CMPE 300
ANALYSIS OF ALGORITMS

HASAN ÖZTÜRK
2017400258

CMPE 300 – MPI PROJECT

PROGRAMMING PROJECT

SUBMISSION DATE: 25.12.2018 – 11.15 PM

INTRODUCTION

This project aims to denoise an image in a parallel manner. A noisy image is the image whose some pixels in the original one are changed . In this project all images are white and black, therefore changing a pixel means flipping the pixel to white or black. Denoising an image means that trying to reach from the noisy image to the original one. We can divide project into 2 main subsections:

- Denoise the image
- Do it in a parallel manner

First part of the project is achieved using a Monte Carlo algorithm. This algorithm first randomly selects a pixel in the image and looks the neighbor pixels of the selected one. Algorithm flips or not the pixel according to the ratio of the surrounding pixels. Technical equations will be elaborated later.

To do this task in a parallel manner, MPI(Message Passing Interface) library is used. This library creates the illusion of processing the code on N processors parallelly. MPI also provides functions for the communication between processors. Detailed explanation will be done later.

PROGRAM INTERFACE

Program takes an text file as an input file. This file contains -1’ s and 1’s which correspond black and white pixels respectively. The program is compiled with the following command:

mpic++ -g main.cpp -o out

After the compilation of the program, the created executable could be run with the following command

mpiexec -n NUM_PROCESSORS ./out input.txt output.txt 0.8 0.15

Here we assumed that the images will 200 * 200 pixels. The number of processors given must a divider of 200 plus 1.

PROGRAM EXECUTION

The program will take a text file as input just as described above. This text file is the text representation of a black and white image. During the execution of the program, a random pixel will be selected and according the Monte Carlo algorithm this pixel will be flipped or not. The number of iterations in this program is up to the programmer's choice. It is optimized based on my experience.

Besides the user should provide alpha ve pi values as arguments. These arguments will be used during the calculation of the probability of flipping a pixel.

After the iterations are over, the program outputs the resulting image again with a text represention. It will be created in the program's directory.

PROGRAM STRUCTURE

Program is structured in a parallel manner. The input image is divided horizontally to equal pieces and each processors will process its own piece. Of course the problem arises between the boundaries of the processors. Because processors should communicate with each other to correctly denoise their image. Remember, every pixel is flipped or not according its surrounding pixels.

In order to provide this communication I used the following MPI functions:

```
MPI_Send(
    void* data,
    int count,
    MPI_Datatype datatype,
    int destination,
    int tag,
    MPI_Comm communicator)
```

```
MPI_Recv(
    void* data,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm communicator,
    MPI_Status* status)
```

In the MPI interface, processors are differentiated according their world ranks. This means that every processor has a distinct rank and programmer arrange the processors using their ranks.

In sending and receiving messages, the logic is similar. If a processor is sending a message, it should provide the rank of the receiver processor and vice versa. The data to be sent and received must be specified and also the number of data to be sent must be given to the functions.

These two functions work in a blocking manner. This means that once a processors calls send function, it cannot continue until its message is received by some other processor. Similarly once a processor calls receive function it cannot continue its execution until it receives the message. There are also other MPI functions but in this project I just used these two functions.

The communication between processors is necessary for the boundary rows. Every processor should send its boundary rows to the neighbor rows and also receive the boundary rows from neighbor processor. The difficulty is the avoidance of deadlock. Because these functions are blocking and they can easily enter deadlock.

In the project I implemented a master slave relationship between processors. There is one master processor whose rank is 0. Master processor takes the input divides it into equal pieces, sends them to slaves, takes the processed output from the slaves, merges them and prints the output to the file. On the other hand, slave processors take their part of the input and process the image by communicating between neighbors.

In this project I divided the processors into 4 categories according to their world ranks.

