# CMPE 300 Programming Project Report

| | |
|---|---|
| **Course ID:** | CMPE 300.01 |
| **Student Name:** | Yiğit Özkavcı |
| **Submitted By:** | Yiğit Özkavcı |
| **Project Title:** | CMPE300 Programming Project |
| **Submission Date:** | 14.12.2016 |

## Introduction

In this project, we are taking a grayscale image represented with pixel values ranging from 0 to 255 and apply two operations to it: smoothening and thresholding. Throughout this document, these operations will be mentioned as “Smoothening Stage” and “Thresholding Stage”.

Smoothening operation: convolution of each point with a 9x9 matrix of values 1/9.
Thresholding operation: convolution of each point with 4 different filters. These filters exist in project description.

## Program Interface

This project follows the standard C project compilation steps. It contains a Makefile which allows us to compile the program just with:

```
$ make
```

After compilation, there will be a executable named `main` (ELF 64-bit LSB executable for Ubuntu 16.04, Mach-O 64-bit executable for Mac OS X Sierra, tested on both). You can run the program with:

```
$ mpiexec -n <n_of_processors> <program> <in_file> <out_file> <threshold>
```

A working example of compiling and running the program:

```
$ make
$ mpiexec -n 21 ./main input.txt output.txt 10
```

This will run the program with 21 processors (1 master and 20 slave) concurrently, read the image input from input.txt and generate a 10-value thresholded version of the image with name `output.txt`.

Program will not run forever, when all slaves are completed, they send message regarding the situation, and master terminates them, then at last, itself.

# Input and Output

Before you run the program, you should supply an **input.txt** in the same directory you are running the program, no other modifications are needed. For instance, supplied image is the first one below:

Original Image

Smooth Image

Thresholded Image (10)

Thresholded Image (25)

Thresholded Image (40)

First, program generates the second image above, which is the smoothened version of the first one, then generates the third one by using the second one.

# Program Structure

This program utilises MPI(Message Passing Interface) for using concurrency to process files given. Program execution starts in `main()` method, then according to their ranks, logic in main decided whether a process is master or slave, and it sends them to their respective methods called `master()` and `slave()`.

Below is the subsections of the structures/concepts that is used throughout the program. They will be referencing each other by the name of the subsection and all subsections has a #(hash) character at the beginning.

**# Master Process**

1. Convert the given image input to a 2D matrix
2. Split it into slices to send to all slaves. If there are 20 slaves, it shall split input into 20 slices.
3. In an infinite loop, wait for messages from any source, and if there is a message, act accordingly:
    1. It’s a debug message: print it.
    2. It’s a other message type: (see **Tag Types** section)

# **Slave Process**

1. Receive serialised slice from the master (see **Serialisation** section).
2. In an infinite loop, do:
   1. If there is a message, process it according to its tag type (see **Tag Types** section):
      1. `FINISH_SMOOTHING_TAG`: Slave goes into standby mode (see **Standby** section).
      2. `START_THRESHOLDING_TAG`: Slave goes out of standby mode and starts thresholding.
      3. `FINISH_THRESHOLDING_TAG`: Slave terminates.
      4. `DEMAND_DATA_*`: Another slave wants information from this slave. It should first fulfil that other slave's demand, so it does.
   2. If there is no message, this slave does its own job. There are two stages: `SMOOTHING` and `THRESHOLDING`. If slave is in standby state, it simply skips the iteration and uses `continue` statement to return to the beginning of the infinite loop.

# **Standby**

Think that we are in the smoothing stage, and the slave with rank 4 finishes its smoothing job. Naturally, 3 and slave 5 would want point data from the slave 4. If we kill slave 4 right away after it finishes its process, slaves 3 and 5 would not be able to get data from slave 4. Instruction flow of slave will be explained in **Slave Process** section further.

# **Serialisation**

When we want to send 2d arrays as messages, we need to first turn them into a flatten 1d array since we can only send one integer pointer across processes.

# **Slice Types**

It matters whether slice is at the top or middle or bottom because if it's at the top, for example, for smoothing, slave will not process the topmost row, whereas a middle slave will process it hence we have 3 types of slices to send out to slaves:

- `SLICE_TYPE_TOP`
- `SLICE_TYPE_MIDDLE`
- `SLICE_TYPE_BOTTOM`

It's been done like this because slave should not know about how to act based on its rank. This logic should be encapsulated and only be decided by the master process.

# **Tag Types**

`SLICE_TAG`:
Master sends slave its slice data. Slice is a 2d matrix being sent in serialised format(see **Serialisation** section)

`SLICE_SIZE_TAG`:
Master sends the slice size to slave since slave will be receiving the data accordingly.

`SLICE_TYPE_TAG`:
Type of the slice. We are sending this to slave in order to tell it how to act according to its position.

`DEMAND_DATA_FROM_UPPER_SLICE_TAG`:
When a slave demands data from a upper slice, it sends a message with this tag. Slave sends the message containing X-position, which means that "I want the data of the points at index X from your bottom-most row.

DEMAND_DATA_FROM_LOWER_SLICE_TAG:
When a slave demands data from a lower slice, it sends a message with this tag. Message means that "I want the data of the points at index X from your top-most row.

DEBUG_MESSAGE_1_TAG: (see **Debugging** section)
DEBUG_MESSAGE_2_TAG: (see **Debugging** section)
DEBUG_MESSAGE_3_TAG: (see **Debugging** section)
DEBUG_MESSAGE_4_TAG: (see **Debugging** section)

SMOOTHING_DONE_TAG:
When a slave finished smoothing, it notifies the master first by sending a message with this tag. It's important that we don't terminate the slave, or switch its job immediately, but first notify the master about it. This behaviour will be explained further (see **Master** and **Slave** sections)

FINISH_SMOOTHING_TAG:
When all slaves notified master with `SMOOTHING_DONE_TAG`, master takes action to tell all slaves to finish their smoothing jobs. "Tell" here means sending message with `FINISH_SMOOTHING_TAG` to all slaves (this has been done with `MPI_Isend()` asynchronously since master does not necessarily needs to wait for each send).
This does not tell them to start thresholding stage though, for convenience. This lets slaves get into standby (see **Standby** section).

START_THRESHOLDING_TAG:
After all slaves finished their smoothing, informed the master (with `SMOOTHING_DONE_TAG`) and received `FINISH_SMOOTHING_TAG` from master, they receive `START_THRESHODING_TAG` from master. This also means that every other slave finished their smoothing as well, meaning they all have their smoothened slices hence they can give its data whenever its needed.

THRESHOLDING_DONE_TAG:
After slaves finish their thresholding, they inform master about that. Slave sends the size of its serialised thresholded slice array with this message, and also sends a following message after this one.

