CMPE 300

ANALYSIS OF ALGORITHMS
FALL 2018

Name: Gurkan Demir

Student ID: 2015400177

Project Name: Programming Project (MPI)

Submission Date: 25.12.2018

Due Date: 26.12.2018

Introduction

In this project, we are expected to experience parallel programming with C/C++ using MPI library. We implemented a parallel algorithm for image denoising with the Ising model using MetropolisHastings algorithm.

Ising Model: The model consists of discrete variables that represent magnetic dipole moments of atomic spins that can be in one of two states (+1 or -1). The spins are arranged in a graph, usually a lattice, allowing each spin to interact with its neighbors. If we assume that a black and white image is generated using The Ising model, then it means that if we take random black pixel from the image, it is more likely that this pixel is surrounded by black pixels (same for the white pixels).

MetropolisHastings Algorithm: Algoritm tells us which candidate Z image is more similar to the original image. In order to reach the noise free image Z we need to make some modification to noised image.

Program Interface

User can start this program using command terminal. By typing the command

"mpic++ -std=c++11 -g main.cpp"

program compiles and constitutes an executable named "a.out". After creating an executable, user can start the program by typing the command

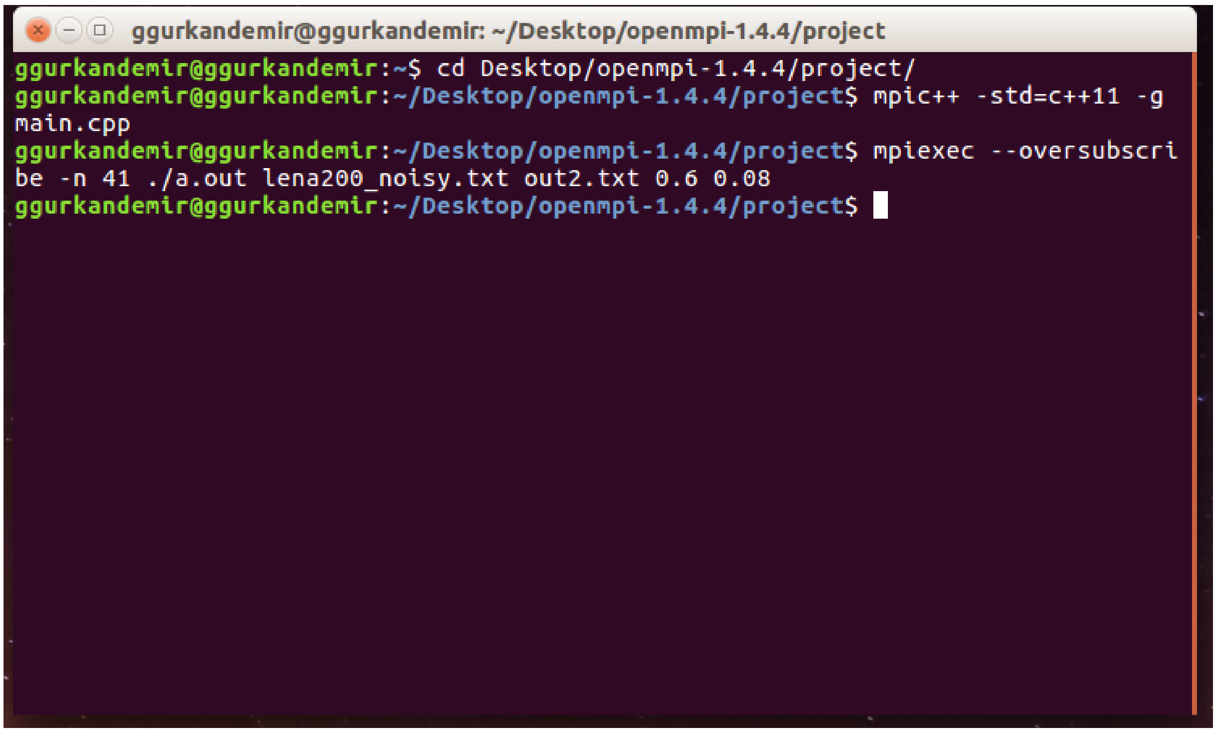
"mpiexec –oversubscribe -n [N] ./a.out [INPUT_FILE] [OUTPUT_FILE] [BETA] [PI]".

After execution, the program terminates itself and prints the expected output to given output file.

