CMPE300

ANALYSIS OF ALGORITHMS

ELİF ÇALIŞKAN

TUNGA GÜNGÖR

IMAGE DENOISING PROGRAMMING PROJECT

DUE: 26.12.2018 - 23:59

1)Introduction:

In this project, we were asked to denoise a noisy picture using parallel programming. We are given a noisy picture that was created by image_to_text.py and make_noise.py. Then with the help of Ising Model, we flip the pixel based on the acceptance probability. After many iterations, we end up with a denoised picture. To achieve that result, in every iteration the program picks a random pixel. This pixel's denoised value depends on both its neighbors and noise probability. The program calculates delta_E. If a random probability is less than exponential of delta_E, it flips the pixel.

Before doing this in parallel programming, first I implemented this algorithm for only one process. Then I split the code into one master and world_size-1 processes. First, I implemented the code without communication between processors. The picture had odd looking lines without communication. After that, I added send and receive functions and achieved the denoised picture.

2)Program Interface:

I used C++ with Open MPI under Linux for this project and to compile the project:

mpic++ -g main.cpp -o program

To execute the project:

mpiexec -n number_of_processors ./program input_file output_file beta pi

For seeing the denoised image:

python(3) text_to_image.py output_file image.png

3)Program Execution:

number_of_processors: This is the number of processors that is needed to make use of parallel programming. 1 processor is the master. This number is used as world_size in the mpi code.

input_file: Input file is the array version of noisy image and it is a text file. To get this file image_to_text.py and make_noise.py should be executed for a png image. The pixels have value 1 if it is white -1 if it is black. Txt file is the array of corresponding values.

output_file: Output file is the array version of denoised image and it is a text file. This is the output of my program. To see the image version, text_to_image.py should be executed. The pixels have value 1 if it is white -1 if it is black. Txt file is the array of corresponding values.

beta: This is the beta value. We expect more consistency between the neighboring pixels if β is higher.

pi: This is the pi value. π implies higher noise on the X.

text_to_image.py: This is a script to see the image of corresponding array.

4)Input and Output:

Image representation of yinyang_noisy.txt

yinyang denoised with 5 processors

beta= 0.6 pi = 0.1

Image representation of lena200_noisy.txt

lena200 denoised with 21 processors

beta=0.8 pi=0.15

5)Program Structure:

int max(int a,int b): This function returns the maximum number of a and b. This is used in finding sum of a pixel's neighbors.

int min(int a,int b): This function returns the minimum number of a and b. This is used in finding sum of a pixel's neighbors.

int main(int argc, char argv):** This is the part which does most of the work. It takes 4 arguments. First one is inputFile. This is a string variable and it takes the path of array version of noisy image. Second one is outputFile. This is a string variable and it takes the path of wanted denoised array version. Third argument is beta and fourth one is pi. They have double types and I used stod for casting.

It initializes MPI and gets world_size and world_rank informations for every processor. world_size is the number of processors that were given at execution. world_rank is the rank of processor and it can have values from 0 to world_size-1.

I compute the value of gamma to use in delta_E. I created an integer T variable for the number of total iterations and an integer N variable for one row's pixel count. According to

the given information, I initialized T to 500000 and N to 200. Also, I created an integer called slaveNum that has world_size-1 value for convenience.

First, I checked if the world_rank is equal to 0 for finding the master processor. If it is the master processor, I read the file with fscan into z[N][N] array. Then I send corresponding arrays to every slave processor in a for loop. I used MPI_send for achieving this. Since it is blocking, master waits for every processor to receive the data. After the slaves finish their job, they send the updated version of their array. In a for loop master waits for every processor to send their data. When every pixel has returned, the z[N][N] gets updated. Then the master writes the output to outputFile in a correct manner.

