CmpE300 - Analysis of Algorithms Fall 2018

Student Name: **Cemal Aytekin**

Student ID: **2015400126**

Project Title: **Image Denoising**

Project Type: **Programming Project**

Submitted Person: **Burak Suyunu**

Submission Date: **26.12.2018**

Introduction

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

The Ising model, named after the physicist Ernst Ising, is a mathematical model of ferromagnetism in statistical mechanics. 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.

Program Interface

In this project we are using MPI environment which enable us to program in parallel. Therefore the MPI environment must be set up. I installed Open MPI under Debian 5.0 and Mac OS X 10.6.8. Here are the steps:

1. Download latest stable release of open mpi from http://www.open-mpi.org/software/ompi/v1.4/downloads/openmpi-1.4.4.tar.gz

2. Type these commands to build Open MPI with default settings.

```
>>    tar -xvf openmpi-1.4.4.tar.gz

>>    ./configure

>>    make all install
```

To compile the program with Open MPI type the following command:

```
>> mpic++ -g your_code.cpp -o your_program
```

To run the program with Open MPI type the following command:

```
>> mpiexec -n NUM_PROCESSORS ./your_program INPUT_FILE OUTPUT_FILE BETA PI
```

Program Execution

In this project, the program convert a noisy image to denoisy form. Therefore we need an input file (INPUT_FILE) which includes a 2D array (200 x 200) consists of 1 and -1. We also need the name of the output file (OUTPUT_FILE) in where the program will print the denoisy image's data. Because of that by randomly flipping some of the pixels of Z, we obtain the noisy image X which we are observing. We assume that the noise-free image Z is generated from an Ising model (parametrized with β) and X is generated by flipping a pixel of Z with a small probability π. Therefore, the user should also enter the Beta (BETA) and Pi (PI) values.

Input and Output

The program will read the input from a text file and print the result in another text file. The input text file will be a 2D array representation (200 x 200) of a black and white noisy image.

Program Structure

The program basically consist of three main parts. Initializing MPI, Master Processor and Slave Processors.

1. MPI Initialization

In the main function, after taking the arguments from terminal, firstly I initialized the MPI.

```
int rank, size;

MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

MPI_Comm_size(MPI_COMM_WORLD, &size);
```

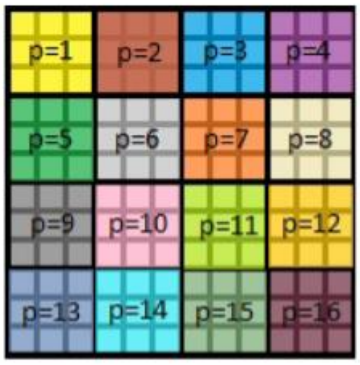
In this code, size keeps the number of total processors. And rank is kind of an id for each processor. Later, we will understand which processor is running by considering this variable. Consequently, we have size - 1 slave processes.

