CMPE 300

ANALYSIS OF ALGORITHMS

MPI
Programming Project

BATURALP YÖRÜK
2015400036

26/12/2018

INTRODUCTION

In this project, our aim was to denoise a noised image to make it look like its original image. To do this, I applied Metropolis-Hastings algorithm which is:

1. Choose a random pixel from the image X

2. Calculate an acceptance probability of ipping the pixel or not.

3. Flip this pixel with the probability of the acceptance probability that is calculated in the second step.

4. Repeat this process untill it converges.

Also to apply this method, I used parallel programming to finish it in an acceptable amount of time. To make the algorithm work parallel, I partition the rows into processor numbers and send them to the slave processors via master processor.

PROGRAM INTERFACE

From terminal, come to the directory which contains the “main.cpp” file. After that, put the 200x200 sized .txt file to the same directory as “main.cpp” file. Then, compile the project with the following command: “mpic++ -g main.cpp -o outmain”. To run it, write the following command: “mpirun -n ‘some integer N that N-1 can divide 200’ ./outmain someinputname.txt someoutputname.txt "beta" "pi"”. After that an output file, named someoutputname.txt will be created. It is the output we wanted. To test whether it denoised the image or not, run the output file with the python code and here you are.

INPUT and OUTPUT

Give a .txt file which has 200*200 entry in it. That entry should only contain “-1” or “1”.

You will get an .txt file as an output. It is also going to contain only “-1” and “1”s in it.

PROGRAM STRUCTURE

The program only contains the main. At the beginning of the main, I first read the beta and pi inputs from terminal, after that, I initialized the MPI environment so that I can run the code in parallel.

Then we get into the master processor’s part. At there, we first read the input file which was given from command line. Then,

partition and send it to the slave processors. At the end of the master processor, we get back the changed input from slave processsors and write it into the output file.

Thirdly, we are getting into the slave processors' part. Here, we first take the input from master processor, after that we do some iterations with that input to denoise it. While doing that sometimes we need the row which is in the neighbour's array. We receive it from that processor and keep progressing. After denoising it with some techniques, we send them back to the master processor, so that it can write the output to the output file.

