CmpE 300 - Analysis of Algorithms

Fall 2018

Programming Project

-

Image denoising with the Ising model using Metropolis- Hastings algorithm

Onur Kılıçoğlu

2015400012

Submission date: 26.12.2018

1) Introduction:

In this project, the goal is to do parallel programming with C/C++ using MPI library. It is expected to implement a parallel algorithm for image denoising with the Ising model using Metropolis- Hastings algorithm. A noisy image will be given in a square matrix of size (n x n). Each index in the matrix denotes a pixel of the image. Black pixels are denoted with -1 and white pixels are denoted with +1. The algorithm selects a random pixel from the image and suggests a pixel flip, changing a pixel from white to black or vice versa. After suggesting a pixel flip, the acceptance probability of that flip is calculated. The pixel is flipped with the probability of the acceptance probability. The image should be denoised in a parallel approach by using the algorithm in an iterative loop. Two different parallel approaches are given. In first approach each processor is responsible from a group of n/p adjacent rows. Each processor works on (n/p × n) pixels and these pixels are stored locally by the processor. When a pixel is inspected, the processor should inspect all of its neighbors. When the pixel is not on the boundary of two adjacent processors, information about the neighbor pixels are present to the processor. When a pixel on the boundary is inspected, the processor needs to obtain information from the adjacent processor. In the second approach, the grid is divided into (n/p × n/p) blocks and each process is responsible from a block. In this approach, each process should communicate with more than 1 process to update adjacent processes. To solve the problem, I have implemented both approaches. In both approaches, at the beginning of the program master processor sends the blocks of the image to the related slave processor. Before starting to apply the algorithm, each slave processor sends all bordering pixels to the related adjacent slave processors. Then the slave processors start to apply the algorithm to their blocks. In each iteration, slave processors send 1 pixel to the adjacent slave with the relevant index. In the first approach, a slave processor sends a pixel and an index to 2 adjacent processors (if exist), one for the block on the top and one for the block on the bottom. In the second approach, a slave processor sends a pixel and an index to 4 adjacent processors (if exist), one for the block on the top, one for the block on the bottom, one for the block on the left and one for the block on the right. A slave processor also sends its corner pixels to the adjacent processors which needs them. After running a total of 500,000 iterations in parallel processes, an image of size 200x200 can be denoised.

2) Program Interface:

This program is implemented to run on UNIX systems. To be able to run the program user should install OpenMPI version 1.4.4. The program is run through a command line interface such as bash. To start the program first the user should compile the program. To compile the program, open the terminal and change directory to the source code's directory. Then to compile the program run the command:

mpicc -g main.c -o pr

To compile the second approach program run the command:

mpicc -g task2.c -o pr

To run the program, run the command:

mpiexec -n <num-of-process> ./pr <input-file > <output-file> <beta-value> <pi-value>

Parameters:

<num-of-processor>: Number of processes to run the program in parallel.

<input-file>: Path to the text file including the matrix of the image of size 200x200 to be denoised by the program.

<output-file>: Path to the text file to write the output of the program which is the denoised version of image.

<beta-value>: β parameter from the Ising Model which is a number between 0 and 1.

<pi-value>: π parameter from the Ising Model which is a number between 0 and 1.

The program terminates itself after execution.

3) Input and Output:

The program takes 5 inputs: Input file, path to output file, beta value, pi value and number of processes.

- Input file: Should contain the binary matrix of the image of size 200x200 to be denoised by the program. The input file should contain 200 hundred lines with 200 integer values separated with whitespace character. The integer values in the file should be either +1 denoting the color white or -1 denoting the color black.
- Path to output file: Should contain the full or relative path of the output file to write the output of the program.
- Beta value: Should be a number between 0 and 1. This parameter comes from the Ising model that is used in the algorithm. This parameter symbolizes the relation

between neighboring pixels. Higher values of beta means that neighboring pixels are more likely to be the same color.

- Pi value: Should be a number between 0 and 1. This parameter comes from the Ising model that is used in the algorithm. This parameter symbolizes the proportion of noisy pixels of the input image.
- Number of processes: Should be a positive integer greater than 1. This number denotes the number of processes to create, to run the program in parallel. One processor is the master process and the other processors are slave processes. To run the first approach number of slave processes should be sub-multiple of 200 therefore number of processes minus one should be sub-multiple of 200. To run the second approach square root of the number of slave processes should be sub-multiple of 200 therefore square root of -number of processes minus one- should be sub-multiple of 200.

The program gives single output. The output of the program is written to the file at path to the output file given as input. The output contains the binary matrix of the denoised image of size 200x200. The output contains 200 lines with 200 hundred integers separated with whitespace characters. The integer values in the file are either +1 denoting the color white or -1 denoting the color black.

4) Program Execution:

Run the program with your parameters. After you run the program it reads the input file which is the matrix of your image of size (200x200) in text. The program divides the image into blocks and applies the algorithm in parallel processes to these blocks to denoise it. After the execution of the algorithm the program writes the denoised image's matrix of size (200x200) in text to the output file in the directory given.

5) Program Structure – First Approach:

The program only has the main function. It starts with initializing random number generator and the MPI environment. Then input from the run command is read. The program checks for illegal input and terminate if inputs are incorrect with printing an error message to the console. Then gamma value which is used to calculate the acceptance probability is calculated. After this calculation the process with rank 0 which is the master process starts executing separate instructions compared to other processes with rank 1 or higher which are slave processes. From that point on master process allocates a multidimensional array with size 200x200. Then master process reads input from the input file to the array. Then the master process sends the blocks of size (200/p x 200) to slave processes where p denotes number of slave processes. Each block contains 200/p rows of

the multidimensional image array. Slave process with rank r gets the r^{th} block of the image. After sending all the blocks to the slave processes the master process starts to wait to receive processed blocks. The slave processors allocate 2 matrices Z and X of size (200/p x 200) which is the block size. Then each slave process receives their block and initialize Z and X. After the initialization each process send bordering rows, top row and bottom row, to the adjacent slave processors if adjacent slave processes exist. Then the slave processes create two arrays with to elements to send and receive information during execution. After this, the execution of the algorithm starts. Denoising algorithm is executed in a for loop. In each iteration, each slave process chooses a random pixel and calculate the acceptance probability. To calculate the acceptance probability, we sum the product of the random pixel's value with its neighbor pixels' values. Nested if statements are used to calculate the sum in different conditions. First if statement checks if the random pixel is on the border rows such as top row or bottom row. If the random pixels are on border rows the first nested if statement checks if the slave process can use the data that came from the adjacent slave processes. For example, the slave process with rank 1 can use the row that comes from process with rank 2 however since the process has no adjacent process on the top it cannot use the data from the top block. The second nested if statement checks if the pixel is on a border column which means that the pixel does not have any column on its left if it is on the leftmost column or the pixel does not have any column on its right if it is on the rightmost column. After the sum is calculated we calculate the acceptance probability of flipping the pixel. And we flip the pixel with the probability of the acceptance probability. At the end of each iteration, the slave processes send 1 pixel to the adjacent slave with the relevant index. For example, in first approach the slave processor changes Z[i][j] using the algorithm and sends pixel Z[0][j] to the processor processing the block on top with the index j. If i = 0 then a pixel flip is sent to the relevant processor. If index i is not equal to "0" then the send is unnecessary in the way that it does not bring any new information. However, with this unnecessary send all processors synchronize in each iterative loop. After completing all iterations, slave processes send the denoised version of blocks to the master process and terminate. After the master processor receives all the blocks it opens the output file and writes the matrix of denoised image. Then the whole program terminates.

