CMPE 300 PROJECT 1 DOCUMENTATION

Course Id: CMPE300

Course Name: Analysis of Algorithms

Name of the Student: Fahri Can Şanlı

Student Number: 2015400096

Name of the Submitted Person: Tunga Güngör

Project Title: Parallel Programming

Type of the Project: Programming Project

Submission Date: 26/12/2018

1-Introduction:

In this project, I implemented a parallel programming project by using C++. To achieve parallel programming, I used MPI library. I downloaded it from the documents provided. The main objective of this project was to design a parallel algorithm for image denoising with the Ising model using Metropolis-Hastings algorithm.

Ising model says that there are two states (+1 or -1) and each thing interacts with its neighbors. In our project, it means that if we select a random black pixel, it is more likely that this pixel is surrounded by black pixels. The same rule is also valid for white pixels.

Metropolis-Hastings algorithm is an algorithm to denoise the noised image. It converts the noised image X to denoised image Z. Its procedure can be listed as taking random pixel, calculating acceptance probability for that pixel, then accepting that new image Z with that acceptance probability. This algorithm requires repeating this procedure until it converges.

2-Program Interface:

This program does not require lots of user interaction. User just needs to compile this program to produce an executable, then he/she needs to run the executable.

To compile program user needs to enter:

```
 Mpic++ -g [projectName.cpp] -o [executableName]
```

To run executable user needs to enter:

```
Mpiexec -n [processorNumber] [executableName] [inputFile] [outputFile] [Beta] [Pi]
```

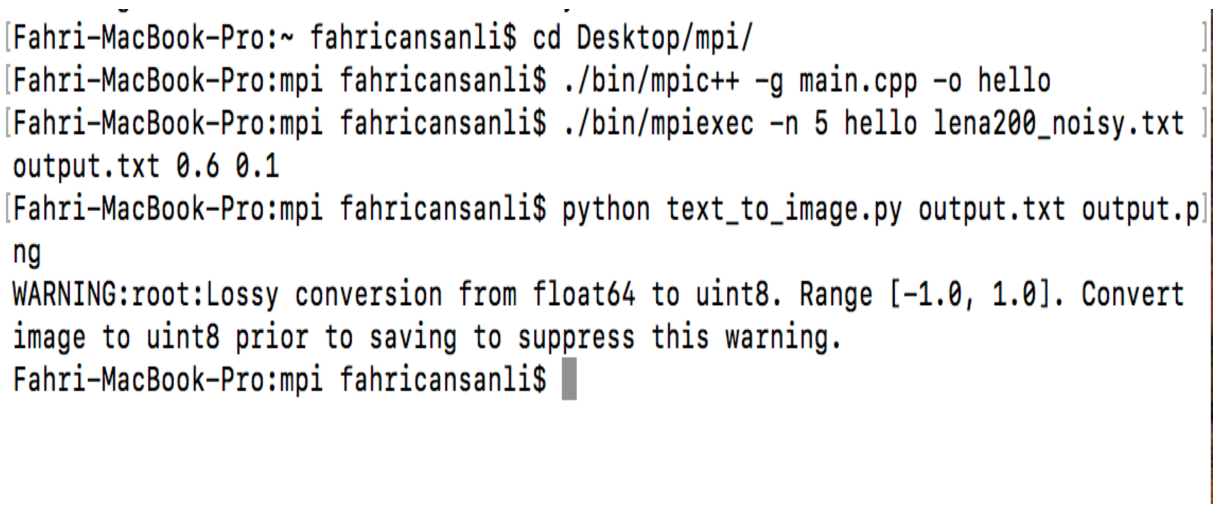
Beta and pi parameters are used in parallel algorithm.

Running executable produces an output file in form of txt and user needs to run a python command to produce an image from that output file.