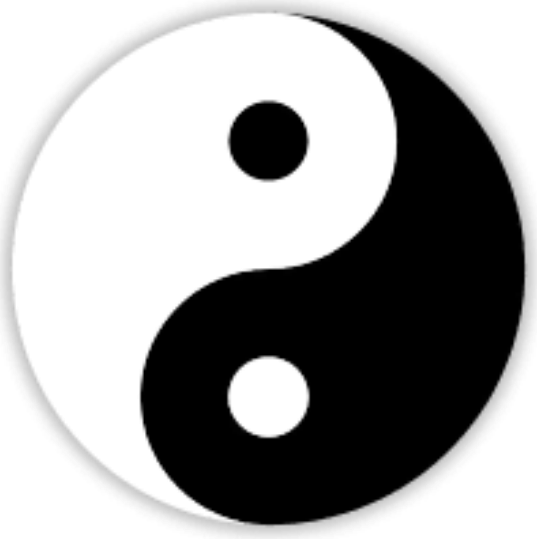
EXAMPLES

Original Image

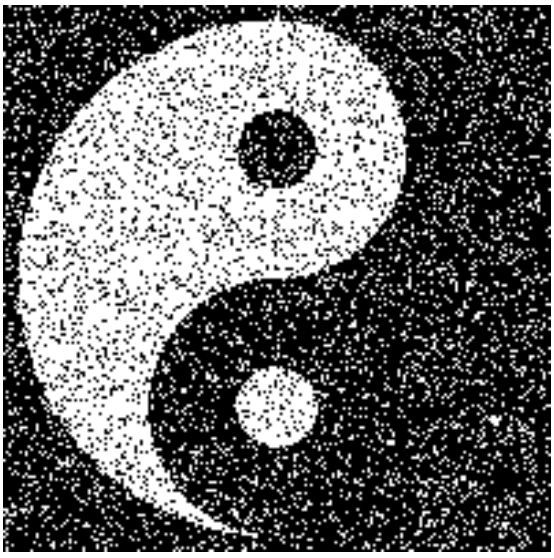
Noised Image

Denoised Image

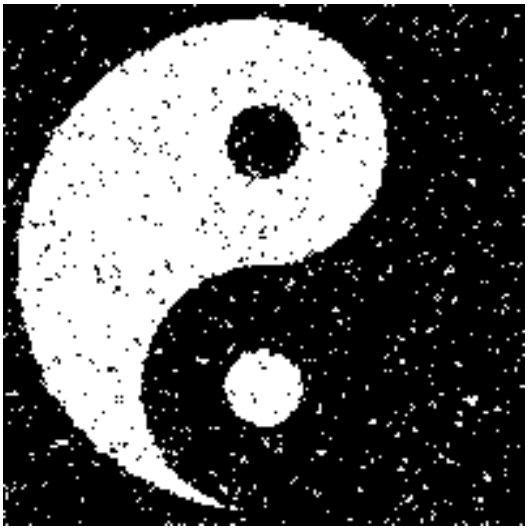
Original Image



Noised Image



Denoised Image



IMPROVEMENTS and EXTENSIONS

This program could be more efficient and it could be more naively implemented.

DIFFICULTIES ENCOUNTERED

I encountered some difficulties while implementing this project because it was my first time with MPI environment, thus I was not into it. Those difficulties was mostly about sending and receiving properly from the processors, shortly ensuring the communication between processors.

CONCLUSION

As a conclusion, I implemented my first MPI project and it succesfully denoised the images which were given to us.