FOLLOWING_THRESHOLDING_DONE_TAG:
Master fills its `master_thresholded_matrix` with the slice it received from the slave which has sent the message with this tag. The `master_thresholded_matrix` will be gradually filled from the serialised arrays it received from slaves. This `master_thresholded_matrix` is the output of the program

FINISH_THRESHOLDING_TAG:
After a slave finishes its thresholding stage, it sends a message with this tag to Master process. BUT slave does not die right away, it just goes to standby. (see **Standby** section).

# Debugging

When it comes to printing data to print for debugging, slaves cannot synchronously do this. This way also considered a bad-practice since it depends on a race-condition, so in this project, I implemented a debugging system like this:

- When a slave wants to print information, it uses one of four methods: debug_1, debug_2, debug_3, debug_4. Their explanations are in the debug.h file, so I will not go into details here, but only inform you about the debug concept that is implemented. These methods simply sends *blocking* message to master, and master prints it.
- Master always listens to debug messages when it does not have anything else to do such as sending FINISH_SMOOTHING_TAG and similar tag to slaves. Any other work that master has to do has precedence over debugging messages. This way, we eliminate the race conditions that would be caused because of slaves' stdouts.

## Examples

You can try running the program with different thresholds. Threshold of value 10 gives the best-looking binary image result, but you can also try with 25, 40 or any other value like this:

```
$ mpiexec -n 21 ./main input.txt output.txt 10
$ mpiexec -n 21 ./main input.txt output.txt 25
$ mpiexec -n 21 ./main input.txt output.txt 40
```

## Improvements and Extensions

Program can be improved further in terms of switching job from smoothing to thresholding;

Current architecture:

All slaves work on smoothing, after finishing they tell it to master and when its time master tells them to start thresholding.

Why?

Suppose slave A is already in its thresholding stage and it demands row data from slave B. If slave B haven't finished its smoothing stage yet, it's possible that slave B does not have its smoothened matrix data that slave A demands.

A better approach:

Current architecture is safe, but it can be optimised to provide a more asynchronous solution. Suppose slave A is already in its thresholding stage and demands row data from slave B. If slave B does not have that row data yet, slave A should wait in a blocking MPI_Recv call. When slave B receives the message which informs it that slave A demands data from it but it does not have the data, slave B should just ignore the message and continue its smoothing job. Eventually, slave B will have the data slave A wants and this way, we can use "concurrency" in a better way.

This approach promotes "concurrency" over "parallelism", which I think is the whole point of this project.

## Difficulties Encountered

Getting used to MPI is hard, because its complexity grows as the number communication channels between slices increase.

An absolute difficulty was debugging the program. There are lots of messages being passed both master-to-slave and slave-to-slave, and it was hard to keep track of every one of them. After segmentation fauts, I used valgrind, a free software for debugging general-purpose C programs, which also works for MPI; and it helped.

Passing multi-dimensional arrays with C and MPI is not straightforward. To do this, I had to implement a serialisation and deserialisation mechanism, which was hard when you think about all memory allocation & deallocation problems.

## Conclusion

This project was a very good example for teaching how to work with multi-threading systems. It's not only a parallel application, but a concurrently working, collaborative one. MPI has a very nice api to work with, and it also has great documentation.

# Appendix

### tag_types.h

```
#ifndef TAG_TYPES_H_
#define TAG_TYPES_H_

#define DIETAG 1
#define SLICE_TAG 2
#define SLICE_SIZE_TAG 3
#define SLICE_TYPE_TAG 4
#define DEMAND_DATA_FROM_UPPER_SLICE_TAG 5
#define DEMAND_DATA_FROM_LOWER_SLICE_TAG 6
#define POINT_DATA_TAG 7
#define DEBUG_MESSAGE_1_TAG 8
#define DEBUG_MESSAGE_2_TAG 9
#define DEBUG_MESSAGE_3_TAG 10
#define DEBUG_MESSAGE_4_TAG 11
#define DEBUG_MESSAGE_FOLLOWUP_TAG 12
#define SMOOTHING_DONE_TAG 13
#define FOLLOWING_SMOOTHING_DONE_TAG 14
#define THRESHOLDING_DONE_TAG 15
#define FOLLOWING_THRESHOLDING_DONE_TAG 16
#define FINISH_SMOOTHING_TAG 17
#define START_THRESHOLDING_TAG 18
#define FINISH_THRESHOLDING_TAG 19

#endif
```

### debug.h

```
#ifndef DEBUG_H_
#define DEBUG_H_

/*********************************************************************
 * Debug without any parameter.
 *********************************************************************/
void debug_1(char *message, int *rank);

/*********************************************************************
 * Debug with 1 integer parameter. Mesage should contain one "%d"
 * inside, just like printf.
 *********************************************************************/
void debug_2(char *message, int *arg1, int *rank);

/*********************************************************************
 * Debug with 2 integer parameters.
 *********************************************************************/
void debug_3(char *message, int *arg1, int *arg2, int *rank);

/*********************************************************************
 * Debug with 3 integer parameters.
 *********************************************************************/
void debug_4(char *message, int *arg1, int *arg2, int *arg3, int *rank);

#endif // DEBUG_H_
```

### debug.c

```
#include "debug.h"
#include "mpi.h"
#include <string.h>
#include "tag_types.h"

void debug_1(char *message, int *rank) {
  MPI_Send(message, strlen(message), MPI_CHAR, 0, DEBUG_MESSAGE_1_TAG, MPI_COMM_WORLD);
}

void debug_2(char *message, int *arg1, int *rank) {
  MPI_Send(message, strlen(message), MPI_CHAR, 0, DEBUG_MESSAGE_2_TAG, MPI_COMM_WORLD);
  MPI_Send(arg1, 1, MPI_INT, 0, DEBUG_MESSAGE_FOLLOWUP_TAG, MPI_COMM_WORLD);
```

```
}

void debug_3(char *message, int *arg1, int *arg2, int *rank) {
  MPI_Send(message, strlen(message), MPI_CHAR, 0, DEBUG_MESSAGE_3_TAG, MPI_COMM_WORLD);
  MPI_Send(arg1, 1, MPI_INT, 0, DEBUG_MESSAGE_FOLLOWUP_TAG, MPI_COMM_WORLD);
  MPI_Send(arg2, 1, MPI_INT, 0, DEBUG_MESSAGE_FOLLOWUP_TAG, MPI_COMM_WORLD);
}

void debug_4(char *message, int *arg1, int *arg2, int *arg3, int *rank) {
  MPI_Send(message, strlen(message), MPI_CHAR, 0, DEBUG_MESSAGE_4_TAG, MPI_COMM_WORLD);
  MPI_Send(arg1, 1, MPI_INT, 0, DEBUG_MESSAGE_FOLLOWUP_TAG, MPI_COMM_WORLD);
  MPI_Send(arg2, 1, MPI_INT, 0, DEBUG_MESSAGE_FOLLOWUP_TAG, MPI_COMM_WORLD);
  MPI_Send(arg3, 1, MPI_INT, 0, DEBUG_MESSAGE_FOLLOWUP_TAG, MPI_COMM_WORLD);
}
```