6) Program Structure – Second Approach:

The program has 4 helper functions and a main function. First function "send" takes a block of size 200/n where n is the square root of the number of slave processes, the

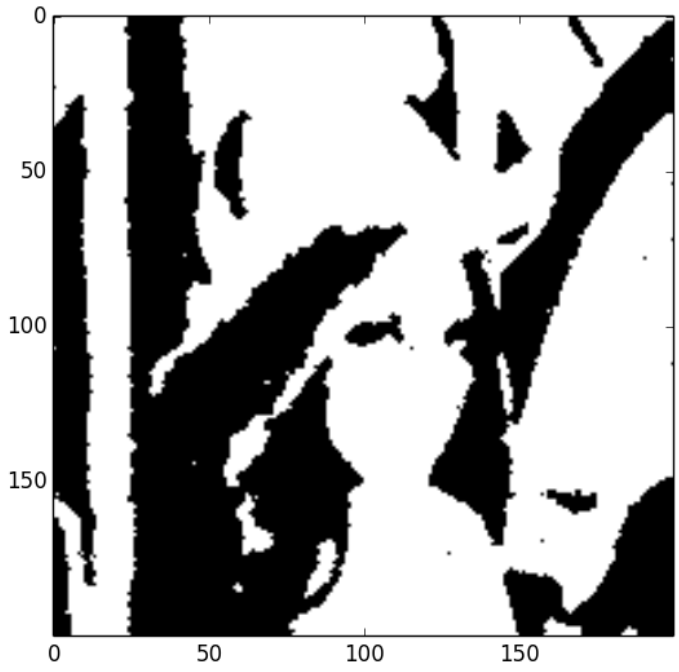
image row size N, n which is the square root of the number of slave processes, rank of the process “world_rank” and the number of total processes created “world_size”. Then the function sends all the bordering pixels of the block that will be used by adjacent processes to the corresponding process. Second function “receive” takes pointers to the variables that are used to store incoming information from adjacent processes “top, bottom, left, right, topLeft, topRight, bottomLeft, bottomRight ”, the image row size N, n which is the square root of the number of slave processes, rank of the process “world_rank” and the number of total processes created “world_size”. Then the function receives all the bordering pixels of the block that will be used in calculation from corresponding adjacent processes of the process. Third (send2) and fourth (receive2) functions are doing fundamentally the same things as send and receive functions, however with one difference. Send and receive functions send and receive all bordering pixels to the adjacent processes on the other hand “send2” and “receive2” functions send and receive only the pixels that are possibly changed in the iteration of the algorithm. To do this “send2” function takes 2 more parameters for the index of the random pixel that is changed. Function “send2” sends the item that may possibly be changed with the corresponding index. For example, in second approach the slave processor changes Z[i][j] using the algorithm and sends pixel Z[i][0] to the processor processing the block on left with the index i. If j = 0 then a pixel flip is sent to the relevant processor. If index j is not equal to “0” then the send is unnecessary in the way that it does not bring any new information. However, with this unnecessary send all processors synchronize in each iterative loop. Main functions in first and second approach runs in the same manner. The first difference from the first approach is that sends and receives are done via the function calls that are mentioned above. The second difference from the first approach occurs when calculating the acceptance probability. We sum the product of the random pixel’s value with its neighbor pixels’ values and since there are more options for bordering pixels those values are included in to the sum in calculation part.

7) Examples

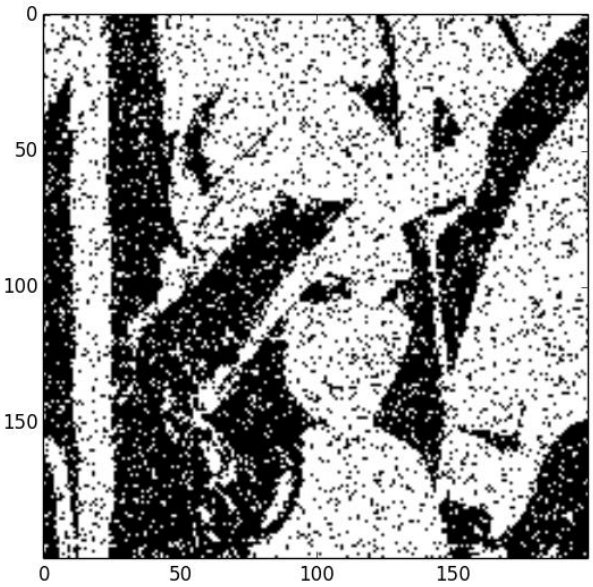
Original Image (Lena):

Noisy Image:

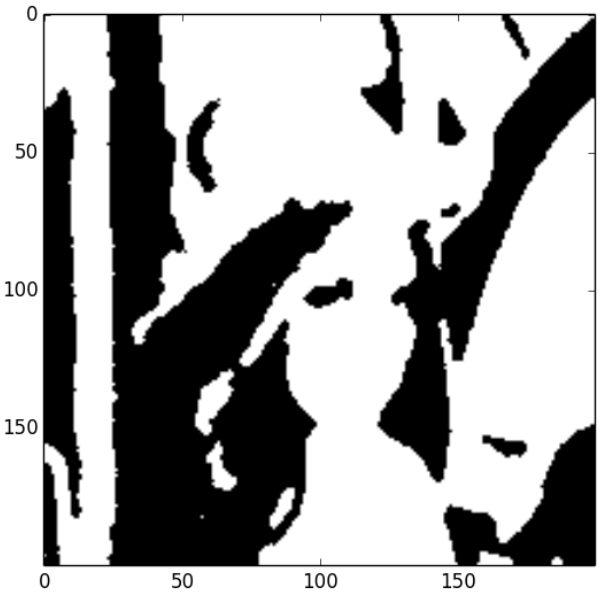
Denoised image (with $\beta = 0.5$ and $\pi = 0.1$):