- Processor whose rank is 1
- Processor whose rank in N
- Processor whose rank is odd
- Processor whose rank is even

Processors whose rank are 1 and N are boundary processors and they should be handled differently from the internal processors. Every processor whose rank is odd communicates with processors whose rank is even. They send bottom and top rows and similarly receive bottom and top rows. Once this communication is over, row number of each processor is increased by 2. They process their images with this extended input. In this way boundary rows are denoised better.

The communication structure of processors whose rank is odd:

```
//Send the topline
 MPI_Send(&X[0][0],200,MPI_INT,world_rank- 1 , 1 , MPI_COMM_WORLD);

 //Receive the topline
 int *topLineX = new int[200];
MPI_Recv(topLineX,200, MPI_INT, world_rank-1, 2, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

//Send the bottomline
MPI_Send(&X[200/N-1][0],200,MPI_INT,world_rank+1,3,MPI_COMM_WORLD);

//Receive the bottomline
int *bottomLineX = new int[200];
MPI_Recv(bottomLineX,200, MPI_INT, world_rank+1, 4, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
```

The communication structure of processors whose rank is even:

```
//Receive the bottomline
int *bottomLineZ = new int[200];
MPI_Recv(bottomLineZ,200, MPI_INT, world_rank+1, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

 //Send the bottomline
 MPI_Send(&Z[(200/N)-1]
[0],200,MPI_INT,world_rank+1,2,MPI_COMM_WORLD);

 //Receive the topline
 int *topLineZ = new int[200];
 MPI_Recv(topLineZ,200, MPI_INT, world_rank-1, 3, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

 //Send the topline
 MPI_Send(&Z[0][0],200,MPI_INT,world_rank-1,4,MPI_COMM_WORLD);
```

The important thing here is that, processors whose rank is odd first call send function whereas processors whose rank is even first call receive function. In this way they do not enter deadlock.

The communication structure of processors whose rank is 1:

```
MPI_Send(&X[(200/N)-1]  [0],  200 , MPI_INT , world_rank +1, 3 ,
MPI_COMM_WORLD ) ;

int *bottomLineX = new int[200];
MPI_Recv(bottomLineX,200, MPI_INT, world_rank+1, 4, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
```

The communication structure of processors whose rank is N:

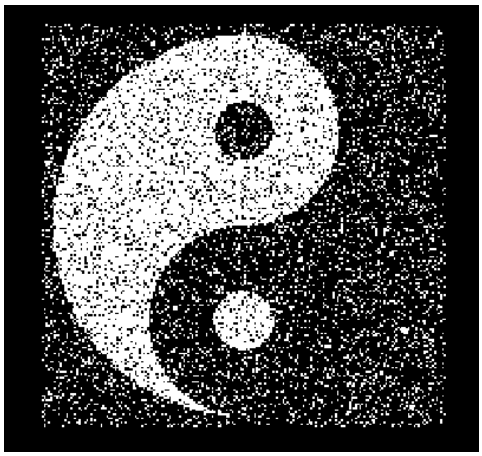
```
int *topLineX = new int[200];
MPI_Recv(topLineX,200, MPI_INT, world_rank-1, 3, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

MPI_Send(&X[0][0],200,MPI_INT,world_rank-1,4,MPI_COMM_WORLD);
```

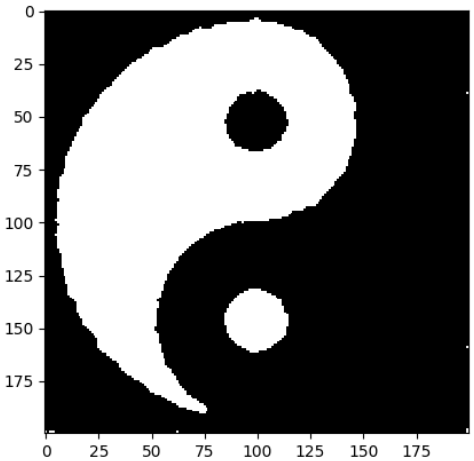
These are the processors which are processing the edge parts of the image. They just send and receive 1 row. Because they only have one neighbor.