To produce an image from that output file:

```
python [pythonCode] [outputFile] [imageName.png]
```

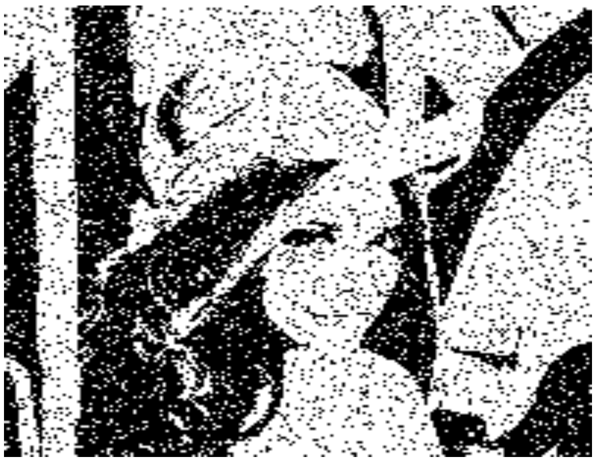
To terminate program, actually user do not need to do particular thing. After image produced, whole program finishes. User just needs to click close button to close image.

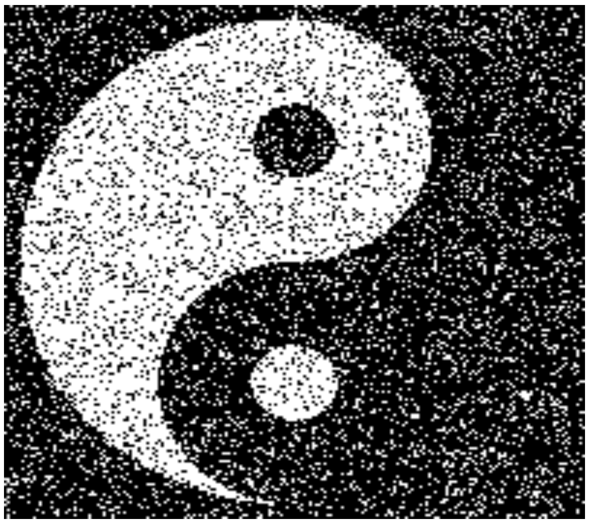


```
[Fahri-MacBook-Pro:~ fahricansanli$ cd Desktop/mpi/                          ]
[Fahri-MacBook-Pro:mpi fahricansanli$ ./bin/mpic++ -g main.cpp -o hello       ]
[Fahri-MacBook-Pro:mpi fahricansanli$ ./bin/mpiexec -n 5 hello lena200_noisy.txt ]
 output.txt 0.6 0.1
[Fahri-MacBook-Pro:mpi fahricansanli$ python text_to_image.py output.txt output.p]
 ng
 WARNING:root:Lossy conversion from float64 to uint8. Range [-1.0, 1.0]. Convert
 image to uint8 prior to saving to suppress this warning.
 Fahri-MacBook-Pro:mpi fahricansanli$ 
```

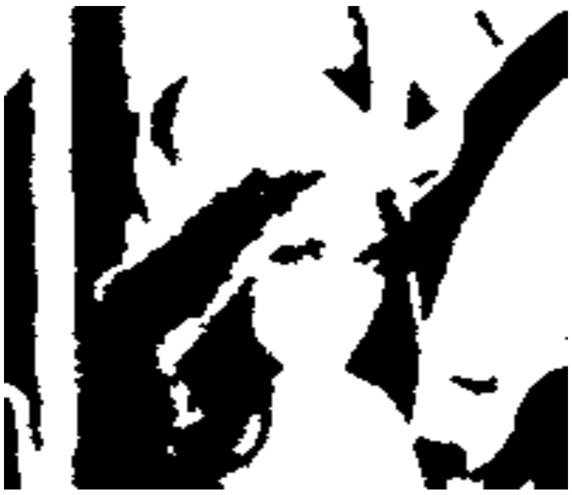
3-Program Execution:

As explained in above parts, the main objective of this program is to produce a denoised image from a noised image and this program achieves this by using parallel algorithm logic. To be consistent with objective of program, user needs to give noised images as input. These noised images can be obtained by flipping some pixels of original image. Following images can be shown as example inputs.