APPENDICES

```
/*
Student Name: Baturalp Yörük
Student Number: 2015400036
Compile Status: Compiling
Program Status: Working
Notes: I have completed the project in Windows but with Ubuntu bash.
Compile it with mpic++ -g main.cpp -o outmain
Then run it with mpirun -n 5 ./outmain someinputname.txt someoutputname.txt "beta" "pi"
*/



#include <iostream>
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <fstream>
#include <cstdlib>
#include <ctime>
#include <cmath>
#include <algorithm>
using namespace std;

int main(int argc, char* argv2[], char** argv) {

  double beta = atof(argv2[3]);
  double pi = atof(argv2[4]);
  double gamma = 0.5 * log((1-pi)/pi);
```

```
// Initialize the MPI environment
MPI_Init(NULL, NULL);
// Find out rank, size
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;
int i, j;

//to make random function more efficient
srand((unsigned)time(NULL));

//if the number of arguments given from command line is not equal to five, give an error message to the user.
if (argc != 5) {
    cout << "Run the code with the [input_file] [output_file] \"beta\" \"pi\"" << endl;
    return 1;
  }

//the master process
if (world_rank == 0) {

        //opening the input file
        ifstream infile(argv2[1]);

  //int arr[200][200];
  //Dynamically allocating the memory.
  int** arr = NULL;
```

```
arr = (int **)malloc(sizeof(int *) * 200);
for(i = 0 ; i < 200 ; i++){
  arr[i] = (int *)malloc(sizeof(int) * 200);
}

//reading the input file and assigning it to "arr" array.
        for(int row = 0; row < 200; row++){
    for(int column = 0; column < 200; column++){
        infile >> arr[row][column];
    }
}

//Sending the corresponding N rows of input to corresponding slave processors.
for(i = 1 ; i <= N ; i++)
  for(j = 0 ; j < (200/N); j++)
    MPI_Send(arr[(200/N)*(i-1)+j], 200, MPI_INT, i, j, MPI_COMM_WORLD);

//Receiving the denoised input from the slave processors.
for(i = 1 ; i <= N ; i++)
  for(j = 0 ; j < (200/N); j++)
    MPI_Recv(arr[(200/N)*(i-1)+j], 200, MPI_INT, i, j, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

//Opening the output file.
ofstream out(argv2[2]);

//Writing into the output file.
for(int m = 0 ; m < 200 ; m++){
  for(int s = 0 ; s < 200; s++){
                out << arr[m][s] << " " ;
  }
```

```
    out << endl;
        }
        //closing the output file
        out.close();

        //end of the master process
 }

 //Slave processors' part
 else{

        //allocating the memory dinamically for the input which will be received from master process.
        int** subarr = NULL;
   subarr = (int **)malloc(sizeof(int *) * (200/N));
   for(i = 0 ; i < 200 ; i++){
     subarr[i] = (int *)malloc(sizeof(int) * 200);
   }

   //Receiving the corresponding rows from the master process.
   for(i = 0 ; i < 200/N ; i++)
     MPI_Recv(subarr[i], 200, MPI_INT, 0, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        //allocating memory for the row which will be taken from upper process(From lower ranked slave process).
   int* upNeighbour = NULL;
   upNeighbour = (int *)malloc(sizeof(int) * 200);

        //allocating memory for the row which will be taken from down process(From higher ranked slave process).
   int* downNeighbour = NULL;
```

```
downNeighbour = (int *)malloc(sizeof(int) * 200);


        //allocating memory for the copy array of the current processor's array.
int** changedSubarr = NULL;
changedSubarr = (int **)malloc(sizeof(int *) * (200/N));
for(i = 0 ; i < 200 ; i++){
  changedSubarr[i] = (int *)malloc(sizeof(int) * 200);
}


//copying the array to the array which is named "changedSubarr"
for(int m = 0 ; m < 200/N ; m++){
  for(int s = 0 ; s < 200; s++){
                changedSubarr[m][s] = subarr[m][s];
  }
        }


//500000 iteration for denoising the image enough
for(int z=0; z<500000; z++){


        //sending the last row to the down slave.
        for(int u=1; u<=N-1; u++){
                if(world_rank<=N-1) //to prevent last slave to try to send the last row to a slave which does not exists.
                        MPI_Send(subarr[(200/N)-1], 200, MPI_INT, world_rank+1, world_rank, MPI_COMM_WORLD);
        }


        //receiving the last row of the upper slave.
        for(int a=2; a<=N; a++){
                if(world_rank>=2) //to prevent first slave to try to get the upper row which does not exists.
```

```
                MPI_Recv(upNeighbour, 200, MPI_INT, world_rank-1, world_rank-1,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }

        //sending the first row to the upper slave.
        for(int t=2; t<=N; t++){
                if(world_rank>=2) //to prevent first slave to try to send the first row which
does not exists.
                MPI_Send(subarr[0], 200, MPI_INT, world_rank-1, world_rank-1,
MPI_COMM_WORLD);
        }

        //receiving the first row of the down slave.
        for(int b=1; b<=N-1; b++){
                if(world_rank<=N-1) //to prevent last slave to try to get the bottom row which
does not exists.
                MPI_Recv(downNeighbour, 200, MPI_INT, world_rank+1,
world_rank, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }

        //this is created for summing the 8 neighbour indexes
        int sum = 0;
        //this is the random column number
        int xrand = 0 + random() % 200;
        //this is the random row number
        int yrand = 0 + random() % (200/N);

        //these for loops are standing for summing the neighbour indexes of the randomly
chosen index.
        for (int w = -1; w < 2; w++){
        for (int q = -1; q < 2; q++){
```

```
            //this if corresponds to the current index and we do not add it to the sum
variable.
            //Also it prevents the attempt to the left of the 0th column, which does not
exists.
            if((w==0 && q==0) || w+xrand < 0){
                  sum = sum + 0;
            }

            else{

                  //if the chosen index is in the current processor.
                  if(w+xrand < 200 && q+yrand < (200/N) && q+yrand >=0){
                        sum = sum + subarr[yrand+q][xrand+w];
                  }
                  //if the chosen index is in the down processor.
                  else if(world_rank!=N && q+yrand == ((200/N)+1))
                        sum = sum + downNeighbour[xrand];
                  //if the chosen index is in the upper processor.
                  else if (world_rank!=1 && q+yrand == -1)
                        sum = sum + upNeighbour[xrand];
                  //in case of anormal index, we do not add to the sum.
                  else
                        sum = sum + 0;
            }

      }
      } // end of the sum calculations


      //to calculate the acceptance possibility.
      double delta_E;
```

```
        delta_E = -2*gamma*(subarr[yrand][xrand])*(changedSubarr[yrand][xrand])-
2*beta*(changedSubarr[yrand][xrand])*sum;

        //if delta_E is larger than some random double
        if (log((double)rand()/RAND_MAX)<delta_E){
                //flip that index
                changedSubarr[yrand][xrand] = -changedSubarr[yrand][xrand];
        }

   } //end of the iteration loop.

        //sending iterated versions of the input back to the master process
        for(j = 0 ; j < (200/N); j++)
        MPI_Send(changedSubarr[j], 200, MPI_INT, 0, j, MPI_COMM_WORLD);

  } // end of the slave process

  //finalize the MPI
  MPI_Finalize();
  return 0;

}
```