EXAMPLES

As mentioned earlier, input file is a text representation of a denoised image. I used the given 200 * 200 images in the project description which are yinyang and lena images. After a few trials I noticed that 1000000 iterations are near optimal for 200 * 200 yinyang image. I used the beta as 0.8 and pi as 0.15. The output image is almost same with the output of the python script which is provided with the project.



Input



Output

IMPROVEMENTS AND EXTENSIONS

In my implementation every slave processor shares boundary rows in every iteration. This brings a performance overhead to the program. However if we do not share boundary rows in each iteration, we cannot be sure if the neighbor processors reach the most recent version of the image. Therefore this seems a good way to denoise the image. In fact there is a trade-off between performance and the denoising process. If we can sacrifise from denosing, we could get more fast run time and vice versa.

DIFFICULTIES ENCOUNTERED

The most difficult part of the project for me is the avoidance of the deadlock in communication between processors. I overcomed this difficulty by dividing the processors into different categories and making send end receive calls according to that.

CONCLUSION

This project was helpful and educational for me. I developed a parallel program for the first time and it really improved my programming skills. Besides the Monte Carlo algorithm was a beautiful one, I really liked the algorithm.

APPENDICES

```
#include <iostream>
#include <fstream>
#include <iterator>
#include <sstream>
#include <vector>
#include <cmath>
#include <cstdlib>
#include <ctime>
#include <mpi.h>


using namespace std;

//This is the split function for parsing inputs
template <class Container>
void splitStr(const string& str, Container& cont)
{
   istringstream iss(str);
```

```
    copy(istream_iterator<string>(iss), istream_iterator<string>(),
back_inserter(cont));
}

int cutAndSum(int array[200][200],int rowStart, int rowEnd, int colStart, int colEnd);
int **alloc_2d_int(int rows, int cols);

int main() {


    //ifstream infile("lena200_noisy.txt");
    //ifstream infile("matrix.txt");
    ifstream infile("yinyang_noisy.txt");

    srand(time(nullptr));
    vector<string> words;
    string line;
    int X[200][200];

    FILE *myfile;
    myfile = fopen("output.txt","w");

    //MPI Initialization
    MPI_Init(NULL,NULL);
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    int N = world_size - 1;

    double beta = 0.8;
    double pi = 0.15;
    double gamma = 0.5*log((1-pi)/pi);
    int T = 1000000;


    /*
     * In this if statement master slave takes the input and sends the input by dividing it
into
     * number of slave processors. Then takes the denoised parts from the slaves and
merges them.
     * Then prints the denosied image to the output file
     */
    if(world_rank==0){
```

```
        //Read Input to 2d Array X
        for(int i=0;i<200;i++){
            getline(infile,line);
            splitStr(line,words);
            for(int j=0;j<200;j++){
                X[i][j] = stoi(words[j]);
            }
            words.clear();
        }

        /*
         * In order to send the input matrix to the slaves, I dynamically allocated an
array A and
         * copied the content of input matrix to A
         */
        int** A;
        A = alloc_2d_int(200,200);
        for(int i=0;i<200;i++){
            for(int j=0;j<200;j++){
                A[i][j] = X[i][j];
            }
        }

        //Send the input matrix to the slaves by dividing it into the number of processors
        for(int i=1;i <= N;i++){

MPI_Send(&A[(200/N)*(i-1)][0],200*(200/N),MPI_INT,i,0,MPI_COMM_WORLD)
;
        }
        free(A);

        //// RECEIVE DENOISED IMAGES FROM SLAVES
        for(int t=1;t <= N;t++){
            int *subarray = new int[200*(200/N)];