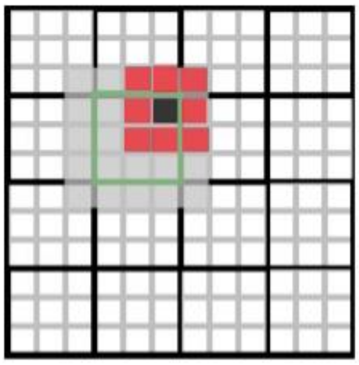
2. Master Processor

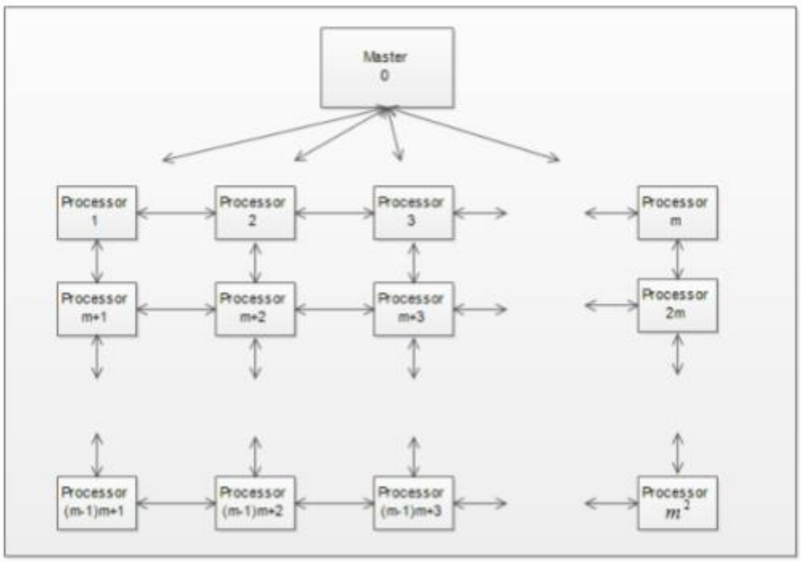
The second part is the implementation of master processor. (if rank==0) The input file is only read by master processor and distributed to slave (rest) processors by the master processor which has rank 0. The whole array is not stored in each processor locally. Therefore in master processor, I divided the 2D array into sub parts in order to send them to the slave processors.

```
for(int i=0; i<N; i++){

    for(int m=0; m<ARRAY_SIZE/n; m++){

        for( int k=0; k<ARRAY_SIZE/n; k++){

                s_subarr[m][k] =
                arr[m+(i/n)*(ARRAY_SIZE/n)][k+(i%n)*(ARRAY_SIZE/n)];

        }

     }


     MPI_Send(s_subarr, (ARRAY_SIZE/n)*(ARRAY_SIZE/n), MPI_INT, i+1, 0,
MPI_COMM_WORLD);

}
```

In this part of the code, I create the subarrays. I am implementing the second approach. In the second approach, the grid is divided into (n/p × n/p) blocks and each process is responsible from a block. Now the updates are more trickier since each process has more than 1 adjacent process.







Then I sent the sub array to the slave processors using MPI_Send function. There are three critical parameter here. The first one is the data, the second one is the size of the data and the final one is the target processor. After sending the data, then the master processor receives the final status of the sub arrays from slave processors. After receiving all the data, then print them into the output file in the right order.

3. Slave Processors

The communication and all the calculations are in slave processors.

```
int X[ARRAY_SIZE/n][ARRAY_SIZE/n];      // sub data, received from master

int Z[ARRAY_SIZE/n][ARRAY_SIZE/n];      // copy of X

int r_upper[ARRAY_SIZE/n];        // received from upper adjacent process

int r_lower[ARRAY_SIZE/n];        // received from lower adjacent process

int r_right[ARRAY_SIZE/n];        // received from right adjacent process
```

```
    int r_left[ARRAY_SIZE/n];         // received from left adjacent process
    int s_upper[ARRAY_SIZE/n];        // send to upper adjacent process
    int s_lower[ARRAY_SIZE/n];        // send to lower adjacent process
    int s_right[ARRAY_SIZE/n];        // send to right adjacent process
    int s_left[ARRAY_SIZE/n];         // send to left adjacent process
    int r_left_top;                   // received from left-top adjacent process
    int r_left_down;                  // received from left-down adjacent process
    int r_right_top;                  // received from right-top adjacent process
    int r_right_down;                 // received from right-down adjacent process
    int s_left_top;                   // send to left-top adjacent process
    int s_left_down;                  // send to left-down adjacent process
    int s_right_top;                  // send to right-top adjacent process
    int s_right_down;                 // send to right-down adjacent process
```