### util.h

```
#ifndef UTIL_H_
#define UTIL_H_

#include "util.h"
#include <stdio.h>
#include <stdlib.h>
#include "tag_types.h"
#include <stdbool.h>

/* There are 3 types of slices: */
#define SLICE_TYPE_TOP 1    /* Meaning slice is at the very top. */
#define SLICE_TYPE_MIDDLE 2 /* Meaning slice is neither at the top or bottom. */
#define SLICE_TYPE_BOTTOM 3 /* Meaning slice is at the very bottom. */

/* There are 2 stages to work on for each slave: */
#define STAGE_SMOOTHING 1
#define STAGE_THRESHOLDING 2

/*********************************************************************
 * Given slice type and dimensions, decides what start and
 * end points of matrix we are going to work on.
 *********************************************************************/
void util_decide_starting_position(
  int slice_type, /* Type of the slice. It can take 3 values, which are defined above
this file. */
  int row_count,      /* Row count of slice matrix. */
  int col_count,      /* Column count of slice matrix. */
  int stage,          /* Stage can be whether SMOOTHING or THRESHOLDING. */
  int is_slice_alone, /* Stage can be whether SMOOTHING or THRESHOLDING. */
  int* start_x,       /* X dimension of starting point. */
  int* start_y,       /* Y dimension of starting point. */
  int* end_x,         /* X dimension of ending point. */
  int* end_y          /* Y dimension of ending point. */
);

/*********************************************************************
 * Given dimensions, allocates a matrix.
 *********************************************************************/
int** util_alloc_matrix(
  int row_count, /* Row count of matrix to return. */
  int col_count  /* Row count of matrix to return. */
);


/*********************************************************************
 * When any other slice demands data of 3 points from you, you simply
 * give them. This method gives information of 3 points of its slice
 * matrix: (x_index - 1, x_index, x_index + 1).
 *
 * Tag is used to determine whether information is demanded from top
 * or bottom. If bottom, this method also needs to know end_y.
 *********************************************************************/
int* util_prepare_points_for_demander(int** slice_matrix, int x_index, int tag, int
end_y, int stage);
```

```
/*******************************************************************
 * Special row is the row of which took data from another slice.
 *
 * Think of a slice;
 * if row is the bottom row of this particular slice, it's called the low row.
 * else if it's the top row, it's high row.
 * Else, it's neither of them.
 *
 * We need this because... well, reasons. Read the implementation.
 *******************************************************************/
int util_determine_special_row(int slice_type, bool is_low_row, bool is_high_row);

#endif
```

**util.c**

```
#include "util.h"

void util_decide_starting_position(
  int slice_type,
  int row_count,
  int col_count,
  int stage,
  int is_slice_alone,
  int* start_x,
  int* start_y,
  int* end_x,
  int* end_y
) {
  if(stage == STAGE_SMOOTHING) {
    *start_x = 1;
    *end_x = col_count - 1;
    if(slice_type == SLICE_TYPE_MIDDLE) {
      *start_y = 0;
      *end_y = row_count;
    } else if(slice_type == SLICE_TYPE_TOP) {
      *start_y = 1;
      *end_y = row_count;
    } else if(slice_type == SLICE_TYPE_BOTTOM) {
      *start_y = 0;
      *end_y = row_count - 1;
    } else {
      printf("Wrong slice type: %d", slice_type);
      exit(0);
    }
  } else if(stage == STAGE_THRESHOLDING) {
    *start_x = 2;
    *end_x = col_count - 2;
    if(slice_type == SLICE_TYPE_MIDDLE) {
      *start_y = 0;
      *end_y = row_count;
    } else if(slice_type == SLICE_TYPE_TOP) {
      *start_y = 2;
      *end_y = row_count;
    } else if(slice_type == SLICE_TYPE_BOTTOM) {
      *start_y = 0;
      *end_y = row_count - 2;
    } else {
      printf("Wrong slice type: %d", slice_type);
      exit(0);
    }
  } else {
    printf("Wrong stage type: %d", stage);
  }
  if(is_slice_alone == 1) {
    if(stage == STAGE_SMOOTHING) {
      *start_y = 1;
      *end_y = row_count - 1;
    } else if(stage == STAGE_THRESHOLDING) {
      *start_y = 2;
      *end_y = row_count - 2;
    }
```

```
  }
}

int** util_alloc_matrix(int row_count, int col_count) {
  // Allocating space for our smoothened_slice
  int** result = (int**)malloc(col_count * sizeof(int*));
  for(int col = 0; col < col_count; col++) {
    *(result + col) = (int*)malloc(row_count * sizeof(int));
    for(int row = 0; row < row_count; row++) {
      *(*(result + col) + row) = 0;
    }
  }
  return result;
}

int* util_prepare_points_for_demander(int** slice_matrix, int x_index, int tag, int
end_y, int stage) {
  int y_index;
  /*
   * Determining y-index based on demand type of data.
   * Slaves can demand data from either higher or lower rows.
   */
  if(tag == DEMAND_DATA_FROM_UPPER_SLICE_TAG) {
    y_index = end_y - 1;
  } else if(tag == DEMAND_DATA_FROM_LOWER_SLICE_TAG) {
    y_index = 0;
  } else {
    printf("[!] Wrong tag: %d\n", tag);
    exit(0);
  }

  /* Writing point data to send */
  int* points;
  if(stage == STAGE_SMOOTHING) {
    points = malloc(sizeof(int) * 3);
    for(int i = x_index - 1; i <= x_index + 1; i++) {
      *(points + i - x_index + 1) = *(*(slice_matrix + y_index) + i);
    }
  } else {
    points = malloc(sizeof(int) * 3);
    for(int i = x_index - 1; i <= x_index + 1; i++) {
      *(points + i - x_index + 1) = *(*(slice_matrix + i) + y_index);
    }
  }


  /* printf("Sending %d %d %d\n", *points, *(points + 1), *(points + 2)); */
  return points;
}

int util_determine_special_row(int slice_type, bool is_low_row, bool is_high_row) {
  int special_row;
  if(slice_type == SLICE_TYPE_TOP) {
    if(is_low_row) {
      special_row = 3;
    } else {
      special_row = 0;
    }
  } else if(slice_type == SLICE_TYPE_MIDDLE) {
    if(is_high_row) {
      special_row = 1;
    } else if(is_low_row) {
      special_row = 3;
    } else {
      special_row = 0;
    }
  } else if(slice_type == SLICE_TYPE_BOTTOM) {
    if(is_high_row) {
      special_row = 1;
    } else {
      special_row = 0;
    }
```

```
  } else {
    printf("Wrong slice type: %d", slice_type);
    exit(0);
  }

  return special_row;
}
```