Program Execution

As I said before, after creating an executable user can run the program by typing "mpiexec –oversubscribe -n [N] ./a.out [INPUT_FILE] [OUTPUT_FILE] [BETA] [PI]" to the command line. In this context;

- N indicates that how many processors will run the program.
- INPUT_FILE indicates the noised image.
- OUTPUT_FILE indicates the where to print the result of program.
- BETA indicates the value of beta in order to implement MetropolisHastings algorithm.
- PI indicates the value of pi in order to implement MetropolisHastings algoritm.



```
ggurkandemir@ggurkandemir: ~/Desktop/openmpi-1.4.4/project
ggurkandemir@ggurkandemir:~$ cd Desktop/openmpi-1.4.4/project/
ggurkandemir@ggurkandemir:~/Desktop/openmpi-1.4.4/project$ mpic++ -std=c++11 -g
main.cpp
ggurkandemir@ggurkandemir:~/Desktop/openmpi-1.4.4/project$ mpiexec --oversubscri
be -n 41 ./a.out lena200_noisy.txt out2.txt 0.6 0.08
ggurkandemir@ggurkandemir:~/Desktop/openmpi-1.4.4/project$ 
```

Input and Output

As I said in program execution part, process takes 5 arguments other than executable name. [N] which indicates the total number of processors, has no limit but in my program [N]-1 is expected to divisible to 200. Moreover, [INPUT_FILE] which is constituted from +1 or -1, expected to contain 200x200 pixels. Also in this context, [OUTPUT_FILE] which is the result of the program contains 200x200 pixels.

Program Structure

When the programs starts to execute, in main function MPI is initialized. Master, which has rank 0, reads the input file and shares them accross to slaves. Then slaves do some calculations to denoise image, then slaves sends it to master process. Then master merges and prints the result to output file.

1. In main method I initiliazed the MPI and get the size of system and get the rank of each processor.

```
MPI_Init(NULL, NULL);

int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
```

2. After initialization, I declare the pixel sizes, number of slaves, each slaves’ number of rows.

```
int pixel_size=200;
int num_slaves=world_size-1;
int slave_pixel=pixel_size/num_slaves;
```

3. Also, set the value of beta and pi from console and calculate the gamma using pi.

```
double beta=stod(argv[3]);
double pi=stod(argv[4]);
double gamma = 0.5 * log((1.0-pi)/pi);
```

MASTER

If the processor’s rank is 0, it means it is master processor. Master processor reads the input from given file, and shares them to slaves. After sharing, it waits for slaves to terminate, then it merges the results of the slaves’ operations. Then prints the result of the program to the given output file.

1. After these configurations, master process reads input file which is supplied from user via terminal.

```
for(int i=0;i<pixel_size;i++){
  vector<string> words;
  getline(infile, line);
  split(line, words);

  for(int j=0;j<pixel_size;j++)
    input[i][j]= stoi(words[j]);
}
```

2. And then shares the corresponding part of the input to the slaves. And waits the result of the slaves’ execution.

```
  for(int i=1;i<world_size; i++){
    for(int j=0;j<slave_pixel; j++)
      MPI_Send(input[(i-1)*slave_pixel+j], pixel_size, MPI_INT, i, j+500,
MPI_COMM_WORLD);
  }
```

3. When all slaves send the result of their execution, master merges them.

```
  for(int i=1; i<world_size;i++){
    for(int j=0; j<slave_pixel;j++)
      MPI_Recv(input[(i-1)*slave_pixel+j], pixel_size, MPI_INT, i, j+1000,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }
```

4. Then prints the result of program which denoised image to the output file.

```
for(int i=0;i<pixel_size;i++){
  for(int j=0;j<pixel_size;j++)
    output<<input[i][j]<<" ";

  output<<endl;
}
```

SLAVES

If processor’s rank is not equal to 0, it means it is slave processor. Slave processors get their own matrix from master processor and execute MetropolisHastings Algorithm for 1.000.000 times. While doing this, slave communicates with its neighbours in order to get the boundary cells. Then after execution, it sends its result to the master processor.

