][k] = resultPart[j][k];
            }
        }
    }
}

    deallocate2DArray(resultPart);
}

/**
 * Processor 0 will use boss mode.
 * Distributes the matrix parts to slaves.
 * Collects and merges smoothed and thresholded images.
 */
void bossMode(int size, char *input, char *output) {
    // The boss will read the input.
    printf("Reading the input.\n");
    int **matrix = allocate2DArray(200, 200);
    readInput(input, matrix);

    // The boss will send the data to the slaves.
    int slaveRows = 200/(size-1);
    printf("Sending data of %d rows to %d slaves.\n", slaveRows, size-1);
    for (int i = 1; i < size; i++) {
        MPI_Send(&matrix[(i-1)*slaveRows][0], 200*slaveRows, MPI_INT, i,
DEFAULT_TAG, MPI_COMM_WORLD);
    }

    // Deallocate matrix.
    deallocate2DArray(matrix);

    // Allocate space for smoothed image.
    int **smoothed = allocate2DArray(198, 198);

    // Receive the smoothed parts from everyone and merge.
    collectParts(size, COLLECT_SMOOTHED, smoothed);
```

```
    printf("collected smoothed\n");

    // Write smoothed output if you want.
    // writeOutput(smoothed, 198, 198, "smoothed.txt");

    // Tell slaves to stop waiting.
    for (int i = 1; i < size; i++) {
        MPI_Send(&i, 1, MPI_INT, i, BOSS_RECEIVED_SMOOTH,
MPI_COMM_WORLD);
    }

    // Deallocate.
    deallocate2DArray(smoothed);

    // Allocate space for thresholded image.
    int **thresholded = allocate2DArray(196, 196);

    // Receive the thresholded parts from everyone and merge.
    collectParts(size, COLLECT_THRESHOLDED, thresholded);
    printf("collected thresholded\n");

    // Tell slaves to stop waiting.
    for (int i = 1; i < size; i++) {
       MPI_Send(&i, 1, MPI_INT, i, BOSS_RECEIVED_THRESHOLDED,
MPI_COMM_WORLD);
    }
    // Write output and deallocate.
    writeOutput(thresholded, 196, 196, output);
    deallocate2DArray(thresholded);
}

/**
 * Can be called for both smoothing and thresholding processes
 * by using tags SMOOTHING_PROCESS and THRESHOLDING_PROCESS.
 *
 * Performs the given process on the given matrix and returns the result.
 * Communicates with other processes for the missing window parts if needed.
 */
int **processPart(int rank, int size, int **matrixPart, int tag, int
thresholdNumber) {
    MPI_Status status;
    MPI_Request otherRequests;

    // Find number of rows for the given matrix
    int rows = 200/(size-1), columns;
    int resultRows = rows;

    if (tag == SMOOTHING_PROCESS) {
```

```
        columns = 200;
        if ((rank == 1) || (rank == (size-1))) {
            resultRows -= 1;
        }
    } else if (tag == THRESHOLDING_PROCESS) {
        columns = 198;
        if ((rank == 1) || (rank == (size-1))) {
            rows -= 1;
            resultRows -= 2;
            // If there are 200 processors this is possible.
            if (resultRows < 0) {
                resultRows = 0;
            }
        }
    }

    // Allocate space for result matrix. First and last parts have one less line.
    int **processResult = allocate2DArray(resultRows, columns-2);

    // Smooth the part.
    for (int resultRow = 0, currentRow = (rank == 1 ? 1 : 0); resultRow <
resultRows; currentRow++, resultRow++) {
        for (int currCol = 1; currCol < columns-1; currCol++) {
            // Prepare request variables.
            MPI_Request requestAbove, requestBelow;
            int waitingAbove = 0, waitingBelow = 0;

            // Construct 3*3 matrix to be processed.
            int conv[3][3];