            MPI_Recv(subarray,200*(200/N), MPI_INT, t, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

            //convert 1D array to 2D
            int D[200/N][200];
            for(int i=0;i<(200/N);i++){
                for(int j=0;j<200;j++){
                    D[i][j] = subarray[200*i + j];
                }
            }
```

```
            //Print the partial matrix to output file
            for(int i=0;i<200/N;i++){
                for(int j=0;j<200;j++){
                    fprintf(myfile,"%d ",D[i][j]);
                }
                fprintf(myfile,"\n");
            }
            delete [] subarray;
        }
    }
    /*
     * Slave processes execute this else block. They first take their part of the input matrix and make
     * operations on it and send it back to master
     */
    else{

        /*
         * Slaves take (200/N) * 200 matrix from the master an 1d array. Once they receive partial input,
         * They convert 1d array to 2d array
         */
        int *subarr = new int[200*(200/N)];

        MPI_Recv(subarr,200*(200/N), MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        /*
        printf("Process %d received elements: ", world_rank);
        for(int i = 0 ; i < 200*(200/N)  ; i++)
            printf("%d ", subarr[i]);
        printf("\n");*/


        //convert 1D array to 2D
        int X[200/N][200];
        for(int i=0;i<(200/N);i++){
            for(int j=0;j<200;j++){
                X[i][j] = subarr[200*i + j];
            }
        }


        //Z = X.copy()
        int Z[200/N][200];
```

```
for(int i=0;i<200/N;i++){
   for(int j=0;j<200;j++){
      Z[i][j] = X[i][j];
   }
}


/*
 * Processors whose rank are 1 and N work at the boundaries of the image and we should handle
 * these processors seperately, because they just share 1 row with neighbors as opposed to internal processors
 * which share 2 rows.
 */

if(world_rank == 1 || world_rank == N){

   if(world_rank ==1){

      /*
       * Processor whose rank is 1 converts its matrix from (200/N) * 200 to (200/N) +1 * 200 by
       * receiving 1 row from the below processor. Plus, it sends the boundary row to its neighbor.
       * Before starting the iterations we first update our X matrix to extX which stands for
       * extended X
       */

      MPI_Send(&X[(200/N)-1] [0],200,MPI_INT,world_rank+1,3,MPI_COMM_WORLD);

      int *bottomLineX = new int[200];
      MPI_Recv(bottomLineX,200, MPI_INT, world_rank+1, 4, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

      //Create the extended X array by taking 1 row from the neighbor
      //Extended X is (200/N + 1)*200
      int extX[(200/N)+1][200];
      //Take the bottomline
      for(int i=0;i<200;i++){
         extX[200/N][i] = bottomLineX[i];
      }
      //Take the rest
      for(int i=0;i<(200/N);i++){
```

```
            for(int j=0;j<200;j++){
                extX[i][j] = X[i][j];
            }
        }

        /*
         * This is the main for loop for processor whose rank is 1. Iterations are achieved here and in each
         * iteration neighbor pixels are shared.
         */
        for(int t=0;t<T/N;t++){


            MPI_Send(&Z[(200/N)-1][0],200,MPI_INT,world_rank+1,3,MPI_COMM_WORLD);

            int *bottomLineZ = new int[200];
            MPI_Recv(bottomLineZ,200, MPI_INT, world_rank+1, 4, MPI_COMM_WORLD, MPI_STATUS_IGNORE);


            /*
             * Calculations are with the extended Z matrix whose dimensions are (200/N) +1 * 200. In this part,
             * we create extended Z.
             */
            int extZ[(200/N)+1][200];
            //Take the topline
            for(int i=0;i<200;i++){
                extZ[200/N][i] = bottomLineZ[i];
            }
            //Take the rest
            for(int i=0;i<(200/N);i++){
                for(int j=0;j<200;j++){
                    extZ[i][j] = Z[i][j];
                }
            }

            //Pick a pixel randomly