To obtain these noised images, user should use the python code provided and run "python [pythonCode] [originalImage] [noisedImage]" command in the command line. As expected, output of the program is denoised version of input image. Our objective is to produce an image which is highly similar to original image. But, since this program depends on random flipping strategies, it might not produce original image exactly. Following images can be shown as example output images.



4-Input and Output:

Until this point, it is explained as user gives an image as input and obtain an image as output. But this is not the case. Actually, user gives an input in the form of txt which contains 1 and -1 in each cell. We can think of our txt file as a two-dimensional array. Each cell contains either 1 or -1 which represent either black or white pixels. After running program, user obtains an output in the form of txt as well. The output file also contains 1 and -1. The difference between input and output file is output file produced from input file by using Metropolis-Hastings algorithm and output files contains more cells that have neighbor cells similar to itself.

A part of noisy input file

```
1 1 1 1 1 1 1 1 1 1 1 1 −1 1 1 1 1 1 1 1 1 1 1 1 1 −1 −1 −1 −1 1 −1 −1 −1 1 −1 −1 −1 −1 −1
−1 −1 −1 −1 1 1 −1 −1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 −1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 −1 1 1 1 −1 1 1 1 1 1 1 1 1 1 1 −1 −1
−1 −1 1 1 1 1 1 1 −1 1 1 −1 −1 1 1 1 1 −1 1 1 1 1 1 1 1 −1 1 −1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 −1 −1 −1 1 1 1 1 −1 1 1 −1 1 1 1 −1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 −1 1 −1 1 1 1 1 1 1 −1 1 1 1 1 −1 1 −1 −1 −1 1 −1 1 −1 −1 −1 −1 −1 −1 −1 −1
−1 −1 −1 −1 −1 1 1 1 1 −1 1 1 1 1 1 1 1 −1 1 1 1 1 1 1 1 1 −1 1 1 1 1 1 −1 1 −1 1 −1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 −1 1 1 1 1 1 1 1 1 1 1 1 −1 1 1 1 1 1 1 1 1 1 1 1 1 −1 1
1 −1 −1 −1 1 1 1 1 1 1 1 1 1 1 1 −1 1 −1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
−1 1 1 −1 1 −1 1 1 −1 −1 −1 1 1 1 −1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 −1 1 1
1 1 1 1 1 1 1 1 1 −1 1 1 −1 1 1 1 1 1 1 1 1 1 1 −1 −1 −1 −1 −1 −1 −1 −1 −1 1 −1 1 −1 1 −1
−1 −1 1 −1 1 1 1 −1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 −1 1 1 1 1 1 1 1 1 1 1 1 −1 1 1 1 −1 1 1 1
1 1 1 1 1 −1 −1 1 1 1 1 1 1 1 1 1 1 −1 1 1 1 1 1 1 −1 −1 1 1 −1 1 −1 1 1 −1 1 1 1 1 1 1 1
1 −1 −1 −1 −1 1 1 1 1 −1 1 1 1 1 1 −1 1 1 1 1 1 1 1 −1 1 1 1 1 −1 −1 1 1 −1 1 1 1 −1 1 1 1
1 1 1 −1 1 −1 1 −1 −1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 −1 1 1 1 1 −1 −1 −1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 −1 1 1 1 1 1 1 1 1 −1 −1 −1 −1 −1 −1 −1 −1 −1 −1 −1 −1 −1 −1
−1 −1 −1 1 1 1 1 −1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 −1 1
−1 1 1 1 −1 −1 1 1 −1 1 −1 1 1 1 −1 1 1 1 1 −1 1 1 1 1 1 1 1 1 1 1 1 1 1 −1 1 1 1 1 −1 1 1
1 −1 −1 −1 −1 −1 1 1 −1 1 1 1 1 −1 1 1 1 1 1 1 1 1 1 1 −1 1 1 −1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 −1 −1 −1 −1 −1 1 1 1 −1 1 1 1 1 1 1 1 1 1 −1 −1 1 1 1 1 1 1 1 1 1 1 −1 −1 −1 −1
−1 1 1 −1 1 1 1 1 1 1 1 −1 1 1 1 1 −1 1 1 1 1 1 1 −1 −1 −1 −1 −1 −1 −1 −1 −1 1 −1 −1 −1 −1
−1 −1 −1 −1 −1 1 1 1 −1 1 1 1 −1 −1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 −1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 −1 1 1 1 1 −1 −1 1 1 1 1 1 1
−1 1 −1 1 −1 −1 1 −1 1 1 −1 1 1 1 1 1 1 1 1 1 1 1 1 1 −1 1 −1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 −1 −1 −1 −1 1 1 −1 1 1 1 1 1 1 1 1 −1 1 −1 1 1 1 1 1 1 1 1 1 1 −1 1 −1 −1 −1
1 1 1 1 1 1 1 −1 1 1 1 1 −1 1 1 1 1 1 1 1 1 −1 1 −1 −1 1 −1 −1 −1 −1 −1 −1 −1 −1 −1 −1 −1
−1 −1 −1 −1 −1 1 1 1 1 1 1 1 1 1 1 1 1 −1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 −1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 −1 1 1 1 1 1 1
```