            // Put the top three. If we need info from above, send request.
            if (currentRow == 0) {
                int startColumn = currCol - 1;
                MPI_Isend(&startColumn, 1, MPI_INT, rank-1,
REQUEST_ABOVE_TAG, MPI_COMM_WORLD, &requestAbove);
                waitingAbove = 1;
            } else {
                conv[0][0] = matrixPart[currentRow-1][currCol-1];
                conv[0][1] = matrixPart[currentRow-1][currCol];
                conv[0][2] = matrixPart[currentRow-1][currCol+1];
            }

            // Put the mid three
            conv[1][0] = matrixPart[currentRow][currCol-1];
            conv[1][1] = matrixPart[currentRow][currCol];
            conv[1][2] = matrixPart[currentRow][currCol+1];

            // Put the bottom three. If we need info from below, send request.
```

```
        if (currentRow == rows-1) {
            // Request an array of 3, starting from current column-1.
            int startColumn = currCol - 1;
            MPI_Isend(&startColumn, 1, MPI_INT, rank+1,
REQUEST_BELOW_TAG, MPI_COMM_WORLD, &requestBelow);
            waitingBelow = 1;
        } else {
            conv[2][0] = matrixPart[currentRow+1][currCol-1];
            conv[2][1] = matrixPart[currentRow+1][currCol];
            conv[2][2] = matrixPart[currentRow+1][currCol+1];
        }

        // Wait until conv matrix has all needed info.
        while(waitingAbove || waitingBelow) {
            int received[3];
            MPI_Recv(&received, 3, MPI_INT, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            if (status.MPI_TAG == INCOMING_LINE_TAG) {
                // Its the above line if it is from rank-1.
                if (status.MPI_SOURCE == rank-1) {
                    conv[0][0] = received[0];
                    conv[0][1] = received[1];
                    conv[0][2] = received[2];
                    waitingAbove = 0;
                }
                // Its the below line if it is from rank+1.
                if (status.MPI_SOURCE == rank+1) {
                    conv[2][0] = received[0];
                    conv[2][1] = received[1];
                    conv[2][2] = received[2];
                    waitingBelow = 0;
                }
            } else if (status.MPI_TAG == REQUEST_ABOVE_TAG) {
                // Processor below is making a request.
                int startColumn = received[0];
                int requested[3];
                requested[0] = matrixPart[rows-1][startColumn];
                requested[1] = matrixPart[rows-1][startColumn+1];
                requested[2] = matrixPart[rows-1][startColumn+2];
                MPI_Isend(&requested, 3, MPI_INT, rank+1,
INCOMING_LINE_TAG, MPI_COMM_WORLD, &otherRequests);
            } else if (status.MPI_TAG == REQUEST_BELOW_TAG) {
                // Processor above is making a request.
                int startColumn = received[0];
                int requested[3];
                requested[0] = matrixPart[0][startColumn];
                requested[1] = matrixPart[0][startColumn+1];
                requested[2] = matrixPart[0][startColumn+2];
```

```
                MPI_Isend(&requested, 3, MPI_INT, rank-1,
INCOMING_LINE_TAG, MPI_COMM_WORLD, &otherRequests);
            }
        }

        // Now the conv matrix is complete, do the smoothing.
        if (tag == SMOOTHING_PROCESS) {
            processResult[resultRow][currCol-1] = smooth(conv);
        } else if (tag == THRESHOLDING_PROCESS) {
            processResult[resultRow][currCol-1] = threshold(conv,
thresholdNumber);
        }
    }
  }
  return processResult;
}

/**
 * Waits until the boss gives the signal that is the same as the given tag.
 * If it receives requests form other slaves, sends them the parts they want.
 */
void waitForBoss(int rank, int size, int **currentPart, int tag) {
  MPI_Status status;
  MPI_Request otherRequests;

  int rows = 200/(size-1);