If the processor is a slave, then the procedure is different. First, I check if the world_rank is 1 since it is treated differently. Then I create subZ and subX arrays of size N*N/slaveNum. I receive the array that was sent by the master into subZ and copy every element to subX. Then the iteration that lasts for T/slaveNum begins. Since this processor uses the upper part of the picture, the only addition will be for the bottom part. So, it receives the first row of the next processor's array and puts it into newLineBottom. Then it sends its last row to the next processor. Since subZ has N*N/slaveNum pixels and it is a one dimensional array, I choose a random number between 0 and N*N/slaveNum. Then I compute the sum of its neighbors. But since the array has one dimension, I made some calculations to find out if it is on edges. Outer for loop starts from the maximum of (row of j -1) and 0 and ends at minimum of (row of j+1) and N*N/slaveNum-1. Inner for loop starts from the maximum of (column of j -1) and 0 and ends at minimum of (column of j+1) and N-1. If the random pixel is on the bottom edge, we should take these neighbors into account. So in an if statement I add those neighbors to sum. This way it calculates the sum of all neighbors including itself. So, I subtract its value from the sum. Then I calculate the delta_E with the given formula.

$$p(Z \mid X, \beta, \pi) \propto \exp\left(\gamma \sum_{ij} Z_{ij} X_{ij} + \beta \sum_{(i,\ j) \sim (k,\ l)} Z_{ij} Z_{kl}\right) \quad (1)$$

If logarithm of a random double between 0 and 1 is smaller than delta_E, it flips the pixel. Then I free the created pointer at the end of every iteration. When the loop ends, the subZ is sent to master processor. Then pointers subZ and subX are deleted.

If the world_rank is world_size-1, it means the processor has the bottom part of the picture. So, only addition will be for the upper part. After receiving the subZ and subX from the master, it starts the iterations. To avoid deadlock, it sends its first row to the previous processor, then it receives the bottom row of the previous processor. Later, the random pixel gets picked and sum is calculated. After the pixels are flipped and the iterations are over, subZ is sent to master.

If the processor's rank is between 1 and world_size-1, there should be two communications. After receiving the subZ and subX from the master, it starts the iterations. To avoid deadlock, it sends its first row to the previous processor, then it receives the bottom row of the previous processor. Then it receives the first row of the next processor and sends its last row to the next processor. Since the ordering is different, the possibility of every processor waits to send or receive doesn't happen. So, deadlock is avoided. After subZ is updated in every iteration, pointers are deleted and, subZ is sent to master processor.

At the end of the program MPI is finalized.