Denoised image (with $\beta = 0.1$ and $\pi = 0.1$):



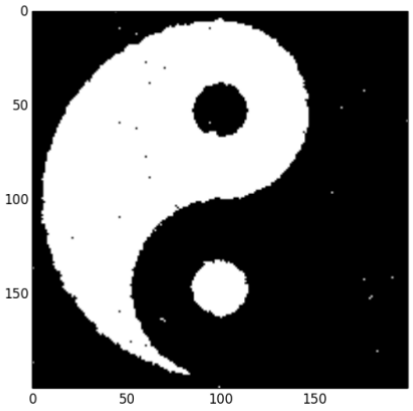
Denoised image (with $\beta = 0.9$ and $\pi = 0.1$):



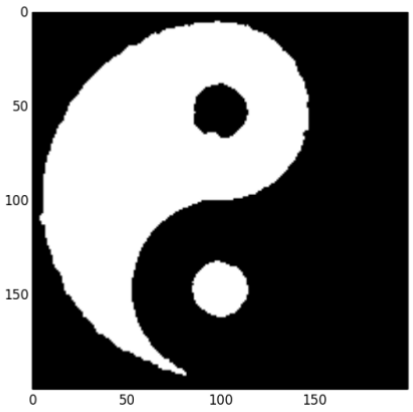
Original image (Yin Yang):

Noisy image:

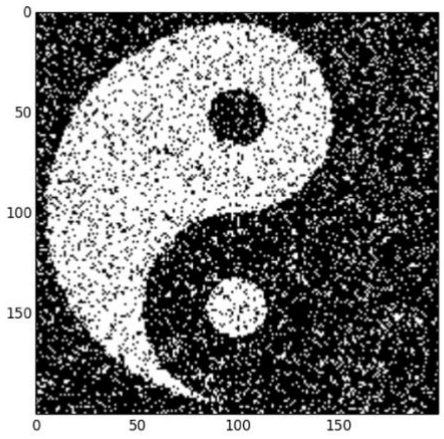
Denoised image (with $\beta = 0.5$ and $\pi = 0.1$):



Denoised image (with $\beta = 0.9$ and $\pi = 0.15$):



Denoised image (with $\beta = 0.1$ and $\pi = 0.15$):



As it can be observed from the images, beta value should be selected carefully. Note that with low beta values the model assumes that the connection between neighboring pixels are weak therefore it fails to denoise the program in both examples.

8) Improvements and Extensions:

A weakness of the program is that when denoising images of size 200x200 some details in images are seen as noise because the specific detail in the image can only be represented with a small number of pixels. It is possible to observe this problem in Lena image example. Details showing the mouth of Lena turned into white because the model treated those pixels as a noise in the image. Possible improvement to the problem in general is allowing the images of all sizes. Now the program can only denoise 200x200 images, but the program can be configured to denoise the images of all sizes including non-square images like 680x1090.

9) Difficulties Encountered:

Getting used to the parallel programming environment was one of the hardest challenges in the project. Debugging the program in parallel programming became a significant issue during the project since the possible sources of errors are more than one single process running the program. Synchronizing the program and deadlock prevention was also a challenge. The program was taking too long to execute because of the message passing overhead. Optimizing message passing definitely was an important issue encountered that has been handled.

10) Conclusion:

In conclusion, I have implemented 2 different working approaches for image denoising in parallel using the given algorithm and MPI libraries in C. Both of the approaches can be

compiled and tested with different images and parameters. For this specific project the first approach performs better since it sends at most 2 messages and receives 2 messages containing 2 integer values in each iteration. After implementing the project, I got a good understanding on parallel programming at general.

11) Appendices:

A. First Task's Code:

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <string.h>
int main(int argc, char** argv) {
  /* Intializes random number generator */
  srand(time(NULL));
        // Initialize the MPI environment
        MPI_Init(NULL, NULL);
        // Find out rank, size
        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
        int world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);
        //N is the row and column size of image matrix
        int N = 200;
        double beta,pi,gamma;
        //Read inputs from command line
        beta = atof(argv[3]);
        pi = atof(argv[4]);
        //Checking Input values
        if(beta < 0 || beta > 1){
                printf("Beta value should be between 0 and 1\n");
                exit(1);
        }
        if(pi <= 0 ||pi >= 1){
                printf("Pi value should be between 0 and 1\n");
                exit(1);
        }
        // Gamma is initialized to use in the algorithm
        gamma = 1.0/2*log((1-pi)/pi);
        // Iteration count for the algorithm
        int iterationCount = 500000/(world_size -1);
        if (world_rank == 0){
                // Allocating memory for picture
                // Picture holds the image matrix
                int ** picture = (int **)malloc(sizeof(int *) * N);
                for(int i=0;i<N;i++)
                        picture[i] = (int *)malloc(sizeof(int) * N);
                //Getting Input
                FILE *fptr;
                fptr = fopen(argv[1],"rb");
                if (fptr == NULL){
                   printf("Error opening file!\n");
                   exit(1);
                }
                char * line = NULL;
           size_t len = 0;
                ssize_t read;
                for(int i=0;i<N;i++){
                        read = getline(&line, &len, fptr);
                        char* token = strtok(line, " ");
                        picture[i][0] =atoi(token);
                        for(int j=1;j < N ;j++) {
                        token = strtok(NULL, " ");
                        picture[i][j] = atoi(token);
                        }
                }
                free(line);
                fclose(fptr);
                //Sending blocks of size (N/p)x200 of pictures to processors
                for(int i = 0; i < world_size-1; i++)
```

```
                        for(int j = 0; j < N/(world_size - 1); j++)
                                MPI_Send(picture[i*N/(world_size - 1)+j], N, MPI_INT, i+1, 0,
MPI_COMM_WORLD);
                //Recieving blocks of size (N/p)x200 of pictures to processors
                for(int i = 0; i < world_size-1; i++)
                        for(int j = 0; j < N/(world_size - 1); j++)
                                MPI_Recv(picture[N/(world_size - 1)*i+j], N, MPI_INT, i+1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