These are all the necessary variables. X is the is the sub array received from master processor. Z is the copy array of the X. All the necessary flips are done on Z. "r_upper, r_lower, r_right, r_left" are the one dimensional arrays. These are the ones that received from four directions. A line from the neighboring processors. "s_upper, s_lower, s_right, s_left" are the one dimensional arrays. These are the ones that send to the four directions. "r_left, r_lower, r_right, r_left" are the one dimensional arrays. These are the ones that received from four directions. A line from the neighboring processors. There are also corners. There are four cases. Left-top, Left-bottom, Right-Top and Right-Bottom. As you can see there are 4 receive and 4 send variables exist in the code. After initializing all of these variables, I do some calculations to determine whether an index is flipped or not. Therefore I have a for-loop which iterates 500.000/N times. In the beginning of the loop, I called the necessary send and receive functions.

```
  // send UPPER
  if(rank>n){
    for(int s=0; s<ARRAY_SIZE/n; s++)
        s_upper[s] = Z[0][s];
    MPI_Send(s_upper, ARRAY_SIZE/n, MPI_INT, rank-n, 0, MPI_COMM_WORLD);
    }


    // send DOWN
    I f(rank<=N-n){
    for(int s=0; s<ARRAY_SIZE/n; s++)
        s_lower[s] = Z[(ARRAY_SIZE/n)-1][s];
    MPI_Send(s_lower, ARRAY_SIZE/n, MPI_INT, rank+n, 0, MPI_COMM_WORLD);
    }
```

```
// send RIGHT
if(rank%n!=0){
    for(int s=0; s<ARRAY_SIZE/n; s++)
        s_right[s] = Z[s][ARRAY_SIZE/n-1];
    MPI_Send(s_right, ARRAY_SIZE/n, MPI_INT, rank+1, 0, MPI_COMM_WORLD);
}

    // send LEFT
    if(rank%n!=1){
      for(int s=0; s<ARRAY_SIZE/n; s++)
          s_left[s] = Z[s][0];
      MPI_Send(s_left, ARRAY_SIZE/n, MPI_INT, rank-1, 0, MPI_COMM_WORLD);
    }

    // send left-top
    if(rank%n!=1 && rank>n){
      s_left_top = Z[0][0];
      MPI_Send(&s_left_top, 1, MPI_INT, rank-n-1, 0, MPI_COMM_WORLD);
    }

    // send left-down
    if(rank%n!=1 && rank<=N-n){
      s_left_down = Z[(ARRAY_SIZE/n)-1][0];
      MPI_Send(&s_left_down,1, MPI_INT, rank+n-1, 0, MPI_COMM_WORLD);
    }

    // send right-top
    if(rank%n!=0 && rank>n){
      s_right_top = Z[(ARRAY_SIZE/n)-1][0];
      MPI_Send(&s_right_top,1, MPI_INT, rank-n+1,  0, MPI_COMM_WORLD);
    }

    // send right-down
    if(rank%n!=0 && rank<=N-n){
      s_right_down = Z[(ARRAY_SIZE/n)-1][(ARRAY_SIZE/n)-1];
      MPI_Send(&s_right_down,1, MPI_INT, rank+n+1, 0, MPI_COMM_WORLD);
    }
```

```
          // receive upper

          if(rank>n)

            MPI_Recv(r_upper, ARRAY_SIZE/n, MPI_INT, rank-n, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);


          // receive lower

          if(rank <= N-n)

            MPI_Recv(r_lower, ARRAY_SIZE/n, MPI_INT, rank+n, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);


          // receive right

          if(rank%n!=0)

            MPI_Recv(r_right, ARRAY_SIZE/n, MPI_INT, rank+1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);


          // receive left

          if(rank%n!=1)

            MPI_Recv(r_left, ARRAY_SIZE/n, MPI_INT, rank-1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);


          // receive left-top

          if(rank%n!=1 && rank>n)

            MPI_Recv(&r_left_top, 1, MPI_INT, rank-n-1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);


          // send left-bottom

          if(rank%n!=1 && rank<=N-n){

            MPI_Recv(&r_left_down, 1, MPI_INT, rank+n-1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

          }


          // send right-top

          if(rank%n!=0 && rank>n){

            MPI_Recv(&r_right_top, 1,  MPI_INT, rank-n+1,  0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

          }


          // send right-down
          if(rank%n!=0 && rank<=N-n){
            MPI_Recv(&r_right_down, 1, MPI_INT, rank+n+1, 0, MPI_COMM_WORLD, 
MPI_STATUS_IGNORE);
          }
```

After that, I generate two random indexes to choose an index from the Z.

```
      int i = rand()%(ARRAY_SIZE/n)+0;
      int j = rand()%(ARRAY_SIZE/n)+0;
```