**slice.h**

```
#ifndef SLICE_H_
#define SLICE_H_

/*******************************************************************
 * Given an array, forms a row_count to col_count matrix based on it,
 *******************************************************************/
int** deserialize_slice(int* slice, int row_count, int col_count);

/*******************************************************************
 * Splits image into given sizes and returns serialized version
 * of the slice in given index.
 *******************************************************************/
int* extract_slice(int** image, int image_size, int image_slice_size, int index);

/*******************************************************************
 * Given a slice matrix, serializes it into an array.
 *******************************************************************/
int* serialize_slice(int** slice, int row_count, int col_count);

#endif // SLICE_H_
```

**slice.c**

```
#include "slice.h"
#include <stdlib.h>
#include <stdio.h>

int* extract_slice(int** image, int image_size, int image_slice_size, int index) {
  // We are keeping slice as contiguous mamory because it'll be passed via MPI
  int* slice = malloc(sizeof(int) * image_slice_size * image_size);
  for(int row = index * image_slice_size; row < (index + 1) * image_slice_size; row++) {
    for(int col = 0; col < image_size; col++) {
      *(slice + col + row * image_size - index*image_size*image_slice_size) = *(*(image +
col) + row);
    }
  }
  return slice;
}

int** deserialize_slice(int* slice, int row_count, int col_count) {
  // Allocating space for new slice
  int** new_slice = (int**)malloc(row_count * sizeof(int*));
  for(int row = 0; row < row_count; row++) {
    *(new_slice + row) = (int*)malloc(col_count * sizeof(int));
  }

  // Filling new slice matrix
  for(int row = 0; row < row_count; row++) {
    for(int col = 0; col < col_count; col++) {
      *(*(new_slice + row) + col) = *(slice + row * col_count + col);
    }
  }

  free(slice);
  return new_slice;
}

int* serialize_slice(int** slice, int row_count, int col_count) {
  int* flat_slice = (int*)malloc(sizeof(int) * row_count * col_count);
  for(int row = 0; row < row_count; row++) {
    for(int col = 0; col < col_count; col++) {
      int num = *(*(slice + col) + row);
```

```
      *(flat_slice + col_count * row + col) = num;
    }
  }

  return flat_slice;
}
```