1. Declarations of local arrays.
   - Upper: Contains bottom row of the previous processor.
   - Lower: Contains top row of the next processor.
   - X: Contains initial matrix, coming from master processor.
   - Z: Contains current matrix, after each iteration.

```
int *upper=NULL;
int *lower=NULL;
upper=(int *)malloc(sizeof(int)*pixel_size);
lower=(int *)malloc(sizeof(int)*pixel_size);
int **X=NULL;
X=(int **)malloc(sizeof(int *)*slave_pixel);
int **Z=NULL;
Z=(int **)malloc(sizeof(int *)*slave_pixel);
```

2. Gets the initial matrix from master processor.

```
for(int i=0;i<slave_pixel;i++){
  X[i] = (int*)malloc(sizeof(int)*pixel_size);
  Z[i] = (int*)malloc(sizeof(int)*pixel_size);
  MPI_Recv(X[i], pixel_size, MPI_INT, 0, i+500, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
}
```

3. First communication done between neighbouring processors. Each processors, send its top row to previous processor, and bottom row to next processor.

```
    if(world_rank!=1)
      MPI_Send(Z[0], pixel_size, MPI_INT, world_rank-1, 2,
MPI_COMM_WORLD);

    if(world_rank!=world_size-1)
      MPI_Send(Z[slave_pixel-1], pixel_size, MPI_INT, world_rank+1, 3,
MPI_COMM_WORLD);

    if(world_rank!=1)
      MPI_Recv(upper, pixel_size, MPI_INT, world_rank-1 , 3,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    if(world_rank!=world_size-1)
      MPI_Recv(lower, pixel_size, MPI_INT, world_rank+1 , 2,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

4. Executes MetropolisHastings Algorithms 1.000.000 times for each processor. Main idea is that, program randomly picks index i and index j and then calculates the delta_E and decide to flip or not flip the pixel according to randomly choosen probabiliy.

```
      index_i = rand() % slave_pixel;
      index_j = rand() % pixel_size;

      sum=0;
      if(index_i==0){
        if(index_j==0){
          sum = Z[0][1] + Z[1][0] + Z[1][1];
          if(world_rank!=1)
            sum+= upper[0]+upper[1];
        }
        else if(index_j==pixel_size-1){
          sum = Z[0][pixel_size-2] + Z[1][pixel_size-1] + Z[1][pixel_size-2];
          if(world_rank!=1)
            sum+=upper[pixel_size-1]+upper[pixel_size-2];
        }
        else{
          sum = Z[0][index_j-1] + Z[0][index_j+1] + Z[1][index_j-1] +
Z[1][index_j] + Z[1][index_j+1];
          if(world_rank!=1)
            sum+=upper[index_j-1]+upper[index_j]+upper[index_j+1];
        }
      }
      else if(index_i==slave_pixel-1){
        if(index_j==0){
          sum = Z[index_i-1][0] + Z[index_i-1][1] + Z[index_i][1];
          if(world_rank!=world_size-1)
          sum+=lower[0]+lower[1];
      }
      else if(index_j==pixel_size-1){
        sum = Z[index_i-1][pixel_size-1] + Z[index_i-1][pixel_size-2] +
Z[index_i][pixel_size-2];
        if(world_rank!=world_size-1)
          sum+=lower[pixel_size-1]+lower[pixel_size-2];
      }
      else{
        sum = Z[index_i][index_j-1] + Z[index_i][index_j+1] + Z[index_i-
1][index_j-1] + Z[index_i-1][index_j] + Z[index_i-1][index_j+1];
        if(world_rank!=world_size-1)
          sum+=lower[index_j-1]+lower[index_j]+lower[index_j+1];
      }
    }
    else{
      if(index_j==0)
        sum = Z[index_i-1][0] + Z[index_i-1][1] + Z[index_i][1] +
Z[index_i+1][1] + Z[index_i+1][0];
      else if(index_j==pixel_size-1)
        sum = Z[index_i-1][index_j] + Z[index_i-1][index_j-1] +
Z[index_i][index_j-1] + Z[index_i+1][index_j-1] + Z[index_i+1][index_j];
      else{
        sum+= Z[index_i-1][index_j-1]+Z[index_i-1][index_j]+Z[index_i-
1][index_j+1]+Z[index_i][index_j-
1]+Z[index_i][index_j+1]+Z[index_i+1][index_j-
1]+Z[index_i+1][index_j]+Z[index_i+1][index_j+1];
      }
    }

    delta_E = -2*gamma*X[index_i][index_j]*Z[index_i][index_j] -
2*beta*Z[index_i][index_j]*sum;
    prob = (double)(rand()) / RAND_MAX;
    change=0;
    if(log(prob) < delta_E){
      Z[index_i][index_j]*=-1;
      change=1;
    }
```

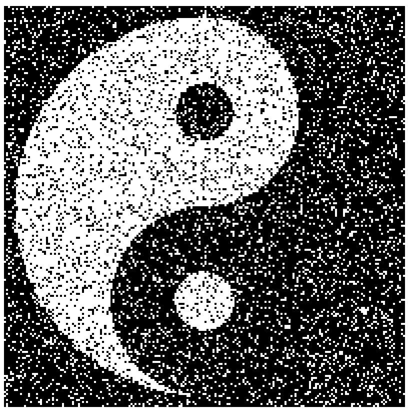
5. After each iteration, communication done between neighbour processors using sendingMessage and receivedMessage arrays. Each array contains 3 elements, first element is whether the flip occured or not, second element is randomly choosen index i, third element is randomly choosen index j. According to the flip, the processor updates its upper or lower parts.