We need to sum of the values of the 8 neighboring index of the choosing one. Therefore, we need to find all of them. But some of them are in borders. Some of them does not exist. Therefore, we need to check all these circumstances.

```
      int sum = 0;              // keeps the sum of the indexes surraounding
      int value = 0;            // value of that index
          for(int k=-1; k<=1; k++){
            for(int m=-1; m<=1; m++){
                value = 0;
                if(m==0 && k==0)
                  continue;

                // left-top
                else if( j+m == -1 && i+k == -1 && rank%n!=1 && rank>n)
                  value = r_left_top;

                // left-bottom
                else if( j+m == -1 && i+k == ARRAY_SIZE/n && rank%n!=1 && rank<=N-
n)
                  value = r_left_down;

                // right-top
                else if( j+m == ARRAY_SIZE/n && i+k == -1 && rank%n!=0 && rank>n)
                  value = r_right_top;

                // right-bottom
                else if( j+m == ARRAY_SIZE/n && i+k == ARRAY_SIZE/n && rank%n!=0 
&& rank<=N-n)
                  value = r_right_down;

                // left bound
                else if(j+m == -1 && rank%n!=1)
                  value = r_left[i+k];

                // right bound
                else if(j+m == ARRAY_SIZE/n && rank%n!=0)
                  value = r_right[i+k];

                // upper bound
                else if(i+k == -1 && rank>n)
                      value = r_upper[j+m];

                //lower bound
                else if(i+k == ARRAY_SIZE/n && rank<= N-n )
                      value = r_lower[j+m];

                else
                  value = Z[i+k][j+m];

                sum+=value;

            }
          }
```

This code checks all the 8 critical cases and calculates the sum. After calculation of the sum, then it is time to determine whether flip or not. To determine this situation we have a formula.

```
  double delta_E = ((-2*GAMMA*X[i][j]*Z[i][j]) + (-2*BETA*Z[i][j]*sum));
```

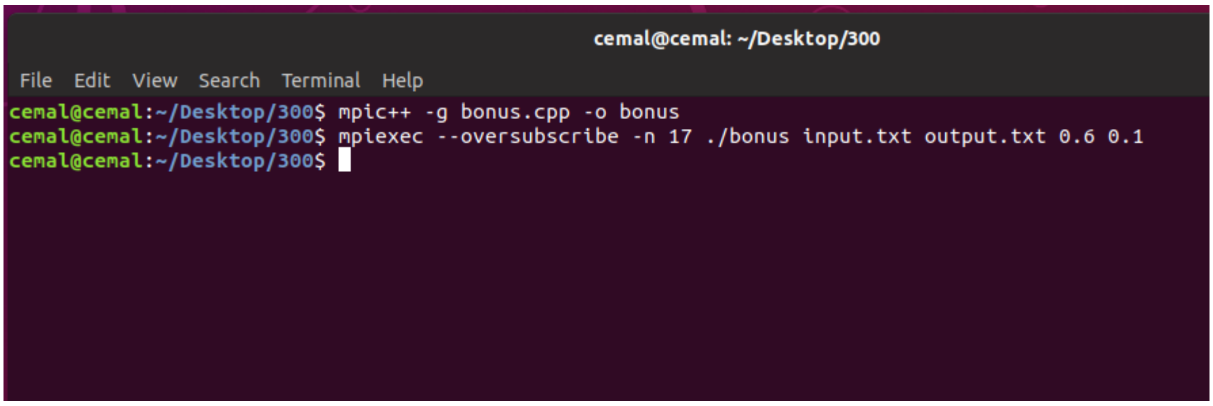
Then we generate a number between 0 and 1 and compare its log with delta_E. If delta_E is greater than the others, then we flip that index.

```
  double random = ((double) rand() / (double)RAND_MAX);
  if(delta_E > log(random))
       Z[i][j] = -Z[i][j];
```

When iteration is completed, I send the final status of the sub array (Z) to the master processor.