**main.c**

```
/* Student Name: Yiğit Özkavcı
 * Student Number: 2013400111
 * Compile Status: Compiling
 * Program Status: Working
 *
 * Notes:
 * You can compile the code on any unix/linux platform with typing:
 * $ make
 *
 * Then run with:
 * $ mpiexec -n <n_of_processors> <program> <in_file> <out_file> <threshold>
 */

#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"
#include <stdbool.h>
#include "string.h"
#include "tag_types.h"
#include "debug.h"
#include "slice.h"
#include "util.h"

const int horizontal_ld[3][3] =  { { -1, -1, -1 }, { 2, 2, 2 }, { -1, -1, -1} };
const int vertical_ld[3][3] =    { { -1, 2, -1 }, { -1, 2, -1 }, { -1, 2, -1} };
const int obl_plus45_ld[3][3] =  { { -1, -1, 2 }, { -1, 2, -1 }, { 2, -1, -1} };
const int obl_minus45_ld[3][3] = { { 2, -1, -1 }, { -1, 2, -1 }, { -1, -1, 2} };

#define IMAGE_SIZE 200
int THRESHOLD;
char* INPUT_FILENAME;
char* OUTPUT_FILENAME;

/*********************************************************************
 * Reads the input and makes a matrix out of it.
 *********************************************************************/
int** image_from_input() {
  int** image = (int**)malloc(sizeof(int*) * IMAGE_SIZE);
  for(int i = 0; i < IMAGE_SIZE; i++) {
    *(image + i) = (int*)malloc(sizeof(int) * IMAGE_SIZE);
  }
  FILE* file = fopen(INPUT_FILENAME, "r");
  for(int i = 0; i < IMAGE_SIZE; i++) {
    for(int j = 0; j < IMAGE_SIZE; j++) {
      int val;
      fscanf(file, "%d", &val);
      *(*(image + j) + i) = val;
    }
  }
  fclose(file);
  return image;
}

/*********************************************************************
 * In order to threshold a point, we need 3 rows.
 *
 * row_1: starting address of row 1
 * row_2: starting address of row 2
 * row_3: starting address of row 3
 *********************************************************************/
int threshold_point(int* row_1, int* row_2, int* row_3, int* rank) {
  int res[4] = { 0, 0, 0, 0 };
```

```
  for(int i = 0; i < 3; i++) {
    res[0] += *(row_1 + i) * horizontal_ld[0][i] + *(row_2 + i) * horizontal_ld[1][i] +
*(row_3 + i) * horizontal_ld[2][i];
    res[1] += *(row_1 + i) * vertical_ld[0][i] + *(row_2 + i) * vertical_ld[1][i] +
*(row_3 + i) * vertical_ld[2][i];
    res[2] += *(row_1 + i) * obl_plus45_ld[0][i] + *(row_2 + i) * obl_plus45_ld[1][i] +
*(row_3 + i) * obl_plus45_ld[2][i];
    res[3] += *(row_1 + i) * obl_minus45_ld[0][i] + *(row_2 + i) * obl_minus45_ld[1][i] +
*(row_3 + i) * obl_minus45_ld[2][i];
  }
  /* bool over_threshold = false; */
  for(int i = 0; i < 4; i++) {
    if(res[i] > THRESHOLD) {
      return 255;
    }
  }
  return 0;
}
```

```
/*******************************************************************
 * In order to smoothen a point, we need 3 rows.
 *
 * row_1: starting address of row 1
 * row_2: starting address of row 2
 * row_3: starting address of row 3
 *******************************************************************/
int smoothen_point(int* row_1, int* row_2, int* row_3, int* rank) {
  double smoother_val = (1.0)/9;
  int total = 0;
  for(int i = 0; i < 3; i++) {
    total += *(row_1 + i) + *(row_2 + i) + *(row_3 + i);
  }
  return (int) (total * smoother_val);
}

/*******************************************************************
 * Demands data of three points from either top or bottom slice.
 *******************************************************************/
bool demand_point_data(int* curr_x, int rank, int* received_vals, char type, bool*
is_demanded) {
  MPI_Status status;
  int send_tag;
  int remote_rank;
  if(type == 'u') {
    send_tag = DEMAND_DATA_FROM_UPPER_SLICE_TAG;
    remote_rank = rank - 1;
  } else if(type == 'l') {
    send_tag = DEMAND_DATA_FROM_LOWER_SLICE_TAG;
    remote_rank = rank + 1;
  } else {
    debug_1("[!] WRONG TYPE FOR demanding_point_data", &rank);
    return false;
  }

  int message_exists = 0;
  if(*is_demanded == false) {
    MPI_Send(
      curr_x,         // initial address of send buffer (choice)
      1,              // number of elements in send buffer (integer)
      MPI_INT,        // type of elements in send buffer (handle)
      remote_rank,    // rank of destination (integer)
      send_tag,       // send tag (integer)
      MPI_COMM_WORLD  // communicator
    );
```

```
    *is_demanded = true;
  }

  while(1) {
    MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &message_exists, &status);
    if(!message_exists || status.MPI_SOURCE == rank) continue;
    if(status.MPI_SOURCE == remote_rank && status.MPI_TAG == (50 + rank)) {
      break;
    } else {
      return false;
    }
  }

  /* Only receive messages that are explicit for me. */
  MPI_Recv(
    received_vals,  /* address of receive buffer */
    3,              /* number of elements in receive buffer (integer) */
    MPI_INT,        /* type of elements in receive buffer (handle) */
    remote_rank,    /* rank of source (integer) */
    (50 + rank),    /* receive tag (integer) */
    MPI_COMM_WORLD, /* communicator */
    &status         /* status */
  );

  return true;
}

bool is_debug_tag(int tag) {
  return (tag == DEBUG_MESSAGE_1_TAG) ||
         (tag == DEBUG_MESSAGE_2_TAG) ||
         (tag == DEBUG_MESSAGE_3_TAG) ||
         (tag == DEBUG_MESSAGE_4_TAG);
}


/*********************************************************************
 * Master process
 *********************************************************************/
void master() {
  int proc_size, rank;
  MPI_Status status;
  MPI_Comm_size(MPI_COMM_WORLD, &proc_size);

  int** image = image_from_input();
  int image_slice_size = IMAGE_SIZE * IMAGE_SIZE / (proc_size - 1);
  int image_slice_type;

  // Giving slaves their slices of image
  int is_slave_alone = (proc_size == 2) ? 1 : 0;
  for(rank = 1; rank < proc_size; rank++) {
    MPI_Send(&is_slave_alone, 1, MPI_INT, rank, SLICE_SIZE_TAG, MPI_COMM_WORLD);
    int* image_slice = extract_slice(image, IMAGE_SIZE, image_slice_size/IMAGE_SIZE, rank
- 1);
    MPI_Send(&image_slice_size, 1, MPI_INT, rank, SLICE_SIZE_TAG, MPI_COMM_WORLD);
    *(image_slice + image_slice_size) = IMAGE_SIZE;
    MPI_Send(image_slice, image_slice_size + 1, MPI_INT, rank, SLICE_TAG,
MPI_COMM_WORLD);
    if(is_slave_alone) {
      image_slice_type = SLICE_TYPE_MIDDLE;
    } else {
      if(rank == 1) {
        image_slice_type = SLICE_TYPE_TOP;
      } else if(rank == proc_size - 1) {
        image_slice_type = SLICE_TYPE_BOTTOM;
      } else {
        image_slice_type = SLICE_TYPE_MIDDLE;
      }
    }
    MPI_Send(&image_slice_type, 1, MPI_INT, rank, SLICE_TYPE_TAG, MPI_COMM_WORLD);
  }

  for(int i = 0; i < IMAGE_SIZE; i++) {
```

```
    free(*(image + i));
  }
  free(image);

  // Listening for debug messages and printing them
  int job_to_finish = 1;
  int smoothing_job_to_finish = 1;
  int message_exists;
  int smoothing_finished_count = 0;
  int thresholding_finished_count = 0;
  bool thresholding_jobs_sent = false;


  /* Allocating space for the new smoothened and thresholded image. */
  int** master_smoothened_image = util_alloc_matrix(IMAGE_SIZE, IMAGE_SIZE);
  int** master_thresholded_image = util_alloc_matrix(IMAGE_SIZE, IMAGE_SIZE);










/***********************************************************************
   * What happens here is this:
   *
   * Master has 2 main jobs that it needs to perform infinitely:
   * - If receives a debug message, prints it.
   * - If a slave informs master about job complete, it master takes act  accordingly.
   *   - If slave completes a SMOOTHING stage, master increases
   *     smoothing_finished_count. If this count equals to slave count, this
   *     means that master needs to transmit stage of all slaves from
   *     SMOOTHING to THRESHOLDING.
   *   - If a slave completes a THRESHOLDING stage, master increases
   *     thresholding_finished_count. If this count equals to slave count, this
   *     means that master needs to send JOB_DONE to all slaves, which kills
   *     their processes.
   *
   ***********************************************************************/
  for(;;) {
    if(job_to_finish == proc_size) {
      printf("Thresholding is finished for all slaves.\n");
      FILE *f;

      f = fopen("smoothened.txt", "w");
      for(int row = 0; row < IMAGE_SIZE; row++) {
        for(int col = 0; col < IMAGE_SIZE; col++) {
          fprintf(f, "%d ", *(*(master_smoothened_image + col) + row));
        }
        fprintf(f, "\n");
      }
      fprintf(f, "\n");
      f = fopen(OUTPUT_FILENAME, "w");
      for(int row = 0; row < IMAGE_SIZE; row++) {
        for(int col = 0; col < IMAGE_SIZE; col++) {
          fprintf(f, "%d ", *(*(master_thresholded_image + col) + row));
        }
        fprintf(f, "\n");
      }
      fprintf(f, "\n");
      break;
    } else if(smoothing_job_to_finish == proc_size && !thresholding_jobs_sent) {
      printf("All smoothing jobs are finished. Sending START_THRESHOLDING_TAG to all\n");
      int temp;
      MPI_Request request;
      for(int i = 1; i < proc_size; i++) {
```

```
        MPI_Isend(&temp, 1, MPI_INT, i, START_THRESHOLDING_TAG, MPI_COMM_WORLD,
&request);
      }
      thresholding_jobs_sent = true;
    } else {
      MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &message_exists, &status);
      if(!message_exists && thresholding_finished_count == proc_size - 1) {
        int temp;
        MPI_Request request;
        MPI_Isend(&temp, 1, MPI_INT, job_to_finish, FINISH_THRESHOLDING_TAG,
MPI_COMM_WORLD, &request);
        job_to_finish++;
        continue;
      } else if(!message_exists && smoothing_finished_count == proc_size - 1 && !
thresholding_jobs_sent) {
        int temp;
        MPI_Request request;
        MPI_Isend(&temp, 1, MPI_INT, smoothing_job_to_finish, FINISH_SMOOTHING_TAG,
MPI_COMM_WORLD, &request);
        smoothing_job_to_finish++;
        continue;
      }
    }

    int sender, arg1, arg2, arg3, message_length, message_exists;
    MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &message_exists, &status);

    sender = status.MPI_SOURCE;
    if(message_exists) {
      if(is_debug_tag(status.MPI_TAG)) {
        int msg_length;
        MPI_Get_count(&status, MPI_CHAR, &msg_length);
        char* message = malloc(msg_length * sizeof(char));
        MPI_Recv(message, msg_length, MPI_CHAR, sender, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);
        MPI_Get_count(&status, MPI_CHAR, &message_length);
        *(message + message_length) = '\0';
        printf("[%d] ", sender);
        if(status.MPI_TAG == DEBUG_MESSAGE_1_TAG) {
          printf("%s", message);
        } else if(status.MPI_TAG == DEBUG_MESSAGE_2_TAG) {
          MPI_Recv(&arg1, 1, MPI_INT, sender, DEBUG_MESSAGE_FOLLOWUP_TAG, MPI_COMM_WORLD,
&status);
          printf(message, arg1);
        } else if(status.MPI_TAG == DEBUG_MESSAGE_3_TAG) {
          MPI_Recv(&arg1, 1, MPI_INT, sender, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
          MPI_Recv(&arg2, 1, MPI_INT, sender, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
          printf(message, arg1, arg2);
        } else if(status.MPI_TAG == DEBUG_MESSAGE_4_TAG) {
          MPI_Recv(&arg1, 1, MPI_INT, sender, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
          MPI_Recv(&arg2, 1, MPI_INT, sender, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
          MPI_Recv(&arg3, 1, MPI_INT, sender, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
          printf(message, arg1, arg2, arg3);
        }
        printf("\n");
        free(message);
      } else if(status.MPI_TAG == SMOOTHING_DONE_TAG) {
        /* Receiving slice array length from slave */
        int slice_arr_length;
        MPI_Recv(&slice_arr_length, 1, MPI_INT, sender, SMOOTHING_DONE_TAG,
MPI_COMM_WORLD, &status);

        /* Receiving deserialized slice array from slave */
        int* slice_arr = malloc(sizeof(int) * slice_arr_length);
        MPI_Recv(slice_arr, slice_arr_length, MPI_INT, sender,
FOLLOWING_SMOOTHING_DONE_TAG, MPI_COMM_WORLD, &status);

        /* Putting that serialized slice array to master_smoothened_image. */
        int slice_row_count = IMAGE_SIZE / (proc_size - 1);
        int slice_col_count = IMAGE_SIZE;
        for(int row = 0; row < slice_row_count; row++) {
          for(int col = 0; col < slice_col_count; col++) {
```

```
            int slice_row_offset = (sender - 1) * slice_row_count;
            int arr_val = *(slice_arr + row * slice_col_count + col);
            *(*(master_smoothened_image + col) + row + slice_row_offset) = arr_val;
          }
        }
        smoothing_finished_count++;
      } else if(status.MPI_TAG == THRESHOLDING_DONE_TAG) {
        /* Receiving slice array length from slave */
        int slice_arr_length;
        MPI_Recv(&slice_arr_length, 1, MPI_INT, sender, THRESHOLDING_DONE_TAG,
MPI_COMM_WORLD, &status);

        /* Receiving deserialized slice array from slave */
        int* slice_arr = malloc(sizeof(int) * slice_arr_length);
        MPI_Recv(slice_arr, slice_arr_length, MPI_INT, sender,
FOLLOWING_THRESHOLDING_DONE_TAG, MPI_COMM_WORLD, &status);

        /* Putting that serialized slice array to master_thresholded_image. */
        int slice_row_count = IMAGE_SIZE / (proc_size - 1);
        int slice_col_count = IMAGE_SIZE;
        for(int row = 0; row < slice_row_count; row++) {
          for(int col = 0; col < slice_col_count; col++) {
            int slice_row_offset = (sender - 1) * slice_row_count;
            int arr_val = *(slice_arr + row * slice_col_count + col);
            *(*(master_thresholded_image + col) + row + slice_row_offset) = arr_val;
          }
        }
        thresholding_finished_count++;
      }
    }
  }
}

/*********************************************************************
* Manages the process of point smoothing. Takes the whole slice
* matrix, the current position and a special case indicator. Special
* case is where we need a row information from another slave.
* For this, we are sending a message using `demand_point_data` method.
*
* SLICE POSITIONING
*
* ------------------------ *
* HIGH   slice HIGH   row  *
* HIGH   slice MIDDLE row  *
* HIGH   slice LOW    row  *
* ------------------------ *
* MIDDLE slice HIGH   row  *
* MIDDLE slice MIDDLE row  *
* MIDDLE slice LOW    row  *
* ------------------------ *
* BOTTOM slice HIGH   row  *
* BOTTOM slice MIDDLE row  *
* BOTTOM slice LOW    row  *
* ------------------------ *

*********************************************************************/
void process_rows_for_smoothing(
  int** slice_matrix,    /* Matrix that this slave is responsible of. */