                // Writing denoised image to the output file
                FILE *f = fopen(argv[2], "w");
                if (f == NULL){
                   printf("Error opening file!\n");
                   exit(1);
                }
                // Writing output and deallocating memory
                for(int i=0;i < N;i++){
                        for(int j=0; j < N; j++)
                                fprintf(f, "%d ", picture[i][j]);
                        fprintf(f, "\n");
                        free(picture[i]);
                }
                fclose(f);
                free(picture);
        }
        else if (world_rank > 0) {
                //Allocating memory for X of the picture used by processor
                //Allocating memory for Z of the picture used by processor
                int ** X = (int **)malloc(sizeof(int*) * N/(world_size - 1));
                int ** Z = (int **)malloc(sizeof(int*) * N/(world_size - 1));
                for(int i=0;i<N/(world_size - 1);i++){
                        X[i] = (int *)malloc(sizeof(int) * N);
                        Z[i] = (int *)malloc(sizeof(int) * N);
                }
                //Allocating memory for getting the bordering pixels of the block at the top
                //Allocating memory for getting the bordering pixels of the block at the bottom
                int* top = (int *)malloc(sizeof(int) * N);
                int* bottom = (int *)malloc(sizeof(int) * N);
                //Processors receive initial block
                for(int i=0;i < N/(world_size - 1);i++)
                        MPI_Recv(X[i], N, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                //Initializing Z(0)
                for(int i = 0;i < N/(world_size - 1); i++){
                        for (int j = 0; j < N; j++)
                                Z[i][j] = X[i][j];
                }
                //Sending top row
                if(world_rank - 1 > 0)
                        MPI_Send(Z[0], N, MPI_INT, world_rank-1, 0, MPI_COMM_WORLD);
                //Sending bottom row
                if(world_rank + 1 < world_size)
                        MPI_Send(Z[N/(world_size - 1) - 1], N, MPI_INT, world_rank+1, 1, MPI_COMM_WORLD);
                //Receiving top row
                if(world_rank - 1 > 0)
                        MPI_Recv(top, N, MPI_INT, world_rank-1, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
                //Receiving bottom row
                if(world_rank + 1 < world_size)
                        MPI_Recv(bottom, N, MPI_INT, world_rank+1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
                int toSend[2];
                int toReceive[2];
                //Calculating acceptance probability
                double alpha;
                double sum;
                for(int it=0;it < iterationCount; it++){
                        //Selecting a random pixel
                        int i = rand()%(N/(world_size - 1));
                        int j = rand()%N;
                        alpha = -2*gamma*Z[i][j]*X[i][j];
                        // Calculating sum of products of neighboring pixels with the random pixel
                        sum = 0;
                        // Checking if the pixel is at the top row
                        if(i == 0){
                                // The block is on the top therefore no pixel at the top border
                                if(world_rank == 1){
                                        // The pixel is on the leftmost column
                                        if(j == 0){
                                                sum += Z[i][j]*Z[i+1][j];
```

```
                                        sum += Z[i][j]*Z[i+1][j+1];
                                        sum += Z[i][j]*Z[i][j+1];
                                }
                                // The pixel is on the rightmost column
                                else if(j == N-1){
                                        sum += Z[i][j]*Z[i][j-1];
                                        sum += Z[i][j]*Z[i+1][j-1];
                                        sum += Z[i][j]*Z[i+1][j];
                                }
                                // The pixel is in the middle and has neighbors on its right and left
                                else{
                                        sum += Z[i][j]*Z[i][j-1];
                                        sum += Z[i][j]*Z[i+1][j-1];
                                        sum += Z[i][j]*Z[i+1][j];
                                        sum += Z[i][j]*Z[i+1][j+1];
                                        sum += Z[i][j]*Z[i][j+1];
                                }
                        }
                        // The block is not on the top therefore pixels at the top border can be used
                        else{
                                // The pixel is on the leftmost column
                                if(j == 0){
                                        sum += Z[i][j]*Z[i+1][j];
                                        sum += Z[i][j]*Z[i+1][j+1];
                                        sum += Z[i][j]*Z[i][j+1];
                                        sum += Z[i][j]*top[j];
                                        sum += Z[i][j]*top[j+1];

                                }
                                // The pixel is on the rightmost column
                                else if(j == N-1){
                                        sum += Z[i][j]*Z[i][j-1];
                                        sum += Z[i][j]*Z[i+1][j-1];
                                        sum += Z[i][j]*Z[i+1][j];
                                        sum += Z[i][j]*top[j];
                                        sum += Z[i][j]*top[j-1];
                                }
                                // The pixel is in the middle and has neighbors on its right and left
                                else{
                                        sum += Z[i][j]*Z[i][j-1];
                                        sum += Z[i][j]*Z[i+1][j-1];
                                        sum += Z[i][j]*Z[i+1][j];
                                        sum += Z[i][j]*Z[i+1][j+1];
                                        sum += Z[i][j]*Z[i][j+1];
                                        sum += Z[i][j]*top[j-1];
                                        sum += Z[i][j]*top[j];
                                        sum += Z[i][j]*top[j+1];
                                }
                        }

                }
                // Checking if the pixel is at the bottom row
                else if(i == N/(world_size - 1) -1){
                        // The block is at the bottom therefore no pixel at the bottom border
                        if(world_rank == world_size-1){
                                // The pixel is on the leftmost column
                                if(j == 0){
                                        sum += Z[i][j]*Z[i-1][j];
                                        sum += Z[i][j]*Z[i-1][j+1];
                                        sum += Z[i][j]*Z[i][j+1];
                                }
                                // The pixel is on the rightmost column
                                else if(j == N-1){
                                        sum += Z[i][j]*Z[i][j-1];
                                        sum += Z[i][j]*Z[i-1][j-1];
                                        sum += Z[i][j]*Z[i-1][j];
                                }
                                // The pixel is in the middle and has neighbors on its right and left
                                else{
                                        sum += Z[i][j]*Z[i][j-1];
                                        sum += Z[i][j]*Z[i-1][j-1];
                                        sum += Z[i][j]*Z[i-1][j];
                                        sum += Z[i][j]*Z[i-1][j+1];
                                        sum += Z[i][j]*Z[i][j+1];
                                }
                        }
                        // The block is not at the bottom therefore pixels at the bottom border can be used
                        else{
                                // The pixel is on the leftmost column
```