A part of denoised output file

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1
-1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 -1 -1
-1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1
-1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 -1 -1
-1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1
-1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 -1 -1
-1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 -1 -1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1
-1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 -1
-1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 -1 -1
-1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 -1
-1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 -1 -1
-1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
-1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1 -1
```

5-Program Structure:

5.1- First Approach:

In this approach, I only used only main.cpp. Firstly, I read beta and pi values from the user and calculate gamma value by using them. Then, I initiated the MPI environment and determine the number of processors that are going to be used in advance. After this step, in the part of master processor, I read each line from the input file and create two-dimensional array representing the noised image and send data to each slave processor and wait to receive edited version of each region. Then, master process opens the output file and writes the last version of input image to output file.

In the case of slave processors, I classified them into three categories.

5.1.1- Rank == 1:

Rank ==1 means that this is the processor responsible from the top region of image. At the beginning, it receives the necessary data from the master processor and place it into temporary two-dimensional array of X and Z. Then, it enters a for loop. Since it is responsible from the top region, it needs to communicate with the processor below it for its lowest row. To prevent deadlock, it first receives from the processor below it and then sends to it. After data transaction, it randomly produces i and j value corresponding to a cell and then calculates the sum of neighbor cells. If i is equal to last row, it uses the data came from the processor below it. After this step, it calculates the acceptance probability by using the equation provided in the project description and produce a random number. If acceptance probability is higher than the produced random number, it flips the pixel and passes to the next iteration. Otherwise, it directly passes to the next iteration.

5.1.2- Rank == N:

Rank ==N means that this is the processor responsible from the last region of image. At the beginning, it receives the necessary data from the master processor and place it into temporary two-dimensional array of X and Z. Then, it enters a for loop. Since it is responsible from the last region, it needs to communicate with the processor above it for its uppest row. To prevent deadlock, it first sends to the processor above it and then receive from it. After data transaction, it randomly produces i and j value corresponding to a cell and then calculates the sum of neighbor cells. If i is equal to first row, it uses the data came from the processor above it. After this step, it calculates the acceptance probability by using the equation provided in the project description and produce a random number. If acceptance probability is higher than the produced random number, it flips the pixel and passes to the next iteration. Otherwise, it directly passes to the next iteration.

5.1.3- Rank != 1 and Rank != N:

This situation represents the processors between the first and last processors. This situation has also the same procedure. Only difference is that these processors communicates with both processor above it and processor below it. Therefore, they hold two extra arrays. To avoid deadlock, they first send to processors above them and then receive from them. After, they receive from the processors below them and then send them. The rest is the same.

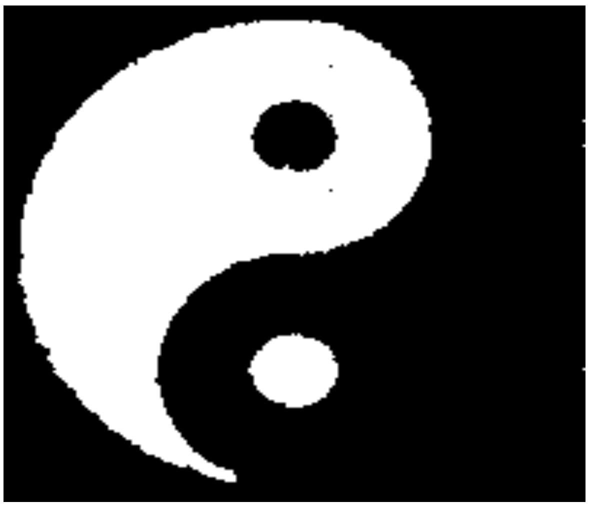
5.2- Second Approach:

For the second approach, I used only main.cpp as well. The general idea is similar to the implementation of first approach. I used again master processor to do the same operations such as reading from file, communicating with slave processors and printing to output file. The differences between first approach and second approach are that I sent data from master processor to each slave processor row by row and I used different communication style in the second version to prevent deadlock. I classified communication type into 4 different categories. First one is from left to right. Second one is from up to down. Third one is from left upper corner to right bottom corner and the last one is from right upper corner to left bottom corner. Besides, for second, third and fourth communication types to prevent deadloack, I gave priority to the processor which locates at the higher position in the tableau because each communication event can occur between two processor. For the first case, I gave priority to the processor leftmost. By priority, I mean to receive first and then send.

After configuring communication, I classsifed slave processors into categories and for each category I selected a random pixel and calculated sum of neighbor cells.

6-Examples:

5 processors with beta=0.6 and pi=0.1



21 processors with beta=0.6 and pi=0.1

7-Improvements and Extensions:

In my implementation, each processor sends to the other processors and receives data from the other processors at each iteration. Normally, this is redundant, but I could not achieve to implement it in a way that processors communicate only when they need. So, this aspect of my implementation can be improved.

8-Difficulties Encountered:

In this project, I encountered with some difficulties. Firstly, installing MPI on my computer was a little bit hard for me, but in the end, I achieved to install it. Besides, understanding how to use the MPI library was hard for me, but practicing codes from PS was very beneficial for me.

9-Conclusion:

This project was a very good experience for me. Learning how to use parallel programming and performing parallel programming for image denoising was very exciting. My program compiles correctly and produces correct output.

10- Appendices:

```
/*
Student Name: Fahri Can Şanlı
Student Number: 2015400096
Compile Status: Compiling
Program Status: Working
*/

#include <mpi.h>
#include <iostream>
#include <stdio.h>
#include <vector>
#include <queue>
#include <sstream>
#include <fstream>
#include <algorithm>
#include <iomanip>
#include <iterator>
#include <stdlib.h>
#include <string>
#include <math.h>
#include <ctime>
using namespace std;

//To split lines read by the master processor
template <class Container>
void split1(const string& str, Container& cont)
{
    istringstream iss(str);
    copy(istream_iterator<string>(iss),
         istream_iterator<string>(),
         back_inserter(cont));
}

int main(int argc, char* argv[]) {
    //To produce random numbers
    srand((unsigned)time(NULL));
    //Rank of each processor and total size
    int rank, size;
    double beta, pi, gamma;
    string be = argv[3];
    string p = argv[4];
    beta = atof(be.c_str());
    pi = atof(p.c_str());
    gamma = 0.5 * log((1-pi)/pi);
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    //Number of slave processor
    int N = size -1;
    //How many row each slave processor will process
    int row= 200/N;
    //In total how many pixels each slave processor will process
    int data = 200 * row;
    //Total iteration amount
    int T=500000;
    //Denotes master process
    if(rank == 0){
        //Noisy image
        int xPicture [200][200];
        //Resulting image
        int result[200][200];
        int lin=0;
        //Reads from input file
        ifstream infile(argv[1]);
        string line;
        while(getline(infile,line)){
            vector<string> words;
            split1(line,words);
            for(int i=0;i<words.size();i++){
                xPicture[lin][i]=atoi(words[i].c_str());
            }
            lin++;
        }
        //Sends to each slave processor
        for(int i = 1 ; i <= N ; i++){
            MPI_Send(xPicture[row*(i-1)], data, MPI_INT, i, 0,
MPI_COMM_WORLD);
        }
        //Receives from each processor in order and place them into result
array
        for(int t=1;t<=N;t++){
            int* subarr1 = NULL;
            subarr1 = (int *)malloc(sizeof(int) * data);
            MPI_Recv(subarr1, data, MPI_INT, t, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            for(int i=0;i<row;i++){
                for(int j=0;j<200;j++){
                    result[i+(t-1)*row][j] = subarr1[i*200 + j];
                }
            }
            delete subarr1;
        }
        //Writes to output file
        ofstream myfile;
        myfile.open(argv[2]);
        for(int i=0;i<200;i++){
            for(int j=0;j<199;j++){
                myfile << result[i][j] << " ";