  int curr_x,            /* Current X position that this slave
                            is processing. */

  int curr_y,            /* Current Y position that this slave is
                            processing. */

  int special_row,       /* Row that we want from other slaves. This can
                            be 1, 3 or 0 (if there is no need). */

  bool* is_demanded,     /* This boolean keeps track of whether a slave
                            already sent its message demanding point
                            data. This is just to prevent deadlock. */
```

```
  bool* should_continue, /* Continue here is the actual `continue`
                            statement. Indicates whether caller method
                            should use continue afterwards. */

  int* total,            /* Result of smoothing process. */

  int* rank               /* Rank of the slave calling this method */
) {

  /* Validation */
  if(special_row != 0 && special_row != 1 && special_row != 3) {
    debug_1("WRONG SPECIAL WRONG IS PASSED!", rank);
    exit(0);
  }

  int *row_1, *row_2, *row_3; /* These rows will be used for smoothing. */

  bool demand_result; /* Here, this variable is very important. `demand_point_data`
                         returns false if some other slave requested data from us
                         while we want to demand data. So we return from that
                         stage and fulfill that slave's demand. */

  bool used_demand = (special_row != 0);  /* If special row is 1 or 3, we should
                                             demand some data from other slaves. */


  if(special_row == 1) {
    row_1 = malloc(sizeof(int) * 3);
    demand_result = demand_point_data(&curr_x, *rank, row_1, 'u', is_demanded);
    row_3 = *(slice_matrix + curr_y + 1) + curr_x - 1;
  } else if(special_row == 3) {
    row_3 = malloc(sizeof(int) * 3);
    demand_result = demand_point_data(&curr_x, *rank, row_3, 'l', is_demanded);
    row_1 = *(slice_matrix + curr_y - 1) + curr_x - 1;
  } else {
    row_1 = *(slice_matrix + curr_y - 1) + curr_x - 1;
    row_3 = *(slice_matrix + curr_y + 1) + curr_x - 1;
  }


  row_2 = *(slice_matrix + curr_y) + curr_x - 1;

  if(used_demand && demand_result == false) {
    *should_continue = true;
  } else {
    *should_continue = false;
    *is_demanded = false;
  }

  *total = smoothen_point(row_1, row_2, row_3, rank);
  if(special_row == 1) {
    free(row_1);
  } else if(special_row == 3) {
    free(row_3);
  }
}

/*********************************************************************
 * This is the worst code I have written in my life. Don't ever
 * touch this function, here is what it does:
 *
 * It takes the current position and smoothened slice, and processes them
 * for thresholding.
 *********************************************************************/
void process_rows_for_thresholding(
  int** smoothened_slice,    /* Matrix that this slave is responsible of. */

  int curr_x,            /* Current X position that this slave
                            is processing. */

  int curr_y,            /* Current Y position that this slave is
                            processing. */
```

```
  int special_row,       /* Row that we want from other slaves. This can
                            be 1, 3 or 0 (if there is no need). */

  bool* is_demanded,     /* This boolean keeps track of whether a slave
                            already sent its message demanding point
                            data. This is just to prevent deadlock. */

  bool* should_continue, /* Continue here is the actual `continue`
                            statement. Indicates whether caller method
                            should use continue afterwards. */

  int* total,            /* Result of smoothing process. */

  int* rank               /* Rank of the slave calling this method */
) {

  /* Validation */
  if(special_row != 0 && special_row != 1 && special_row != 3) {
    debug_1("WRONG SPECIAL WRONG IS PASSED!", rank);
    exit(0);
  }

  int *received; /* These rows will be used for smoothing. */
  int *row_1 = malloc(sizeof(int) * 3);
  int *row_2 = malloc(sizeof(int) * 3);
  int *row_3 = malloc(sizeof(int) * 3);

  bool demand_result; /* Here, this variable is very important. `demand_point_data`
                         returns false if some other slave requested data from us
                         while we want to demand data. So we return from that
                         stage and fulfill that slave's demand. */

  bool used_demand = (special_row != 0);  /* If special row is 1 or 3, we should
                                             demand some data from other slaves. */

  if(special_row == 1) { /* All rows takes first address from demand result */
    received = malloc(sizeof(int) * 3);
    demand_result = demand_point_data(&curr_x, *rank, received, 'u', is_demanded);
    *(row_1 + 0) = *(received + 0);
    *(row_2 + 0) = *(received + 1);
    *(row_3 + 0) = *(received + 2);

    *(row_1 + 1) = *(*(smoothened_slice + curr_x - 1) + curr_y);
    *(row_2 + 1) = *(*(smoothened_slice + curr_x) + curr_y);
    *(row_3 + 1) = *(*(smoothened_slice + curr_x + 1) + curr_y);

    *(row_1 + 2) = *(*(smoothened_slice + curr_x - 1) + curr_y + 1);
    *(row_2 + 2) = *(*(smoothened_slice + curr_x) + curr_y + 1);
    *(row_3 + 2) = *(*(smoothened_slice + curr_x + 1) + curr_y + 1);

  } else if(special_row == 3) { /* All rows takes last address from demand result */
    received = malloc(sizeof(int) * 3);
    demand_result = demand_point_data(&curr_x, *rank, received, 'l', is_demanded);

    *(row_1 + 0) = *(*(smoothened_slice + curr_x - 1) + curr_y - 1);
    *(row_2 + 0) = *(*(smoothened_slice + curr_x) + curr_y - 1);
    *(row_3 + 0) = *(*(smoothened_slice + curr_x + 1) + curr_y - 1);

    *(row_1 + 1) = *(*(smoothened_slice + curr_x - 1) + curr_y);
    *(row_2 + 1) = *(*(smoothened_slice + curr_x) + curr_y);
    *(row_3 + 1) = *(*(smoothened_slice + curr_x + 1) + curr_y);

    *(row_1 + 2) = *(received + 0);
    *(row_2 + 2) = *(received + 1);
    *(row_3 + 2) = *(received + 2);
  } else {
    *(row_1 + 0) = *(*(smoothened_slice + curr_x - 1) + curr_y - 1);
    *(row_2 + 0) = *(*(smoothened_slice + curr_x) + curr_y - 1);
    *(row_3 + 0) = *(*(smoothened_slice + curr_x + 1) + curr_y - 1);

    *(row_1 + 1) = *(*(smoothened_slice + curr_x - 1) + curr_y);
```

```
    *(row_2 + 1) = *(*(smoothened_slice + curr_x) + curr_y);
    *(row_3 + 1) = *(*(smoothened_slice + curr_x + 1) + curr_y);

    *(row_1 + 2) = *(*(smoothened_slice + curr_x - 1) + curr_y + 1);
    *(row_2 + 2) = *(*(smoothened_slice + curr_x) + curr_y + 1);
    *(row_3 + 2) = *(*(smoothened_slice + curr_x + 1) + curr_y + 1);
  }

  *total = threshold_point(row_1, row_2, row_3, rank);

  if(used_demand && demand_result == false) {
    *should_continue = true;
  } else {
    *should_continue = false;
    *is_demanded = false;
  }

}

/*********************************************************************
 * Slave process
 *********************************************************************/
void slave() {
  MPI_Status status;
  int rank, slice_size, slice_type, row_count, col_count;
  int* slice;         // Slice that this slave is going to work on. It's an array.