```
                                        if(j == 0){
                                                sum += Z[i][j]*Z[i-1][j];
                                                sum += Z[i][j]*Z[i-1][j+1];
                                                sum += Z[i][j]*Z[i][j+1];
                                                sum += Z[i][j]*bottom[j];
                                                sum += Z[i][j]*bottom[j+1];
                                        }
                                        // The pixel is on the rightmost column
                                        else if(j == N-1){
                                                sum += Z[i][j]*Z[i][j-1];
                                                sum += Z[i][j]*Z[i-1][j-1];
                                                sum += Z[i][j]*Z[i-1][j];
                                                sum += Z[i][j]*bottom[j-1];
                                                sum += Z[i][j]*bottom[j];
                                        }
                                        // The pixel is in the middle and has neighbors on its right and left
                                        else{
                                                sum += Z[i][j]*Z[i][j-1];
                                                sum += Z[i][j]*Z[i-1][j-1];
                                                sum += Z[i][j]*Z[i-1][j];
                                                sum += Z[i][j]*Z[i-1][j+1];
                                                sum += Z[i][j]*Z[i][j+1];
                                                sum += Z[i][j]*bottom[j-1];
                                                sum += Z[i][j]*bottom[j];
                                                sum += Z[i][j]*bottom[j+1];
                                        }
                                }
                        }
                        // The pixel is not on the border of an adjacent block
                        else{
                                // The pixel is on the leftmost column
                                if(j == 0){
                                        sum += Z[i][j]*Z[i-1][j];
                                        sum += Z[i][j]*Z[i-1][j+1];
                                        sum += Z[i][j]*Z[i][j+1];
                                        sum += Z[i][j]*Z[i+1][j];
                                        sum += Z[i][j]*Z[i+1][j+1];
                                }
                                // The pixel is on the rightmost column
                                else if(j == N-1){
                                        sum += Z[i][j]*Z[i][j-1];
                                        sum += Z[i][j]*Z[i-1][j-1];
                                        sum += Z[i][j]*Z[i-1][j];
                                        sum += Z[i][j]*Z[i+1][j-1];
                                        sum += Z[i][j]*Z[i+1][j];
                                }
                                // The pixel is in the middle and has all neighbors
                                else{
                                        sum += Z[i][j]*Z[i][j-1];
                                        sum += Z[i][j]*Z[i-1][j-1];
                                        sum += Z[i][j]*Z[i-1][j];
                                        sum += Z[i][j]*Z[i-1][j+1];
                                        sum += Z[i][j]*Z[i][j+1];
                                        sum += Z[i][j]*Z[i+1][j-1];
                                        sum += Z[i][j]*Z[i+1][j];
                                        sum += Z[i][j]*Z[i+1][j+1];
                                }
                        }
                        alpha += -2 * beta * sum;
                        // Flipping the pixel with the probability of the acceptance probability
                        if(alpha > log((double)rand() / (double)RAND_MAX))
                                Z[i][j] *= -1;
                        //Sending top row
                        if(world_rank - 1 > 0){
                                toSend[0] = Z[0][j];
                                toSend[1] = j;
                                MPI_Send(&toSend, 2, MPI_INT, world_rank-1, 0, MPI_COMM_WORLD);
                        }
                        //Sending bottom row
                        if(world_rank + 1 < world_size){
                                toSend[0] = Z[N/(world_size-1)-1][j];
                                toSend[1] = j;
                                MPI_Send(&toSend, 2, MPI_INT, world_rank+1, 1, MPI_COMM_WORLD);
                        }
                        //Receiving top row
                        if(world_rank - 1 > 0){
                                MPI_Recv(&toReceive, 2, MPI_INT, world_rank-1, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
                                top[toReceive[1]] = toReceive[0];
```

```
                        }
                        //Receiving bottom row
                        if(world_rank + 1 < world_size){
                                MPI_Recv(&toReceive, 2, MPI_INT, world_rank+1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
                                bottom[toReceive[1]] = toReceive[0];
                        }
                }
                // Sending the back to the master processor
                for(int i=0;i<N/(world_size - 1);i++){
                        MPI_Send(Z[i], N, MPI_INT, 0, 0, MPI_COMM_WORLD);
                        free(X[i]);
                        free(Z[i]);
                }
                free(X);
                free(Z);
                free(top);
                free(bottom);