            int i,j;
            i = rand() % 200/N+1;
            j = rand() % 200;

            //Main equation
```

```
                double delta_E = -2*gamma*extX[i][j]*extZ[i][j] - 2*beta*extZ[i]
[j]*(cutAndSum(extZ,max(i-1,0),i+2,max(j-1,0),j+2)-extZ[i][j]);

                //Flip the pixel with the probability
                double r = ((double) rand() / (RAND_MAX));
                if(log(r) < delta_E)
                   extZ[i][j] = -extZ[i][j];

                //Convert the extended Z to original Z by deleting extra row
                for(int i=0;i<(200/N);i++){
                   for(int j=0;j<200;j++){
                      Z[i][j] = extZ[i][j];
                   }
                }

             }

          }
          //if world rank == N
          else{

             /*
              * Processor whose rank is N converts its matrix from (200/N) * 200 to
(200/N) +1 * 200 by
              * receiving 1 row from the above processor. Plus, it sends the boundary
row to its neighbor.
              * Before starting the iterations we first update our X matrix to extX which
stands for
              * extended X
              */

             int *topLineX = new int[200];
             MPI_Recv(topLineX,200, MPI_INT, world_rank-1, 3,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

             MPI_Send(&X[0][0],200,MPI_INT,world_rank-
1,4,MPI_COMM_WORLD);

             //Create the extended X array by taking 1 row from the neighbor
             //Extended X is (200/N + 1)*200
             int extX[(200/N)+1][200];
             //Take the topline
             for(int i=0;i<200;i++){
                extX[0][i] = topLineX[i];
             }
```

```
//Take the rest
for(int i=0;i<(200/N);i++){
    for(int j=0;j<200;j++){
        extX[i+1][j] = X[i][j];
    }
}

/*
 * This is the main for loop for processor whose rank is N. Iterations are achieved here and in each
 * iteration neighbor pixels are shared.
 */
for(int t=0;t<T/N;t++){

    int *topLineZ = new int[200];
    MPI_Recv(topLineZ,200, MPI_INT, world_rank-1, 3, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    MPI_Send(&Z[0][0],200,MPI_INT,world_rank-1,4,MPI_COMM_WORLD);


    /*
     * Calculations are with the extended Z matrix whose dimensions are (200/N) +1 * 200. In this part,
     * we create extended Z.
     */
    int extZ[(200/N)+1][200];
    //Take the topline
    for(int i=0;i<200;i++){
        extZ[0][i] = topLineZ[i];
    }
    //Take the rest
    for(int i=0;i<(200/N);i++){
        for(int j=0;j<200;j++){
            extZ[i+1][j] = Z[i][j];
        }
    }

    //Pick a pixel randomly
    int i,j;
    i = rand() % 200/N+1;
    j = rand() % 200;

    //Main equation
```

```
                double delta_E = -2*gamma*extX[i][j]*extZ[i][j] - 2*beta*extZ[i]
[j]*(cutAndSum(extZ,max(i-1,0),i+2,max(j-1,0),j+2)-extZ[i][j]);

                //Flip the pixel with the probability
                double r = ((double) rand() / (RAND_MAX));
                if(log(r) < delta_E)
                   extZ[i][j] = -extZ[i][j];

                //Convert the extended Z to original Z by deleting extra row
                for(int i=0;i<(200/N);i++){
                   for(int j=0;j<200;j++){
                      Z[i][j] = extZ[i+1][j];
                   }
                }

             }

          }

       }
       /* To avoid deadlock between processor, I divide the processors into to with the
following manner:
        * Processors whose rank is odd does send - recv - send - recv and processors
whose rank is even
        * does recv - send - recv - send to communicate with each other. All odd ranked
processors communicate
        * with even ranked processors and vice versa. Different from the boundary
processors, these internal
        * processors share two rows with two neighbors
       */
       else{
          //Even ranked processors
          if(world_rank % 2 ==0){