  int** slice_matrix; // We are receiving slice as an array but then deserializing it to
matrix.
  int i_am_alone;

  // Setting rank
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  // Receiving information that if I am alone or not
  MPI_Recv(&i_am_alone, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

  // Receiving slice size
  MPI_Recv(&slice_size, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

  // Receiving slice on our allocated slice space.
  //
  // Assume a 6x6 image seperated into 3 slices:
  // 1 slice has 12 points and we are sending 13 integers being first 12 is
  // slice data, and 13th is column count which is 6 since image is 6x6.
  slice = (int*) malloc((slice_size + 1) * sizeof(int)); // Allocating space for slave's
slice
  MPI_Recv(slice, slice_size + 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
  col_count = *(slice + slice_size);
  row_count = slice_size/col_count;

  // Receiving type of slice which indicates whether it's at the top, bottom or middle
  MPI_Recv(&slice_type, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

  slice_matrix = deserialize_slice(slice, row_count, col_count);

  int** smoothened_slice = util_alloc_matrix(row_count, col_count);
  int** thresholded_slice = util_alloc_matrix(row_count, col_count);

  // Starting smoothing process. After smoothing each point, slave is checking whether
  // there is a message from other slaves
  int start_x, start_y;                  // Starting point of slice processing
  int end_x, end_y;                      // Ending point of slice processing (exclusive)
  int stage = STAGE_SMOOTHING;
  util_decide_starting_position(slice_type, row_count, col_count, stage, i_am_alone,
&start_x, &start_y, &end_x, &end_y);

  // Starting smoothing job
  int curr_x = start_x, curr_y = start_y; // Current point of slice processing
  bool job_finished = false;
  int message_exists;
  bool is_demanded = false;
  bool someone_need_me;
```

```
  bool should_continue;

  for(;;) {
    /* If any of other slave wants demands a point, they send messages.
     * Here, we check whether another slave demands point data from us. */
    MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &message_exists, &status);
    someone_need_me = status.MPI_TAG == DEMAND_DATA_FROM_UPPER_SLICE_TAG
                   || status.MPI_TAG == DEMAND_DATA_FROM_LOWER_SLICE_TAG;

    if(message_exists) {
      int message_size, message_data, message_source;
      MPI_Get_count(&status, MPI_INT, &message_size);
      MPI_Recv(&message_data, message_size, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, &status);
      message_source = status.MPI_SOURCE;

      if(status.MPI_TAG == FINISH_SMOOTHING_TAG) {
        job_finished = true;
        continue;
      } else if(status.MPI_TAG == START_THRESHOLDING_TAG) {
        stage = STAGE_THRESHOLDING;
        util_decide_starting_position(slice_type, row_count, col_count, stage,
i_am_alone, &start_x, &start_y, &end_x, &end_y);
        curr_x = start_x;
        curr_y = start_y;
        job_finished = false; // Work work work work work
        continue;
      } else if(status.MPI_TAG == FINISH_THRESHOLDING_TAG) {
        for(int row = 0; row < row_count; row++) {
          free(*(slice_matrix + row));
        }
        free(slice_matrix);
        return;
      } else if(someone_need_me) {
        int x_index = message_data;
        int demander_source = message_source;

        int* points;
        if(stage == STAGE_SMOOTHING) {
          points = util_prepare_points_for_demander(slice_matrix, x_index,
status.MPI_TAG, end_y, stage);
        } else {
          points = util_prepare_points_for_demander(smoothened_slice, x_index,
status.MPI_TAG, end_y, stage);
        }
        /* Sending point data */
        MPI_Send(points, 3, MPI_INT, demander_source, (50 + demander_source),
MPI_COMM_WORLD);
        /* free(points); */
      } else {
        debug_1("Received an unidentified message, there is something wrong!", &rank);
        exit(0);
      }
    } else { // Do your own job
      if(job_finished) continue;
      if(stage == STAGE_SMOOTHING) {
        /*
         * Checking boundary conditions where we need to rearrange position
         */
        if(curr_x == end_x) {
          if(curr_y == end_y - 1) {
            /* Informing master that my job is done */
            int slice_size = row_count * col_count;
            MPI_Send(&slice_size, 1, MPI_INT, 0, SMOOTHING_DONE_TAG, MPI_COMM_WORLD);
            int* slice = serialize_slice(smoothened_slice, row_count, col_count);
            MPI_Send(slice, row_count * col_count, MPI_INT, 0,
FOLLOWING_SMOOTHING_DONE_TAG, MPI_COMM_WORLD);
            job_finished = true;
          } else {
            curr_x = 1;
            curr_y++;
          }
```

```
        } else {  // Do your own job

          bool is_low_row = curr_y == end_y - 1;
          bool is_high_row = curr_y == start_y;
          int total, special_row;

          if(i_am_alone) {
            special_row = 0;
          } else {
            special_row = util_determine_special_row(slice_type, is_low_row,
is_high_row);
          }

          process_rows_for_smoothing(slice_matrix, curr_x, curr_y, special_row,
                                     &is_demanded, &should_continue, &total, &rank);

          if(should_continue) {
            continue;
          }
          *(*(smoothened_slice + curr_x) + curr_y) = total;
          curr_x++;
        }
      } else if(stage == STAGE_THRESHOLDING) {
        if(curr_x == end_x) {
          if(curr_y == end_y - 1) {
            job_finished = true;
            /* Informing master that my job is done */
            int slice_size = row_count * col_count;
            MPI_Send(&slice_size, 1, MPI_INT, 0, THRESHOLDING_DONE_TAG, MPI_COMM_WORLD);
            int* slice = serialize_slice(thresholded_slice, row_count, col_count);
            MPI_Send(slice, row_count * col_count, MPI_INT, 0,
FOLLOWING_THRESHOLDING_DONE_TAG, MPI_COMM_WORLD);
          } else {
            curr_x = 2;
            curr_y++;
          }
        } else {
          bool is_low_row = curr_y == end_y - 1;
          bool is_high_row = curr_y == start_y;
          int total;

          int special_row;
          if(i_am_alone) {
            special_row = 0;
          } else {
            special_row = util_determine_special_row(slice_type, is_low_row,
is_high_row);
          }


          process_rows_for_thresholding(smoothened_slice, curr_x, curr_y, special_row,
                                        &is_demanded, &should_continue, &total, &rank);

          if(should_continue) {
            continue;
          }
          *(*(thresholded_slice + curr_x) + curr_y) = total;

          curr_x++;
        }
      }
    }
  }
}

int main(int argc, char* argv[]) {
  int rank;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if(argc != 4) {
```

```
    if(rank == 0)
      printf("Wrong input format, it should be:\nmpiexec -n <n_of_processors> <program>
<in_file> <out_file> <threshold>\n");
    exit(0);
  }

  INPUT_FILENAME = argv[1];
  OUTPUT_FILENAME = argv[2];
  THRESHOLD = atoi(argv[3]);

  if(rank == 0) {
    master();
  } else {
    slave();
    printf("Slave is finished.\n");
  }

  printf("Finalizing %d\n", rank);
  MPI_Finalize();
}
```