```
      sendingMessage[0]=change;
      sendingMessage[1]=index_i;
      sendingMessage[2]=index_j;

      if(world_rank!=1)
        MPI_Send(sendingMessage, 3, MPI_INT, world_rank-1, 4,
MPI_COMM_WORLD);

      if(world_rank!=world_size-1)
        MPI_Send(sendingMessage, 3, MPI_INT, world_rank+1, 5,
MPI_COMM_WORLD);

      if(world_rank!=1){
        MPI_Recv(receivedMessage, 3, MPI_INT, world_rank-1 , 5,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        if(receivedMessage[0])
          if(receivedMessage[1]==slave_pixel-1)
            upper[receivedMessage[2]]*=-1;
      }

      if(world_rank!=world_size-1){
        MPI_Recv(receivedMessage, 3, MPI_INT, world_rank+1 , 4,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        if(receivedMessage[0])
          if(receivedMessage[1]==0)
            lower[receivedMessage[2]]*=-1;
      }
```

6. Sends the result of the iterations to the master processor.

```
for(int i=0;i<slave_pixel;i++)
      MPI_Send(Z[i], pixel_size, MPI_INT, 0, i+1000, MPI_COMM_WORLD);
```

Examples



Improvements and Extensions

The program that I implemented at first terminated execution in 30seconds, since it had lots of dummy statements which makes my program slower. Actually, the program gets slower due to the many factors.

One of that is the number of iterations that each slave does. Since the number of iterations are mostly depends on the pixel size of input image, 500.000 iterations are enough for the 200x200 pixels. But I want to guarentee and I fixed it to 1.000.000 which is still large but enough for image to be denoised.

Other factor is how communication between neighbour processors are done. Firstly, I sent top row and bottom row of each processors to the neighbours after each iterations. But it takes some time, then I choose the send only the randomly choosen pixel and whether I flip or not flip the that pixel to the neighbours. It makes my program much faster.

Also, the program may get much faster if we make some good assumptions. Since the randomly choosen pixel is not always in the boundary cell, we do not have to make communication with neighbours after each iterations. Slave processors has (200/(N-1)) rows, and only 2 is in boundary cell. The probability that the randomly choosen pixel is in boundary cell is (2*(N-1)/200). So we can communicate with neighbours in every (200/(2*(N-1))) iteration.

Difficulties Encountered

Before starting to implement my code, I had difficulty in installing openmpi. Then I solved it with the help of the internet. While doing my project, communication with neighbours processors is really hard for me to avoid deadlocks.

Conclusion

I implemented the expected project, which makes noisy image to denoised one. The algorithm works almost in 5 seconds, which is really fast in my opinion. Moreover, with the help of this project, now I have better understandings of how parallel architectures works.

Appendicies

```
/*
  Student Name: Gurkan Demir
  Stundet Number: 2015400177
  Compile Status: Compiling
  Program Status: Working
  Notes: Compile with the command "mpic++ -std=c++11 -g main.cpp"
        Then execute with the command "mpiexec --oversubscribe -n [N] ./a.out [input_file] [output_file] [beta]
[pi]"
*/
#include <mpi.h>
#include <cmath>
#include <vector>
#include <fstream>
#include <sstream>
#include <iterator>
using namespace std;

/*
  Method in order to split lines while reading input file.
*/
template <class Container>
void split(const string& str, Container& cont)
{
    istringstream iss(str);
    copy(istream_iterator<string>(iss),
         istream_iterator<string>(),
         back_inserter(cont));
}

/*
  Main function; master, which has rank 0, reads the input file and shares them accross to slaves.
  Then slaves do some calculations to denoise image, then slaves sends it to master process. Then master merges
and prints the result to output file.
*/
int main(int argc, char *argv[]){