            }
            myfile << result[i][199] << endl;
        }
    }
    //Represents the slave processor at the top
    else if(rank == 1){
        //At first receive from master processor and replace each pixel
temporary X image array and temporary Z image array
        int* subarr = NULL;
        subarr = (int *)malloc(sizeof(int) * data);
        MPI_Recv(subarr, data, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        int tempX[row][200];
        int tempZ[row][200];
        int lower[200];
        for(int i=0;i<data;i++){
            tempX[i/200][i%200]=subarr[i];
            tempZ[i/200][i%200]=subarr[i];
        }
        delete subarr;
        //Start to iterate
        for(int a=0;a<(T/N);a++){
            //Firstly receives from the processor below it and then sends
to it
            if(N!=1){
                int* sub = NULL;
                sub = (int *)malloc(sizeof(int) * 200);
                MPI_Recv(sub, 200, MPI_INT, 2, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
                for(int s=0;s<200;s++){
                    lower[s]=sub[s];
                }
                delete sub;
                MPI_Send(tempZ[row-1], 200, MPI_INT, 2, 0, MPI_COMM_WORLD);
            }
            //Chooses a random pixel
            int i = rand() % row;
            int j= rand() % 200;
            int sum=0;
            //Calculates the sum of neighbor cells
            for(int e=i-1;e<=i+1;e++){
                for(int f=j-1;f<=j+1;f++){
                    if(i!=row-1 && e>=0 && e<=row-1 && f>=0 && f<=199){
                        sum+=tempZ[e][f];
                    }
                    if(i==row-1 && e<=row-1 && f>=0 && f<=199){
                        sum+=tempZ[e][f];
                    }
                    if(i==row-1 && e==row && f>=0 && f<=199 && N!=1){
                        sum+=lower[f];
                    }
                }
            }
            sum-=tempZ[i][j];
            double alfa = min(1.0 ,exp((-2 * gamma * tempX[i][j] *
tempZ[i][j]) + (-2 * beta * tempZ[i][j] * sum)));
            double random=((double)rand()/(double)RAND_MAX);
            if(random < alfa){
                tempZ[i][j] *= -1;
            }
        }
        //After finishing all iterations sends the final Z image to the
master processor
        MPI_Send(tempZ, data, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    //Represents the slave processor at the down
    else if(rank == N && N!=1){
        //At first receive from master processor and replace each pixel
temporary X image array and temporary Z image array
        int* subarr = NULL;
        subarr = (int *)malloc(sizeof(int) * data);
        MPI_Recv(subarr, data, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        int tempX[row][200];
        int tempZ[row][200];
        int upper[200];
        for(int i=0;i<data;i++){
            tempX[i/200][i%200]=subarr[i];
            tempZ[i/200][i%200]=subarr[i];
        }
        delete subarr;
        //Start to iterate
        for(int a=0;a<(T/N);a++){
            //Firstly sends to the processor above it and then receives
from it
            MPI_Send(tempZ[0], 200, MPI_INT, N-1, 0, MPI_COMM_WORLD);
            int* sub = NULL;
            sub = (int *)malloc(sizeof(int) * 200);
            MPI_Recv(sub, 200, MPI_INT, N-1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            for(int s=0;s<200;s++){
                upper[s]=sub[s];
            }
            delete sub;
            //Chooses a random pixel
            int i = rand() % row;
            int j= rand() % 200;
            int sum=0;
            //Calculates the sum of neighbor cells
            for(int e=i-1;e<=i+1;e++){
                for(int f=j-1;f<=j+1;f++){
                    if(i!=0 && e>=0 && e<=row-1 && f>=0 && f<=199){