        }

        MPI_Finalize();
}
```

B. Second Task's Code:

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <string.h>

/* send function takes the matrix of the block of image that is processed by a block, matrix row size N,
* number of blocks in a row n, the rank of the slave processor and the world size and sends initial
* version of all the bordering pixels to the necessary processor
*/
void send(int ** Z,int N,int n,int world_rank, int world_size){
        //Send bottom row
        if(world_rank + n < world_size)
                MPI_Send(Z[N/n -1], N/n, MPI_INT, world_rank + n, 0, MPI_COMM_WORLD);
        //Send top row
        if(world_rank - n > 0)
                MPI_Send(Z[0], N/n, MPI_INT, world_rank - n, 1, MPI_COMM_WORLD);
        //Send right column;
        if(world_rank % n  != 0)
                for(int i=0; i < N/n; i++)
                        MPI_Send(&Z[i][N/n-1], 1, MPI_INT, world_rank + 1, 2, MPI_COMM_WORLD);
        //Send Left Column
        if(world_rank % n  != 1)
                for(int i=0; i < N/n; i++)
                        MPI_Send(&Z[i][0], 1, MPI_INT, world_rank - 1, 3, MPI_COMM_WORLD);
        //Send top left corner
        if(world_rank % n  != 1 && world_rank - n > 0)
                MPI_Send(&Z[0][0], 1, MPI_INT, world_rank -n - 1, 4, MPI_COMM_WORLD);
        //Send top right corner
        if(world_rank % n  != 0 && world_rank - n > 0)
                MPI_Send(&Z[0][N/n-1], 1, MPI_INT, world_rank -n + 1, 5, MPI_COMM_WORLD);
        //Send bottom left corner
        if(world_rank % n  != 1 && world_rank + n < world_size)
                MPI_Send(&Z[N/n-1][0], 1, MPI_INT, world_rank +n - 1, 6, MPI_COMM_WORLD);
        //Send bottom right corner
        if(world_rank % n  != 0 && world_rank + n < world_size)
                MPI_Send(&Z[N/n-1][N/n-1], 1, MPI_INT, world_rank +n +1, 7, MPI_COMM_WORLD);
}
/* receive function takes pointers to the arrays that hold the information about bordering pixels top,bottom,left,right,
* the pointers to the variables that hold the information about bordering pixels topLeft,topRight,bottomLeft,bottomRight,
* matrix row size N, number of blocks in a row n, the rank of the slave processor and the world size
* and receives the initial version of all bordering pixels from the necessary processor
*/
void receive(int* top,int* bottom,int* left,int* right,int* topLeft,int* topRight,int* bottomLeft,int* bottomRight,int N,int n,int
world_rank,int world_size){
        //Receive top row
        if(world_rank - n > 0)
                MPI_Recv(top, N/n, MPI_INT, world_rank - n, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        //Receive bottom row
        if(world_rank + n < world_size)
                MPI_Recv(bottom, N/n, MPI_INT, world_rank + n, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        //Receive left column;
        if(world_rank % n  != 1)
```

```
                        for(int i=0; i < N/n; i++)
                                MPI_Recv(&left[i], 1, MPI_INT, world_rank -1 , 2, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        //Receive right Column
        if(world_rank % n  != 0)
                        for(int i=0; i < N/n; i++)
                                MPI_Recv(&right[i], 1, MPI_INT, world_rank + 1, 3, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        //Receive bottom right corner
        if(world_rank % n  != 0 && world_rank + n < world_size)
                        MPI_Recv(bottomRight, 1, MPI_INT, world_rank +n+ 1, 4, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        //Receive bottom left corner
        if(world_rank % n  != 1 && world_rank + n < world_size)
                        MPI_Recv(bottomLeft, 1, MPI_INT, world_rank +n- 1, 5, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        //Receive top right corner
        if(world_rank % n  != 0 && world_rank - n > 0)
                        MPI_Recv(topRight, 1, MPI_INT, world_rank-n+1, 6, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        //Receive top left corner
        if(world_rank % n  != 1 && world_rank - n > 0)
                        MPI_Recv(topLeft, 1, MPI_INT, world_rank-n-1, 7, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

/* send2 function takes the matrix of the block of image that is processed by a block, matrix row size N,
* number of blocks in a row n, the rank of the slave processor, the world size and indices of the most recently changed pixel
* and sends the version of the possibly changed bordering pixel to the necessary processor with the index of the possible change
*/
void send2(int ** Z,int N,int n,int world_rank, int world_size,int i,int j){
        int toSend[2];
        //Send bottom row
        if(world_rank + n < world_size){
                toSend[0] = Z[N/n-1][j];
                toSend[1] = j;
                MPI_Send(&toSend, 2, MPI_INT, world_rank + n, 0, MPI_COMM_WORLD);
        }
        //Send top row
        if(world_rank - n > 0){
                toSend[0] = Z[0][j];
                toSend[1] = j;
                MPI_Send(&toSend, 2, MPI_INT, world_rank - n, 1, MPI_COMM_WORLD);
        }
        //Send right column;
        if(world_rank % n  != 0){
                toSend[0] = Z[i][N/n-1];
                toSend[1] = i;
                MPI_Send(&toSend, 1, MPI_INT, world_rank + 1, 2, MPI_COMM_WORLD);
        }
        //Send Left Column
        if(world_rank % n  != 1){
                toSend[0] = Z[i][0];
                toSend[1] = i;
                MPI_Send(&toSend, 1, MPI_INT, world_rank - 1, 3, MPI_COMM_WORLD);
        }
        //Send top left corner
        if(world_rank % n  != 1 && world_rank - n > 0)
                MPI_Send(&Z[0][0], 1, MPI_INT, world_rank -n - 1, 4, MPI_COMM_WORLD);
        //Send top right corner
        if(world_rank % n  != 0 && world_rank - n > 0)
                MPI_Send(&Z[0][N/n-1], 1, MPI_INT, world_rank -n + 1, 5, MPI_COMM_WORLD);
        //Send bottom left corner
        if(world_rank % n  != 1 && world_rank + n < world_size)
                MPI_Send(&Z[N/n-1][0], 1, MPI_INT, world_rank +n - 1, 6, MPI_COMM_WORLD);
        //Send bottom right corner
        if(world_rank % n  != 0 && world_rank + n < world_size)
                MPI_Send(&Z[N/n-1][N/n-1], 1, MPI_INT, world_rank +n +1, 7, MPI_COMM_WORLD);
}

/* receive2 function takes pointers to the arrays that hold the information about bordering pixels top,bottom,left,right,
* the pointers to the variables that hold the information about bordering pixels topLeft,topRight,bottomLeft,bottomRight,
* matrix row size N, number of blocks in a row n, the rank of the slave processor and the world size
* and receives the version of the possibly changed bordering pixel from the necessary processor with the index of the possible
change
* then changes the pixel at the index to the most recent one
*/
void receive2(int* top,int* bottom,int* left,int* right,int* topLeft,int* topRight,int* bottomLeft,int* bottomRight,int N,int n,int
world_rank,int world_size){
        int toReceive[2];
        //Receive top row
```

```
	if(world_rank - n > 0){
		MPI_Recv(&toReceive, 2, MPI_INT, world_rank - n, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
		top[toReceive[1]] =toReceive[0];
	}
	//Receive bottom row
	if(world_rank + n < world_size){
		MPI_Recv(&toReceive, 2,MPI_INT, world_rank + n, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
		bottom[toReceive[1]] =toReceive[0];
	}
	//Receive left column;
	if(world_rank % n  != 1){
		MPI_Recv(&toReceive, 2, MPI_INT, world_rank -1 , 2, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
		left[toReceive[1]] =toReceive[0];
	}
	//Receive right Column
	if(world_rank % n  != 0){
		MPI_Recv(&toReceive, 2, MPI_INT, world_rank + 1, 3, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
		right[toReceive[1]] =toReceive[0];
	}
	//Receive bottom right corner
	if(world_rank % n  != 0 && world_rank + n < world_size)
		MPI_Recv(bottomRight, 1, MPI_INT, world_rank +n+ 1, 4, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
	//Receive bottom left corner
	if(world_rank % n  != 1 && world_rank + n < world_size)
		MPI_Recv(bottomLeft, 1, MPI_INT, world_rank +n- 1, 5, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
	//Receive top right corner
	if(world_rank % n  != 0 && world_rank - n > 0)
		MPI_Recv(topRight, 1, MPI_INT, world_rank-n+1, 6, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