/*
  Initializes the MPI.
*/
  MPI_Init(NULL, NULL);

  int world_size;
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);

  int world_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

/*
  Declarations of pixel size(200), number of slaves(total number of processors -1),
  slave pixel(each slaves' number of rows)
  Reading beta, pi from console and calculates gamma using pi.
  Initializes T, which is how many executions slaves must do.
*/
  int pixel_size=200;
  int num_slaves=world_size-1;
  int slave_pixel=pixel_size/num_slaves;
  double beta=stod(argv[3]);
  double pi=stod(argv[4]);
  double gamma = 0.5 * log((1.0-pi)/pi);
  int T=1000000;

/*
  If it is master process, reads the input from file. And shares them to slaves.
  Then it merges results of slaves' operations. Then prints the result to the output file.
*/
  if(world_rank == 0){
    fstream infile(argv[1]);
    string line="";
    int input[pixel_size][pixel_size];

/*
  Reads input file.
*/
    for(int i=0;i<pixel_size;i++){
      vector<string> words;
      getline(infile, line);
      split(line, words);

      for(int j=0;j<pixel_size;j++)
        input[i][j]= stoi(words[j]);
    }

/*
  Shares input to the slaves.
*/
    for(int i=1;i<world_size; i++){
      for(int j=0;j<slave_pixel; j++)
        MPI_Send(input[(i-1)*slave_pixel+j], pixel_size, MPI_INT, i, j+500, MPI_COMM_WORLD);
    }

/*
  Receives result of slaves' operations and merges them.
*/
    for(int i=1; i<world_size;i++){
      for(int j=0; j<slave_pixel;j++)
        MPI_Recv(input[(i-1)*slave_pixel+j], pixel_size, MPI_INT, i, j+1000, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
    }

    ofstream output;
    output.open(argv[2]);

/*
  Prints the result to the output file.
*/
    for(int i=0;i<pixel_size;i++){
      for(int j=0;j<pixel_size;j++)
        output<<input[i][j]<<" ";

      output<<endl;
    }

    output.close();
  }

/*
  Slaves function; gets own matrix from master. Then does 1.000.000 executions to whether flip or not flip
  the pixel with communicating with other slaves. Then sends result to the master processor.
```

```
*/
  else{
/*
  Declarations of arrays.
  Upper: Contains bottom row of the previous processor.
  Lower: Contains top row of the next processor.
  X: Contains initial matrix, coming from master processor.
  Z: Contains current matrix, after each execution.
*/
    int *upper=NULL;
    int *lower=NULL;
    upper=(int *)malloc(sizeof(int)*pixel_size);
    lower=(int *)malloc(sizeof(int)*pixel_size);
    int **X=NULL;
    X=(int **)malloc(sizeof(int *)*slave_pixel);
    int **Z=NULL;
    Z=(int **)malloc(sizeof(int *)*slave_pixel);

/*
  Gets initial matrix from master processor.
*/
    for(int i=0;i<slave_pixel;i++){
      X[i] = (int*)malloc(sizeof(int)*pixel_size);
      Z[i] = (int*)malloc(sizeof(int)*pixel_size);
      MPI_Recv(X[i], pixel_size, MPI_INT, 0, i+500, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    for(int i=0;i<slave_pixel;i++)
      for(int j=0;j<pixel_size;j++)
        Z[i][j]=X[i][j];

/*
  First communication done with neighbour processors.
*/
    if(world_rank!=1)
      MPI_Send(Z[0], pixel_size, MPI_INT, world_rank-1, 2, MPI_COMM_WORLD);

    if(world_rank!=world_size-1)
      MPI_Send(Z[slave_pixel-1], pixel_size, MPI_INT, world_rank+1, 3, MPI_COMM_WORLD);

    if(world_rank!=1)
      MPI_Recv(upper, pixel_size, MPI_INT, world_rank-1 , 3, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

    if(world_rank!=world_size-1)
      MPI_Recv(lower, pixel_size, MPI_INT, world_rank+1 , 2, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

/*
  Arrays in order make communication faster.
  Contains 3 elements; index0: whether we flip the pixel or not
  index1: i'th index
  index2: j'th index
*/
    int *sendingMessage=NULL;
    int *receivedMessage=NULL;
    sendingMessage=(int *)malloc(sizeof(int)*3);
    receivedMessage=(int *)malloc(sizeof(int)*3);

    srand(time(NULL));
    int index_i, index_j, sum=0, change=0;
    double delta_E, prob;