  // First and last processor's row number is less. Lack keeps number of
missing lines.
  int lack = 0;
  if ((rank == 1 || rank == (size-1)) && tag == COLLECT_THRESHOLDED) {
    lack = 1;
  }

  if ((rank == 1) || (rank == (size-1))) {
    //  processedRows -= lack;
  }

  int bossSaidOk = 0;
  while (!bossSaidOk) {
    int startColumn;
    MPI_Recv(&startColumn, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, &status);
    if (status.MPI_TAG == REQUEST_ABOVE_TAG) {
      // Processor below is making a request.
      int requested[3];
      requested[0] = currentPart[rows-lack-1][startColumn];
      requested[1] = currentPart[rows-lack-1][startColumn+1];
```

```
            requested[2] = currentPart[rows-lack-1][startColumn+2];
            MPI_Isend(&requested, 3, MPI_INT, rank+1, INCOMING_LINE_TAG,
MPI_COMM_WORLD, &otherRequests);
        } else if (status.MPI_TAG == REQUEST_BELOW_TAG) {
            // Processor above is making a request.
            int requested[3];
            requested[0] = currentPart[0][startColumn];
            requested[1] = currentPart[0][startColumn+1];
            requested[2] = currentPart[0][startColumn+2];
            MPI_Isend(&requested, 3, MPI_INT, rank-1, INCOMING_LINE_TAG,
MPI_COMM_WORLD, &otherRequests);
        } else if (status.MPI_TAG == tag) {
            bossSaidOk = 1;
        }
    }
}

/**
 * Processors 1, 2, ...,n will use slave mode.
 */
void slaveMode(int rank, int size, int thresholdNumber) {
    MPI_Status status;
    MPI_Request otherRequests;

    // Allocate space.
    int rows = 200/(size-1);
    int **matrixPart = allocate2DArray(rows, 200);

    // Receive matrix part.
    MPI_Recv(&matrixPart[0][0], rows*200, MPI_INT, 0, DEFAULT_TAG,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    // Find the number of rows for this processor's part.
    int smoothedRows = rows;
    if ((rank == 1) || (rank == (size-1))) {
        smoothedRows -= 1;
    }

    // Calculate smoothed part, send the result to the boss and wait for its
approval before continuing.
    int **smoothedPart = processPart(rank, size, matrixPart,
SMOOTHING_PROCESS, thresholdNumber);
    MPI_Isend(&smoothedPart[0][0], smoothedRows*198, MPI_INT, 0,
SMOOTHING_RESULT_TAG, MPI_COMM_WORLD, &otherRequests);
    waitForBoss(rank, size, matrixPart, BOSS_RECEIVED_SMOOTH);

    // After boss said continue, slave no longer needs the original matrix part.
    deallocate2DArray(matrixPart);
```

```
    // Find the number of rows for this process.
    int thresholdRows = rows;
    if ((rank == 1) || (rank == (size-1))) {
        thresholdRows -= 2;
        // This is possible if there are 200 processors.
        if (thresholdRows < 0) {
            thresholdRows = 0;
        }
    }
    int **thresholdPart = processPart(rank, size, smoothedPart,
THRESHOLDING_PROCESS, thresholdNumber);
    MPI_Isend(&thresholdPart[0][0], thresholdRows*196, MPI_INT, 0,
THRESHOLDING_RESULT_TAG, MPI_COMM_WORLD, &otherRequests);
    waitForBoss(rank, size, smoothedPart, BOSS_RECEIVED_THRESHOLDED);

    // Deallocate remaining matrices.
    deallocate2DArray(smoothedPart);
    deallocate2DArray(thresholdPart);
}


/**
 * Program starts from main function.
 * Main function redirects processors to the relevant function,
 * depending on if they are slaves or the boss.
 */
int main(int argc, char* argv[])
{
    // Get rank and size.
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Processor 0 is the boss and the others are slaves.
    if (rank == 0) {
        // Send input and output file names to the boss.
        char *input = argv[1];
        char *output = argv[2];
        bossMode(size, input, output);
    } else {
        // Send threshold number to each slave.
        int thresholdNumber = atoi(argv[3]);
        slaveMode(rank, size, thresholdNumber);
    }

    MPI_Barrier(MPI_COMM_WORLD);
```

```
    MPI_Finalize();
    return 0;
}
```