	//Receive top left corner
	if(world_rank % n  != 1 && world_rank - n > 0)
		MPI_Recv(topLeft, 1, MPI_INT, world_rank-n-1, 7, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
int main(int argc, char** argv) {
	// Initialize the MPI environment
	MPI_Init(NULL, NULL);
	// Find out rank, size
	int world_rank;
	MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
	int world_size;
	MPI_Comm_size(MPI_COMM_WORLD, &world_size);
	//N is the row and column size of image matrix
	int N = 200;
	double beta,pi,gamma;
	//Read inputs from command line
	beta = atof(argv[3]);
	pi = atof(argv[4]);
	int n = (int)sqrt(world_size-1);
	//Checking Input values
	if(beta < 0 || beta > 1){
		printf("Beta value should be between 0 and 1\n");
		exit(1);
	}
	if(pi <= 0 ||pi >= 1){
		printf("Pi value should be between 0 and 1\n");
		exit(1);
	}
	// Gamma is initialized to use in the algorithm
	gamma = 1.0/2*log((1-pi)/pi);
	// Iteration count for the algorithm
	int iterationCount = 500000/(world_size -1);
	if (world_rank == 0){
		// Allocating memory for picture
		// Picture contains blocks to be sent to slave processors
		int *** picture = (int ***)malloc(sizeof(int **) * (world_size-1));
		for(int i=0;i < (world_size-1);i++){
			picture[i] = (int **)malloc(sizeof(int*) * N/n);
			for(int j =0;j < N/n; j++)
				picture[i][j] = (int *)malloc(sizeof(int) * N/n);
		}
		// Getting Input
		FILE *fptr;
		fptr = fopen(argv[1],"rb");
		if (fptr == NULL){
```

```
                printf("Error opening file!\n");
                exit(1);
            }
            char * line = NULL;
        size_t len = 0;
            ssize_t read;
            for(int i=0;i<N;i++){
                read = getline(&line, &len, fptr);
                char* token = strtok(line, " ");
                // Putting each pixel of the image in the right block
                picture[i/(N/n)][i%(N/n)][0] =atoi(token);
                for(int j=1;j < N ;j++) {
                token = strtok(NULL, " ");
                picture[i/(N/n)*n+ j/(N/n)][i%(N/n)][j%(N/n)] = atoi(token);
                }
            }
            free(line);
            fclose(fptr);


            //Sending blocks of size (N/n)x(N/n) of pictures to processors
            for(int i = 0; i < world_size-1; i++)
                for(int j = 0; j < N/n; j++)
                    MPI_Send(picture[i][j], N/n, MPI_INT, i+1, 0, MPI_COMM_WORLD);
            //Recieving blocks of size (N/n)x(N/n) of pictures to processors
            for(int i = 0; i < world_size-1; i++)
                for(int j = 0; j < N/n; j++)
                    MPI_Recv(picture[i][j], N/n, MPI_INT, i+1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            FILE *f = fopen(argv[2], "w");
            if (f == NULL){
                printf("Error opening file!\n");
                exit(1);
            }
            for(int i=0;i < N;i++){
                for(int j=0; j < N; j++){
                    fprintf(f, "%d ", picture[i/(N/n)*n+ j/(N/n)][i%(N/n)][j%(N/n)]);
                }
                fprintf(f, "\n");
            }
            fclose(f);
            for(int i=0;i < n;i++){
                for(int j =0;j < N/n; j++)
                    free(picture[i][j]);
                free(picture[i]);
            }
            free(picture);
        }
        else if (world_rank > 0) {
            //Initializing random number generaror.
            time_t t;
            srand((int)time(&t) % world_rank);
            //Allocating memory for X of the picture used by processor
            int ** X = (int **)malloc(sizeof(int*) * N/n);
            int ** Z = (int **)malloc(sizeof(int*) * N/n);
            for(int i=0;i<N/n;i++){
                X[i] = (int *)malloc(sizeof(int) * N/n);
                Z[i] = (int *)malloc(sizeof(int) * N/n);
            }
            int* top = (int *)malloc(sizeof(int) * N/n);
            int* bottom = (int *)malloc(sizeof(int) * N/n);
            int* right = (int *)malloc(sizeof(int) * N/n);
            int* left = (int *)malloc(sizeof(int) * N/n);
            int topLeft, topRight, bottomLeft, bottomRight;
            //Processors receive initial frame
            for(int i=0;i < N/n;i++)
                MPI_Recv(X[i], N/n, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            //Initializing Z(0)
            for(int i = 0;i < N/n; i++){
                for (int j = 0; j < N/n; j++){
                    Z[i][j] = X[i][j];
                }
            }
            if(world_size >2){
                send(Z,N,n,world_rank,world_size);

    receive(top,bottom,left,right,&topLeft,&topRight,&bottomLeft,&bottomRight,N,n,world_rank,world_size);
            }
            //Calculating acceptance probability
```

```
double alpha;
double sum;
for(int it=0;it < iterationCount; it++){
	//Selecting a random pixel
	int i = rand()%(N/n);
	int j = rand()%(N/n);
	alpha = -2*gamma*Z[i][j]*X[i][j];
	// Calculating sum of products of neighboring pixels with the random pixel
	sum = 0;
	// Checking if the pixel is at the top row
	if(i == 0){
		// The block is on the top therefore no pixel at the top border
		if(world_rank - n <= 0){
			// The pixel is on the leftmost column
			if(j == 0){
				sum += Z[i][j]*Z[i+1][j];
				sum += Z[i][j]*Z[i+1][j+1];
				sum += Z[i][j]*Z[i][j+1];
				// If the block is not on the leftmost part of picture pixels from left border can be used
				if(world_rank %n != 1){
					sum += Z[i][j]*left[i];
					sum += Z[i][j]*left[i+1];
				}
			}
			// The pixel is on the rightmost column
			else if(j == N/n-1){
				sum += Z[i][j]*Z[i][j-1];
				sum += Z[i][j]*Z[i+1][j-1];
				sum += Z[i][j]*Z[i+1][j];
				// If the block is not on the rightmost part of picture pixels from right border can be used
				if(world_rank %n != 0){
					sum += Z[i][j]*right[i];
					sum += Z[i][j]*right[i+1];
				}
			}
			// The pixel is in the middle and has neighbors on its right and left
			else{
				sum += Z[i][j]*Z[i][j-1];
				sum += Z[i][j]*Z[i+1][j-1];
				sum += Z[i][j]*Z[i+1][j];
				sum += Z[i][j]*Z[i+1][j+1];
				sum += Z[i][j]*Z[i][j+1];
			}
		}
		// The block is not on the top therefore pixel at the top border can be used
		else{
			// The pixel is on the leftmost column
			if(j == 0){
				sum += Z[i][j]*Z[i+1][j];
				sum += Z[i][j]*Z[i+1][j+1];
				sum += Z[i][j]*Z[i][j+1];
				sum += Z[i][j]*top[j];
				sum += Z[i][j]*top[j+1];
				// If the block is not on the leftmost part of picture pixels from left border can be used
				if(world_rank %n != 1){
					sum += Z[i][j]*left[i];
					sum += Z[i][j]*left[i+1];
					sum += Z[i][j]*topLeft;
				}

			}
			// The pixel is on the rightmost column
			else if(j == N/n-1){
				sum += Z[i][j]*Z[i][j-1];
				sum += Z[i][j]*Z[i+1][j-1];
				sum += Z[i][j]*Z[i+1][j];
				sum += Z[i][j]*top[j];
				sum += Z[i][j]*top[j-1];
				// If the block is not on the rightmost part of picture pixels from right border can be used
				if(world_rank %n != 0){
					sum += Z[i][j]*right[i];
					sum += Z[i][j]*right[i+1];
					sum += Z[i][j]*topRight;
				}
			}
```

```
                                        // The pixel is in the middle and has neighbors on its right and left
                                        else{
                                                sum += Z[i][j]*Z[i][j-1];
                                                sum += Z[i][j]*Z[i+1][j-1];
                                                sum += Z[i][j]*Z[i+1][j];
                                                sum += Z[i][j]*Z[i+1][j+1];
                                                sum += Z[i][j]*Z[i][j+1];
                                                sum += Z[i][j]*top[j-1];
                                                sum += Z[i][j]*top[j];
                                                sum += Z[i][j]*top[j+1];
                                        }
                                }

                        }
                        // Checking if the pixel is at the bottom row
                        else if(i == N/n -1){
                                // The block is at the bottom therefore no pixel at the bottom border
                                if(world_rank + n > world_size){
                                        // The pixel is on the leftmost column
                                        if(j == 0){
                                                sum += Z[i][j]*Z[i-1][j];