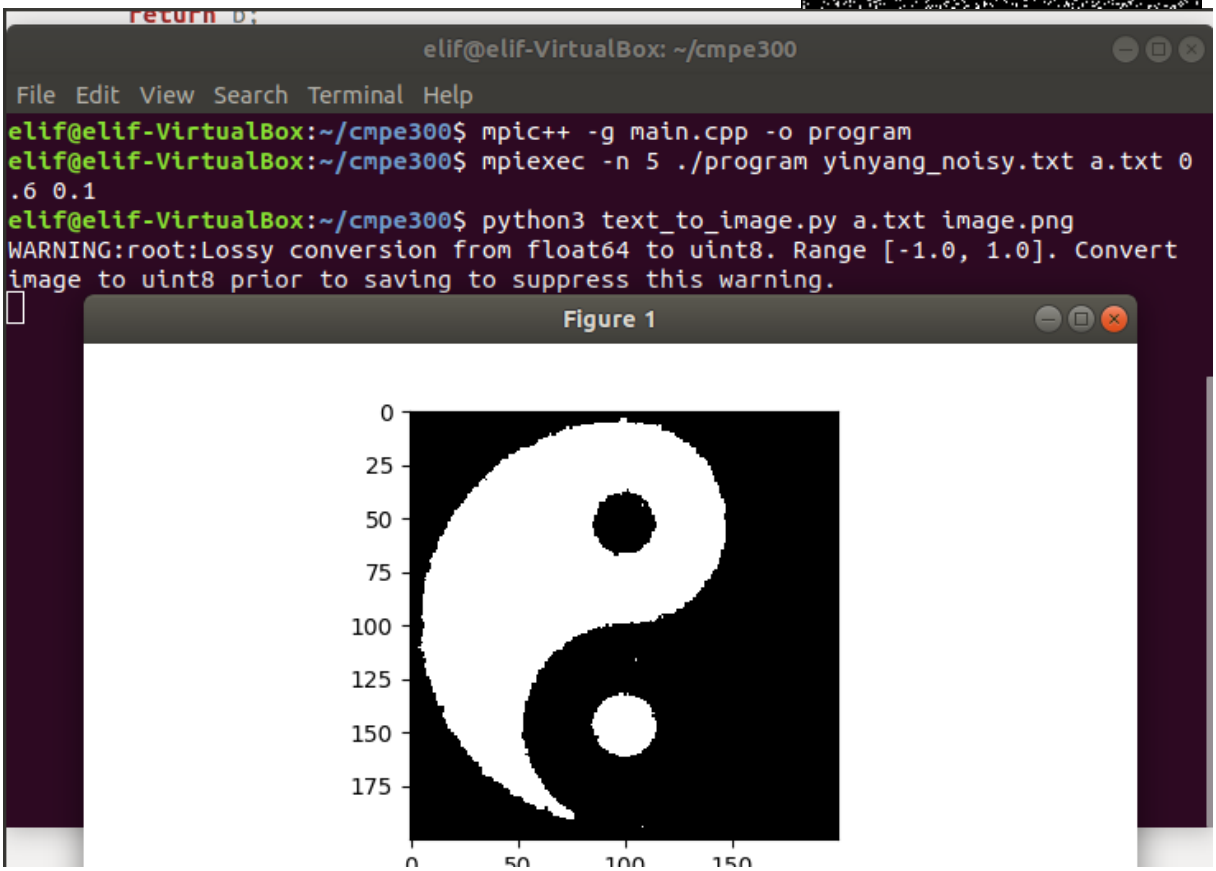
6)Examples: yinyang_noisy

Commands:

mpic++ -g main.cpp -o program

mpiexec -n 5 ./program yinyang_noisy.txt a.txt 0.6 0.1

python3 text_to_image.py a.txt image.png



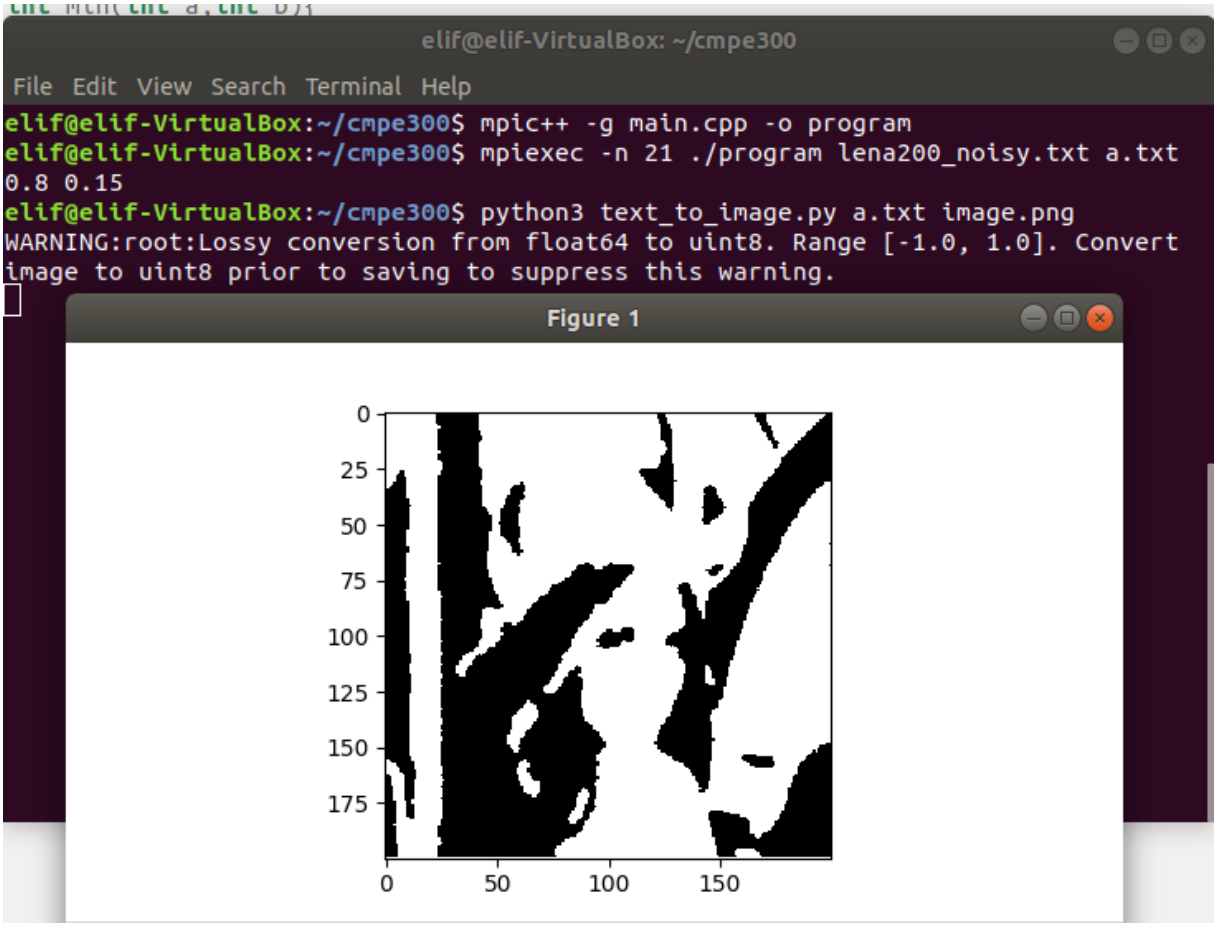
lena200_noisy

Commands:

mpic++ -g main.cpp -o program

mpiexec -n 21 ./program lena200_noisy.txt a.txt 0.8 0.15

python3 text_to_image.py a.txt image.png



7)Improvements and Extensions:

The picture could be divided into vertically to make the program faster and more parallel. But since diagonal neighbors would make the code more complicated to write, I didn't do it. Also, in this code I send and receive in every iteration. But the pixels on the edge won't change after every iteration. So, I use these send and receive operations more than necessary. The code can be improved if arrays are sent to next or previous processes only when they are changed.

8) Difficulties Encountered:

Since this was the first time that I used MPI, I had difficulties installing it. Then I learned how it worked. First, I made the project by only one processor without MPI. Then I divided the work into processors with the help of MPI. At this point I had some doubts about the communication between processors. I was not sure whether I should send subX or subZ. But since the picture should be updated after every iteration, I decided to send subZ.

9) Conclusion:

In this project, we were given the array version of a noisy image. We are asked to denoise this picture. To denoise the picture, we used the Ising model. Every pixel's probable

value depends on its neighbors and there is a constant relation beta between them. Also, it depends on its noisy value and the picture's noise probability pi. So, we compute delta_E to find the acceptance probability. Acceptance probability is the exponential of delta_E. If the probability is less than acceptance probability, (log of random between 0 and 1 is less than delta_E) then we flip the pixel. Our aim is to become closer to the original picture. So, the iteration count must be high in order to be sure that every pixel is visited. This process can be done with only one processor, but this will take more time. So, we divided the work between slave processors and assigned a master for reading the file, dividing the array and writing to output file. Since the arrays in different processes change after every iteration, the communication must be done for the borders of each processors. This is achieved with using MPI_Send and MPI_Receive.