             //Receive the bottom row
             int *bottomLineX = new int[200];
             MPI_Recv(bottomLineX,200, MPI_INT, world_rank+1, 1,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

             //Send the bottom row
             MPI_Send(&X[(200/N)-1]
[0],200,MPI_INT,world_rank+1,2,MPI_COMM_WORLD);

             //Receive the top row
             int *topLineX = new int[200];
```

```
        MPI_Recv(topLineX,200, MPI_INT, world_rank-1, 3,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        //Send the top row
        MPI_Send(&X[0][0],200,MPI_INT,world_rank-
1,4,MPI_COMM_WORLD);

        //Create the extended X array by taking 2 rows from the neighbor
        //Extended X is (200/N + 2)*200
        int extX[(200/N)+2][200];
        //Take the topline
        for(int i=0;i<200;i++){
           extX[0][i] = topLineX[i];
        }
        //Take the bottomline
        for(int i=0;i<200;i++){
           extX[(200/N)+1][i] = bottomLineX[i];
        }
        //Copy the rest
        for(int i=0;i<(200/N);i++){
           for(int j=0;j<200;j++){
              extX[i+1][j] = X[i][j];
           }
        }

        /*
         * This is the main for loop for the internal processors whose rank are even.
         * Iterations are achieved here and in each iteration neighbor pixels are
shared.
         */
        for(int t=0;t<T/N;t++){

           //Receive the bottomline
           int *bottomLineZ = new int[200];
           MPI_Recv(bottomLineZ,200, MPI_INT, world_rank+1, 1,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

           //Send the bottomline
           MPI_Send(&Z[(200/N)-1]
[0],200,MPI_INT,world_rank+1,2,MPI_COMM_WORLD);

           //Receive the topline
           int *topLineZ = new int[200];
           MPI_Recv(topLineZ,200, MPI_INT, world_rank-1, 3,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
            //Send the topline
            MPI_Send(&Z[0][0],200,MPI_INT,world_rank-
1,4,MPI_COMM_WORLD);


            /*
             * Calculations are with the extended Z matrix whose dimensions are
(200/N) +2 * 200. In this part,
             * we create extended Z.
             */
            int extZ[(200/N)+2][200];
            //Take the topline
            for(int i=0;i<200;i++){
               extZ[0][i] = topLineZ[i];
            }
            //take the bottomline
            for(int i=0;i<200;i++){
               extZ[200/N+1][i] = bottomLineZ[i];
            }
            //Take the rest
            for(int i=0;i<(200/N);i++){
               for(int j=0;j<200;j++){
                  extZ[i+1][j] = Z[i][j];
               }
            }

            //Pick a random pixel
            int i,j;
            i = rand() % 200/N+1;
            j = rand() % 200;

            //Main equation
            double delta_E = -2*gamma*extX[i][j]*extZ[i][j] - 2*beta*extZ[i]
[j]*(cutAndSum(extZ,max(i-1,0),i+2,max(j-1,0),j+2)-extZ[i][j]);

            //Flip the pixel with a probability
            double r = ((double) rand() / (RAND_MAX));
            if(log(r) < delta_E)
               extZ[i][j] = -extZ[i][j];

            //Convert the extended Z to original Z by deleting extra row
            for(int i=0;i<(200/N);i++) {
               for (int j = 0; j < 200; j++) {
                  Z[i][j] = extZ[i + 1][j];
```

```
                }
            }

        }

    }
    //Processors whose rank is odd
    else{

        //Send the topline
        MPI_Send(&X[0][0],200,MPI_INT,world_rank-
1,1,MPI_COMM_WORLD);

        //Receive the topline
        int *topLineX = new int[200];
        MPI_Recv(topLineX,200, MPI_INT, world_rank-1, 2,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        //Send the bottomline
        MPI_Send(&X[200/N-1]
[0],200,MPI_INT,world_rank+1,3,MPI_COMM_WORLD);

        //Receive the bottomline
        int *bottomLineX = new int[200];