                                                sum += Z[i][j]*Z[i-1][j+1];
                                                sum += Z[i][j]*Z[i][j+1];
                                                // If the block is not on the leftmost part of picture pixels from
left border can be used
                                                if(world_rank %n != 1){
                                                        sum += Z[i][j]*left[i-1];
                                                        sum += Z[i][j]*left[i];
                                                }
                                        }
                                        // The pixel is on the rightmost column
                                        else if(j == N/n-1){
                                                sum += Z[i][j]*Z[i][j-1];
                                                sum += Z[i][j]*Z[i-1][j-1];
                                                sum += Z[i][j]*Z[i-1][j];
                                                // If the block is not on the rightmost part of picture pixels
from right border can be used
                                                if(world_rank %n != 0){
                                                        sum += Z[i][j]*right[i-1];
                                                        sum += Z[i][j]*right[i];
                                                }
                                        }
                                        // The pixel is in the middle and has neighbors on its right and left
                                        else{
                                                sum += Z[i][j]*Z[i][j-1];
                                                sum += Z[i][j]*Z[i-1][j-1];
                                                sum += Z[i][j]*Z[i-1][j];
                                                sum += Z[i][j]*Z[i-1][j+1];
                                                sum += Z[i][j]*Z[i][j+1];
                                        }
                                }
                                // The block is not at the bottom therefore pixels at the bottom border can be used
                                else{
                                        // The pixel is on the leftmost column
                                        if(j == 0){
                                                sum += Z[i][j]*Z[i-1][j];
                                                sum += Z[i][j]*Z[i-1][j+1];
                                                sum += Z[i][j]*Z[i][j+1];
                                                sum += Z[i][j]*bottom[j];
                                                sum += Z[i][j]*bottom[j+1];
                                                // If the block is not on the leftmost part of picture pixels from
left border can be used
                                                if(world_rank %n != 1){
                                                        sum += Z[i][j]*left[i-1];
                                                        sum += Z[i][j]*left[i];
                                                        sum += Z[i][j]*bottomLeft;
                                                }
                                        }
                                        // The pixel is on the rightmost column
                                        else if(j == N/n-1){
                                                sum += Z[i][j]*Z[i][j-1];
                                                sum += Z[i][j]*Z[i-1][j-1];
                                                sum += Z[i][j]*Z[i-1][j];
                                                sum += Z[i][j]*bottom[j-1];
                                                sum += Z[i][j]*bottom[j];
                                                // If the block is not on the rightmost part of picture pixels
from right border can be used
                                                if(world_rank %n != 0){
                                                        sum += Z[i][j]*right[i-1];
```

```
                                                sum += Z[i][j]*right[i];
                                                sum += Z[i][j]*bottomRight;
                                        }
                                }
                                // The pixel is in the middle and has neighbors on its right and left
                                else{
                                        sum += Z[i][j]*Z[i][j-1];
                                        sum += Z[i][j]*Z[i-1][j-1];
                                        sum += Z[i][j]*Z[i-1][j];
                                        sum += Z[i][j]*Z[i-1][j+1];
                                        sum += Z[i][j]*Z[i][j+1];
                                        sum += Z[i][j]*bottom[j-1];
                                        sum += Z[i][j]*bottom[j];
                                        sum += Z[i][j]*bottom[j+1];
                                }
                        }
                }
                // The pixel is not on the border of an adjacent block from top or bottom
                else{
                        // The pixel is on the leftmost column
                        if(j == 0){
                                sum += Z[i][j]*Z[i-1][j];
                                sum += Z[i][j]*Z[i-1][j+1];
                                sum += Z[i][j]*Z[i][j+1];
                                sum += Z[i][j]*Z[i+1][j];
                                sum += Z[i][j]*Z[i+1][j+1];
                                // If the block is not on the leftmost part of picture pixels from left border
can be used
                                if(world_rank %n != 1){
                                        sum += Z[i][j]*left[i-1];
                                        sum += Z[i][j]*left[i];
                                        sum += Z[i][j]*left[i+1];
                                }
                        }
                        // The pixel is on the rightmost column
                        else if(j == N/n-1){
                                sum += Z[i][j]*Z[i][j-1];
                                sum += Z[i][j]*Z[i-1][j-1];
                                sum += Z[i][j]*Z[i-1][j];
                                sum += Z[i][j]*Z[i+1][j-1];
                                sum += Z[i][j]*Z[i+1][j];
                                // If the block is not on the rightmost part of picture pixels from right
border can be used
                                if(world_rank %n != 0){
                                        sum += Z[i][j]*right[i-1];
                                        sum += Z[i][j]*right[i];
                                        sum += Z[i][j]*right[i+1];
                                }
                        }
                        // The pixel is in the middle and has neighbors on its right and left
                        else{
                                sum += Z[i][j]*Z[i][j-1];
                                sum += Z[i][j]*Z[i-1][j-1];
                                sum += Z[i][j]*Z[i-1][j];
                                sum += Z[i][j]*Z[i-1][j+1];
                                sum += Z[i][j]*Z[i][j+1];
                                sum += Z[i][j]*Z[i+1][j-1];
                                sum += Z[i][j]*Z[i+1][j];
                                sum += Z[i][j]*Z[i+1][j+1];
                        }

                }
                alpha += -2 * beta * sum;
                if(alpha > log((double)rand() / (double)RAND_MAX))
                        Z[i][j] *= -1;
                // Sending possible changes to the adjacent processors if exists
                if(world_size >2){
                        send2(Z,N,n,world_rank,world_size,i,j);

        receive2(top,bottom,left,right,&topLeft,&topRight,&bottomLeft,&bottomRight,N,n,world_rank,world_size);
                }

        }
        // Sending the back to the master processor
        // Deallocating memory
        for(int i=0;i<N/n;i++){
                MPI_Send(Z[i], N/n, MPI_INT, 0, 0, MPI_COMM_WORLD);
                free(X[i]);
                free(Z[i]);
```

```
                }
                free(X);
                free(Z);
                free(top);
                free(bottom);
                free(right);
                free(left);
        }

        MPI_Finalize();
}
```