Examples

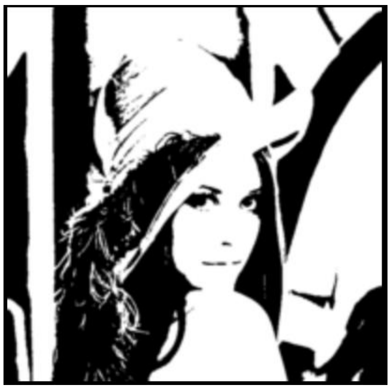
Run the program as in Fıgure 1.


cemal@cemal: ~/Desktop/300
File Edit View Search Terminal Help

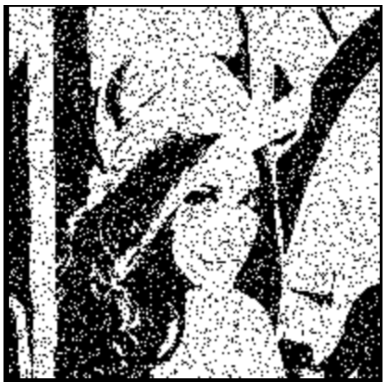
```
cemal@cemal:~/Desktop/300$ mpic++ -g bonus.cpp -o bonus
cemal@cemal:~/Desktop/300$ mpiexec --oversubscribe -n 17 ./bonus input.txt output.txt 0.6 0.1
cemal@cemal:~/Desktop/300$
```

Fıgure 1.

Example 1:

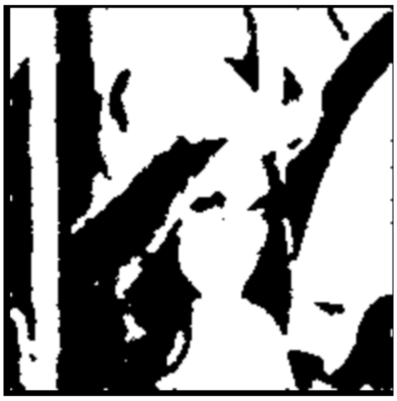


1. Original Image



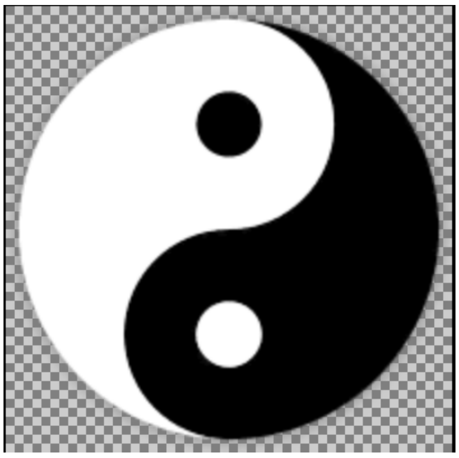
2. Noisy Image (input)

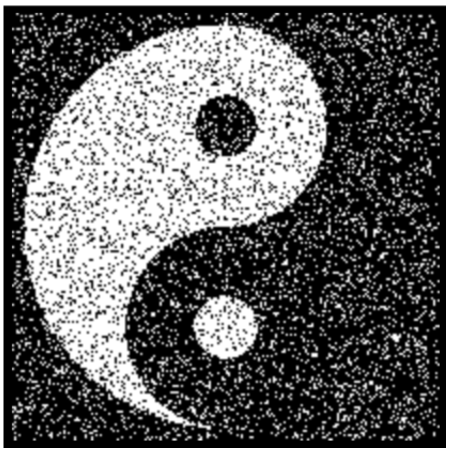
After program runs by takin noisy image as input the output is:



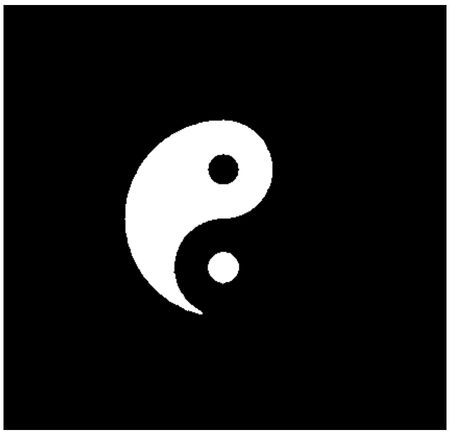
2. Output

Example 2:


1. Original Image


2. Noisy Image (input)

After program runs by taking noisy image as input the output is:


2. Output

Conclusion

In conclusion, this project was a great opportunity to get an idea about multiprocessing system and MPI library. I experienced how to use MPI library , send and receive functions with their parameters.