/*
  Executes 1.000.000 times for each processor. Randomly picks index i and j and then
  calculates delta_E and flip or not flip the pixel according to randomly choosen probability.
*/
    for(int i=0;i<T;i++){
      index_i = rand() % slave_pixel;
      index_j = rand() % pixel_size;

/*
  Calculates the sum of surroundings pixels.
*/
      sum=0;
      if(index_i==0){
        if(index_j==0){
          sum = Z[0][1] + Z[1][0] + Z[1][1];
          if(world_rank!=1)
            sum+= upper[0]+upper[1];
        }
        else if(index_j==pixel_size-1){
          sum = Z[0][pixel_size-2] + Z[1][pixel_size-1] + Z[1][pixel_size-2];
          if(world_rank!=1)
            sum+=upper[pixel_size-1]+upper[pixel_size-2];
        }
        else{
          sum = Z[0][index_j-1] + Z[0][index_j+1] + Z[1][index_j-1] + Z[1][index_j] + Z[1][index_j+1];
          if(world_rank!=1)
            sum+=upper[index_j-1]+upper[index_j]+upper[index_j+1];
        }
      }
      else if(index_i==slave_pixel-1){
        if(index_j==0){
          sum = Z[index_i-1][0] + Z[index_i-1][1] + Z[index_i][1];
          if(world_rank!=world_size-1)
            sum+=lower[0]+lower[1];
        }
        else if(index_j==pixel_size-1){
          sum = Z[index_i-1][pixel_size-1] + Z[index_i-1][pixel_size-2] + Z[index_i][pixel_size-2];
          if(world_rank!=world_size-1)
            sum+=lower[pixel_size-1]+lower[pixel_size-2];
        }
        else{
          sum = Z[index_i][index_j-1] + Z[index_i][index_j+1] + Z[index_i-1][index_j-1] + Z[index_i-1][index_j]
+ Z[index_i-1][index_j+1];
          if(world_rank!=world_size-1)
            sum+=lower[index_j-1]+lower[index_j]+lower[index_j+1];
        }
      }
      else{
        if(index_j==0)
          sum = Z[index_i-1][0] + Z[index_i-1][1] + Z[index_i][1] + Z[index_i+1][1] + Z[index_i+1][0];
        else if(index_j==pixel_size-1)
          sum = Z[index_i-1][index_j] + Z[index_i-1][index_j-1] + Z[index_i][index_j-1] + Z[index_i+1][index_j-
1] + Z[index_i+1][index_j];
        else{
          sum+= Z[index_i-1][index_j-1]+Z[index_i-1][index_j]+Z[index_i-1][index_j+1]+Z[index_i][index_j-
1]+Z[index_i][index_j+1]+Z[index_i+1][index_j-1]+Z[index_i+1][index_j]+Z[index_i+1][index_j+1];
        }
      }

/*
  Calculates delta_E and decide to flip or not flip according to randomly choosen probability.
*/
      delta_E = -2*gamma*X[index_i][index_j]*Z[index_i][index_j] -2*beta*Z[index_i][index_j]*sum;
      prob = (double)(rand()) / RAND_MAX;
      change=0;
      if(log(prob) < delta_E){
        Z[index_i][index_j]*=-1;
        change=1;
      }

/*
  After each execution, communication done with neighbour processors.
*/
      sendingMessage[0]=change;
      sendingMessage[1]=index_i;
      sendingMessage[2]=index_j;

      if(world_rank!=1)
        MPI_Send(sendingMessage, 3, MPI_INT, world_rank-1, 4, MPI_COMM_WORLD);

      if(world_rank!=world_size-1)
        MPI_Send(sendingMessage, 3, MPI_INT, world_rank+1, 5, MPI_COMM_WORLD);

      if(world_rank!=1){
        MPI_Recv(receivedMessage, 3, MPI_INT, world_rank-1 , 5, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        if(receivedMessage[0])
          if(receivedMessage[1]==slave_pixel-1)
            upper[receivedMessage[2]]*=-1;
      }

      if(world_rank!=world_size-1){
        MPI_Recv(receivedMessage, 3, MPI_INT, world_rank+1 , 4, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        if(receivedMessage[0])
          if(receivedMessage[1]==0)
            lower[receivedMessage[2]]*=-1;
      }
    }

/*
  Send the result to the master processor.
*/
    for(int i=0;i<slave_pixel;i++)
      MPI_Send(Z[i], pixel_size, MPI_INT, 0, i+1000, MPI_COMM_WORLD);
  }

/*
  Finalizes the MPI and process.
*/
  MPI_Finalize();
  return 0;
}
```