        MPI_Recv(bottomLineX,200, MPI_INT, world_rank+1, 4,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        //Create the extended X array by taking 2 rows from the neighbor
        //Extended X is (200/N + 2)*200
        int extX[(200/N)+2][200];
        for(int i=0;i<200;i++){
            extX[0][i] = topLineX[i];
        }
        for(int i=0;i<200;i++){
            extX[(200/N)+1][i] = bottomLineX[i];
        }
        for(int i=0;i<(200/N);i++){
            for(int j=0;j<200;j++){
                extX[i+1][j] = X[i][j];
            }
        }

        /*
         * This is the main for loop for the internal processors whose rank are even.
```

```
            * Iterations are achieved here and in each iteration neighbor pixels are
shared.
            */
           for(int t=0;t<T/N;t++){

              //Send the topline
              MPI_Send(&Z[0][0],200,MPI_INT,world_rank-
1,1,MPI_COMM_WORLD);

              //Receive the topline
              int *topLineZ = new int[200];
              MPI_Recv(topLineZ,200, MPI_INT, world_rank-1, 2,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

              //Send the bottomline
              MPI_Send(&Z[(200/N)-1]
[0],200,MPI_INT,world_rank+1,3,MPI_COMM_WORLD);

              //Receive the bottomline
              int *bottomLineZ = new int[200];
              MPI_Recv(bottomLineZ,200, MPI_INT, world_rank+1, 4,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);


              /*
               * Calculations are with the extended Z matrix whose dimensions are
(200/N) +2 * 200. In this part,
               * we create extended Z.
               */
              int extZ[(200/N)+2][200];
              //Take the topline
              for(int i=0;i<200;i++){
                 extZ[0][i] = topLineZ[i];
              }
              //take the bottomline
              for(int i=0;i<200;i++){
                 extZ[200/N+1][i] = bottomLineZ[i];
              }
              //Take the rest
              for(int i=0;i<(200/N);i++){
                 for(int j=0;j<200;j++){
                    extZ[i+1][j] = Z[i][j];
                 }
              }
```

```
            //Pick a random pixel
            int i,j;
            i = rand() % 200/N+1;
            j = rand() % 200;

            //Main equation
            double delta_E = -2*gamma*extX[i][j]*extZ[i][j] - 2*beta*extZ[i]
[j]*(cutAndSum(extZ,max(i-1,0),i+2,max(j-1,0),j+2)-extZ[i][j]);

            //Flip the pixel with the probability
            double r = ((double) rand() / (RAND_MAX));
            if(log(r) < delta_E)
               extZ[i][j] = -extZ[i][j];

            //Convert the extended Z to original Z by deleting extra row
            for(int i=0;i<(200/N);i++) {
               for (int j = 0; j < 200; j++) {
                  Z[i][j] = extZ[i + 1][j];
               }
            }

         }
      }

   }

   //// SEND DENOISED IMAGES TO MASTER

   //In order to send the partial images I created dynamic array
   int** D;
   D = alloc_2d_int(200/N,200);

   //Fill the dynamic array
   for(int i=0;i<200/N;i++){
      for(int j=0;j<200;j++){
         D[i][j] = Z[i][j];
      }
   }

   //send to master
   MPI_Send(&D[0][0],200*(200/N),MPI_INT,0,0,MPI_COMM_WORLD);

   free(D);

   delete [] subarr;
```

```
    }

    MPI_Finalize();
    fclose(myfile);

    return 0;
}

//Dynamically creates 2d array
int **alloc_2d_int(int rows, int cols) {
    int *data = (int *)malloc(rows*cols*sizeof(int));
    int **array= (int **)malloc(rows*sizeof(int*));
    for (int i=0; i<rows; i++)
        array[i] = &(data[cols*i]);

    return array;
}

//Converts a matrix into a small matrix according to arguments and returns the sum
int cutAndSum(int array[200][200],int rowStart, int rowEnd, int colStart, int colEnd)
{

    int sum = 0;

    for(int i=rowStart;i < rowEnd;i++){
        for(int j=colStart;j<colEnd;j++){
            sum += array[i][j];
        }
    }
    return sum;
}
```