                        sum+=tempZ[e][f];
                    }
                    if(i==0 && e>=0 && f>=0 && f<=199){
                        sum+=tempZ[e][f];
                    }
                    if(i==0 && e==-1 && f>=0 && f<=199){
                        sum+=upper[f];
                    }
                }
            }
            sum-=tempZ[i][j];
            double alfa = min(1.0 ,exp((-2 * gamma * tempX[i][j] *
tempZ[i][j]) + (-2 * beta * tempZ[i][j] * sum)));
            double random=((double)rand()/(double)RAND_MAX);
            if(random < alfa){
                tempZ[i][j] *= -1;
            }
        }
        //After finishing all iterations sends the final Z image to the
master processor
        MPI_Send(tempZ, data, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    //Represents the slave processors between the first and the last one
    else if(N!=1 && rank!=1 && rank!=N){
        //At first receive from master processor and replace each pixel
temporary X image array and temporary Z image array
        int* subarr = NULL;
        subarr = (int *)malloc(sizeof(int) * data);
        MPI_Recv(subarr, data, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        int tempX[row][200];
        int tempZ[row][200];
        int upper[200];
        int lower[200];
        for(int i=0;i<data;i++){
            tempX[i/200][i%200]=subarr[i];
            tempZ[i/200][i%200]=subarr[i];
        }
        delete subarr;
        //Start to iterate
        for(int a=0;a<(T/N);a++){
            //Firstly sends to the processor above it and then receives
from it
            MPI_Send(tempZ[0], 200, MPI_INT, rank-1, 0, MPI_COMM_WORLD);
            int* sub = NULL;
            sub = (int *)malloc(sizeof(int) * 200);
            MPI_Recv(sub, 200, MPI_INT, rank-1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            for(int s=0;s<200;s++){
                upper[s]=sub[s];
            }
            delete sub;
            //Secondly receives from the processor below it and then sends
to it
            int* subb = NULL;
            subb = (int *)malloc(sizeof(int) * 200);
            MPI_Recv(subb, 200, MPI_INT, rank+1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            for(int s=0;s<200;s++){
                lower[s]=subb[s];
            }
            delete subb;
            MPI_Send(tempZ[row-1], 200, MPI_INT, rank+1, 0,
MPI_COMM_WORLD);
            //Chooses a random pixel
            int i = rand() % row;
            int j= rand() % 200;
            int sum=0;
            //Calculates the sum of neighbor cells
            for(int e=i-1;e<=i+1;e++){
                for(int f=j-1;f<=j+1;f++){
                    if(i!=row-1 && i!=0 && e>=0 && e<=row-1 && f>=0 &&
f<=199){
                        sum+=tempZ[e][f];
                    }
                    if(i==row-1 && e<=row-1 && f>=0 && f<=199){
                        sum+=tempZ[e][f];
                    }
                    if(i==row-1 && e==row && f>=0 && f<=199){
                        sum+=lower[f];
                    }
                    if(i==0 && e>=0 && f>=0 && f<=199){
                        sum+=tempZ[e][f];
                    }
                    if(i==0 && e==-1 && f>=0 && f<=199){
                        sum+=upper[f];
                    }
                }
            }
            sum-=tempZ[i][j];
            double alfa = min(1.0 ,exp((-2 * gamma * tempX[i][j] *
tempZ[i][j]) + (-2 * beta * tempZ[i][j] * sum)));
            double random=((double)rand()/(double)RAND_MAX);
            if(random < alfa){
                tempZ[i][j] *= -1;
            }
        }
        //After finishing all iterations sends the final Z image to the
master processor
        MPI_Send(tempZ, data, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    MPI_Finalize();

    return 0;
}
```