10)Appendices:

```
/*
Student Name: Elif Çalışkan
Student Number: 2016400183
Compile Status: Compiling
Program Status: Working
Notes: -
*/

#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>
#include <mpi.h>
#include <string>
using namespace std;
//it returns the maximum of a and b
int max(int a,int b){
    if(a>b)
        return a;
    else
        return b;
}
//it returns the minimum of a and b
int min(int a,int b){
    if(a<b)
        return a;
    else
        return b;
}
//it takes four arguments arg[1] is input's path, arg[2] is output's path,
arg[3] is beta arg[4] is pi
int main(int argc, char** argv) {
    string inputFile=argv[1];
    string outputFile=argv[2];
    double beta = stod(argv[3]);
    double pi= stod(argv[4]);
    //MPI is initialized
    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);
```

```
    double gamma = 0.5*log((1-pi)/pi);
    //iteration count
    int T=500000;
    int N=200;
    //z matrix is the noisy matrix and it is distributed among processors
    int z[N][N];
    //slaveNum is the number of processors that will work on the picture
    int slaveNum=world_size-1;
    //if it is master processor, the input file gets read and the array is
distributed among the processors
    //then the data is received and written to output file
    if(world_rank==0){
        FILE *myFile;
        //I assumed the inputFile exists
        myFile = fopen(inputFile.c_str(), "r");
        for(int i=0;i<N;i++){
            for(int j=0;j<N;j++){
                int k=0;
                fscanf(myFile, "%2d", &k);
                z[i][j]=k;
            }
        }
        //this for loop sends arrays to corresponding processors
        //since MPI_Send is blocking, it waits the data to be received
        for(int i = 1 ; i <=slaveNum ; i++){
            MPI_Send(z[(N/slaveNum)*(i-1)], (N/slaveNum)*N, MPI_INT, i, 0,
MPI_COMM_WORLD);
        }
        //this for loop receives the updated arrays and changes z[N][N]
accordingly
        for(int i = 1 ; i <= slaveNum ; i++){
            int* subarr = NULL;
            subarr = new int[(N/slaveNum)*N];
            MPI_Recv(subarr, (N/slaveNum)*N, MPI_INT, i, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            for(int k=0;k<(N/slaveNum)*N;k++){
                z[(N/slaveNum)*(i-1)+k/N][k%N]=subarr[k];
            }
        }
        //it writes z array into output file
        freopen(outputFile.c_str(), "w",stdout);
        for(int i=0;i<N;i++){
            for(int j=0;j<N;j++){
                printf("%d ", z[i][j]);
            }
            printf("\n");
        }
        fclose(stdout);
    }
    else if(world_rank==1){
        //subZ is received from master and subX is copied from subZ
        //subX is the original array and it is used for finding sum in
delta_E
        int* subZ = NULL;
        subZ = new int[(N/slaveNum)*N];
        int* subX = NULL;
        subX = new int[(N/slaveNum)*N];
        MPI_Recv(subZ, (N/slaveNum)*N, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        for(int k=0;k<(N/slaveNum)*N;k++) {
```

```
            subX[k] = subZ[k];
        }
        //in every iteration the first row of the next processor's array is
received and the last row is sent
        //since it is the first processor there is no previous slave
processor
        for(int t=0;t<T/slaveNum;t++){
            //newLineBottom is the first row of the next processor's array
            int* newLineBottom = NULL;
            newLineBottom = new int[N];
            MPI_Recv(newLineBottom, N, MPI_INT, 2, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            //it sends its last row to second processor
            MPI_Send(&subZ[(N/slaveNum-1)*N], N, MPI_INT, 2, 0,
MPI_COMM_WORLD);
            //a random pixel is picked
            int i=rand() % ((N/slaveNum)*N-1);
            int sum=0;
            //sum is calculated this also has its own value
            for(int a=max(i/N-1,0);a<=min(i/N+1,N/slaveNum-1);a++){
                for(int b=max(i%N-1,0);b<=min(i%N+1,N-1);b++){
                    sum+=subZ[a*N+b];
                }
            }
            //this is the communication part
            //if the picked pixel is on the last row of processor, it
should add the corresponding values in newLineBottom
            if(i/N==N/slaveNum-1){
                for(int b=max(i%N-1,0);b<=min(i%N+1,N-1);b++){
                    sum+=newLineBottom[b];
                }
            }
            //its value is subtracted this only has its neighbors' values
            sum-=subZ[i];
            //delta_E is computed according to the formula
            double delta_E = -2*gamma*subX[i]*subZ[i] -2*beta*subZ[i]*sum;
            double random= rand() / (double)RAND_MAX ;
            //if the probability is less than acceptance probability, a
flip occurs
            if(log(random)<delta_E){
                subZ[i] = -subZ[i];
            }
            //pointers are freed
            delete newLineBottom;
            newLineBottom=NULL;
        }
        //the updated subZ is sent to master
        MPI_Send(subZ, (N/slaveNum)*N, MPI_INT, 0, 0, MPI_COMM_WORLD);
        delete subZ;
        subZ=NULL;
        delete subX;
        subX=NULL;
    }
    //if this is the last processor there is no next processor
    else if(world_rank==world_size-1){
        int* subZ = NULL;
        subZ = new int[(N/slaveNum)*N];
        int* subX = NULL;
        subX = new int[(N/slaveNum)*N];
        MPI_Recv(subZ, (N/slaveNum)*N, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
```

```
        for(int k=0;k<(N/slaveNum)*N;k++) {
            subX[k] = subZ[k];
        }
        for(int t=0;t<T/slaveNum;t++){
            //it sends its first row to the previous processor
            MPI_Send(&subZ[0], N, MPI_INT, world_rank-1, 0,
MPI_COMM_WORLD);
            int* newLineTop = NULL;
            newLineTop = new int[N];
            //it receives the last row of previous processor's array
            MPI_Recv(newLineTop, N, MPI_INT, world_rank-1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

            int i=rand() % ((N/slaveNum)*N-1);
            int sum=0;
            for(int a=max(i/N-1,0);a<=min(i/N+1,N/slaveNum-1);a++){
                for(int b=max(i%N-1,0);b<=min(i%N+1,N-1);b++){
                    sum+=subZ[a*N+b];
                }
            }
            //if the pixel is at the first row, newLineTop is used for sum
            if(i/N==0){
                for(int b=max(i%N-1,0);b<=min(i%N+1,N-1);b++){
                    sum+=newLineTop[b];
                }
            }
            sum-=subZ[i];
            double delta_E = -2*gamma*subX[i]*subZ[i] -2*beta*subZ[i]*sum;
            double random= rand() / (double)RAND_MAX ;
            if(log(random)<delta_E){
                subZ[i] = -subZ[i];
            }
            delete newLineTop;
            newLineTop=NULL;

        }
        MPI_Send(subZ, (N/slaveNum)*N, MPI_INT, 0, 0, MPI_COMM_WORLD);
        delete subZ;
        subZ=NULL;
        delete subX;
        subX=NULL;
    }
    //if this process between 1 and world_size-1, it needs to have
communication in two ways
    else{
        int* subZ = NULL;
        subZ = new int[(N/slaveNum)*N];
        int* subX = NULL;
        subX = new int[(N/slaveNum)*N];
        MPI_Recv(subZ, (N/slaveNum)*N, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        for(int k=0;k<(N/slaveNum)*N;k++) {
            subX[k] = subZ[k];
        }
        for(int t=0;t<T/slaveNum;t++){
            //first it sends its first row to previous one
            MPI_Send(&subZ[0], N, MPI_INT, world_rank-1, 0,
MPI_COMM_WORLD);
            int* newLineTop = NULL;
            newLineTop = new int[N];
            //then it gets the last row from previous process
```

```
            MPI_Recv(newLineTop, N, MPI_INT, world_rank-1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            int* newLineBottom = NULL;
            newLineBottom = new int[N];
            //it gets the first row of the next process
            MPI_Recv(newLineBottom, N, MPI_INT, world_rank+1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            //it sends the last row to the next process
            MPI_Send(&subZ[(N/slaveNum-1)*N], N, MPI_INT, world_rank+1, 0,
MPI_COMM_WORLD);

            int i=rand() % ((N/slaveNum)*N-1);
            int sum=0;

            for(int a=max(i/N-1,0);a<=min(i/N+1,N/slaveNum-1);a++){
                for(int b=max(i%N-1,0);b<=min(i%N+1,N-1);b++){
                    sum+=subZ[a*N+b];
                }
            }
            //if the pixel is at the last row, newLineBottom is used for
sum
            if(i/N==N/slaveNum-1){
                for(int b=max(i%N-1,0);b<=min(i%N+1,N-1);b++){
                    sum+=newLineBottom[b];
                }
            }
            //if the pixel is at the first row, newLineTop is used for sum
            if(i/N==0){
                for(int b=max(i%N-1,0);b<=min(i%N+1,N-1);b++){
                    sum+=newLineTop[b];
                }
            }
            sum-=subZ[i];
            double delta_E = -2*gamma*subX[i]*subZ[i] -2*beta*subZ[i]*sum;
            double random= rand() / (double)RAND_MAX ;
            if(log(random)<delta_E){
                subZ[i] = -subZ[i];
            }
            delete newLineTop;
            newLineTop=NULL;
            delete newLineBottom;
            newLineBottom=NULL;

        }
        MPI_Send(subZ, (N/slaveNum)*N, MPI_INT, 0, 0, MPI_COMM_WORLD);
        delete subZ;
        subZ=NULL;
        delete subX;
        subX=NULL;
    }
    MPI_Finalize();
    return 0;
}
```