BOĞAZİÇİ UNIVERSITY

CMPE 300

ANALYSIS OF ALGORITHMS

PARALLEL PROGRAMMING WITH C++ USING MPI LIBRARY

PROGRAMMING PROJECT

METE HAN KURT

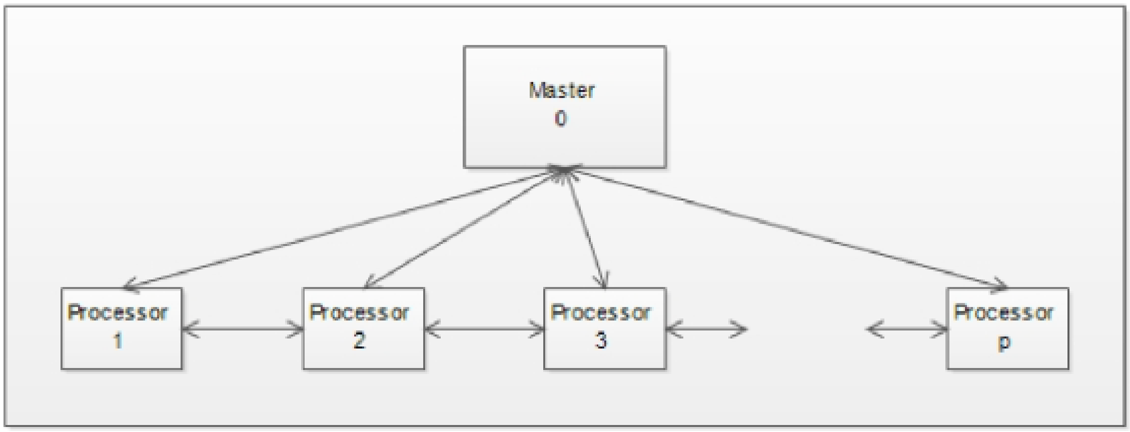
2016400339

Submitted to Burak SUYUNU

December 26, 2018

1. Introduction

This project aims to turn a noisy picture into a denoisy picture as good as possible with the help of processors using MPI Parallel Programming. The problem is solved using Open MPI environment in C++ language. Project follows an algorithm explained with a figure below.



There is one master process and (size-1) number of slave processors.

Assumptions and requirements regarding the project:
- Input file will given noisy.
- It is text file containing 200 rows and columns with -1 and 1s.
- The number of processors should be given in form (200/n-1) 6,11,21,26 etc.
- Input and output handled by master processor.
- In case of empty file, program flow will be interrupted after input reading.
- Number of words given to each processor may vary depending on input size and number of processors.

To be able to see the result of the projects, a Python script is included in the Project (text_to_image.py). With this script, we can accurately visualize the output.txt file and it results 200*200 image, which is a denoisy version of the noisy image. The beta and pi values to calculate exponential distrubition are given by the user, so we can see that optimal values gives us the closest image to our output and a random values of pi and beta can give a more complicated image with lots of pixels.

2. Program Interface

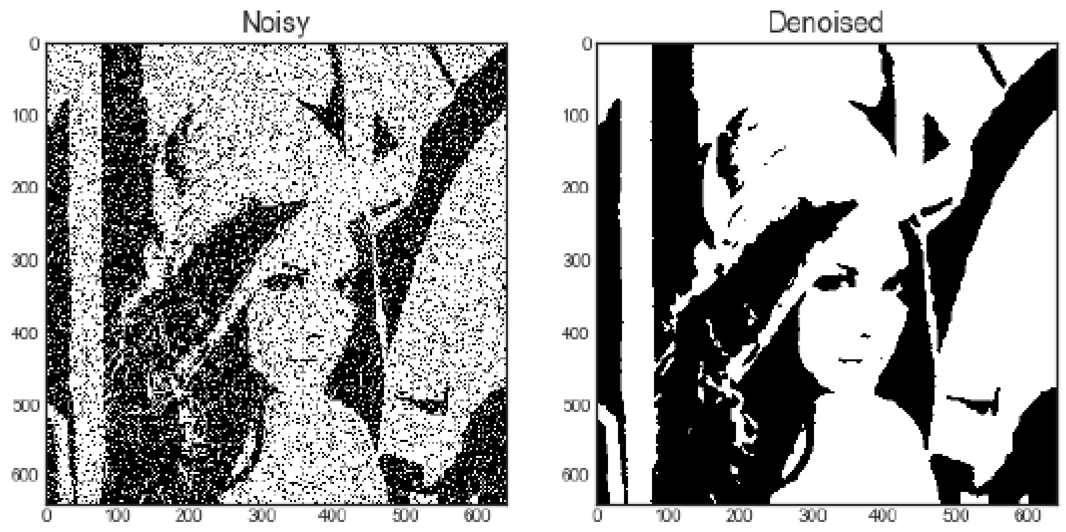
This project is implemented in C++, using MPI libraries. To be able to build and run this projects, user should have these installed. Also, the visualization script is written in Python, therefore the user should have Python installed to be able to see the results. ***Project.cpp*** file which is wanted to be compiled and the input files should be in the same folder. To build the project, user should open a terminal in the folder which contains the project file, and enter the command ***<mpic++ project.cpp -o ready>.*** This command takes in the ***project.cpp*** file, compiles it and writes the executable in another file which is named *ready.* User can change the executable's name however he wants. To run the project, user should enter the following command to the terminal which is opened at the same folder:

mpiexec -n <Processors> <executable> <input> <output> <beta&pi values>

3. Program Execution

As stated above, the program can be run by *mpiexec -n <Processors> <executable> <input> <output> <beta&pi values>* command. In this command, *<Processors>* is an integer, the total number of processors, n-1 slave processors and 1 master processor. Since this is a parallel programming project that gives every slave processor equal jobs, user must choose the processor number so that the image size 200 should be divisible by the slave processor count (n-1). *<executable>* is the name of the executable file that user created in the command above. *<input>* and *<output>* are the names of the input and output files. Input file should contain 200 lines with each line containing 200 integers 1 or -1. The *<beta&pi values>* are random values, which determines how to convert the image. An example command would be: *<mpiexec -n 21 ready(executable) input.txt output.txt 1 0.1>* which runs the executable with 1 master and 20 slave processors, input is taken from *input.txt* and the output is written to *output.txt* with the values of 1 and 0.1. To be able to see the results of the processes, user should use the *<python text_to_image.py output.txt m.img>* command to run the visualize of the output file. User should manually change the python script if he wants to convert an image file into a text file.

4. Input and Output



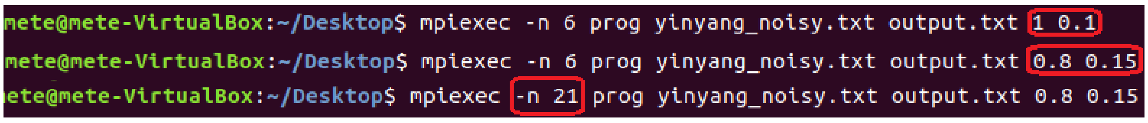
The first image is given as input and when we compile the program we get the second image as output. Input file should be at the same directory as the source code. If not, the address of the input file should be specified/changed at the code. The name of the generated output file is output.txt and user can find output file in the same directory with program file. After successfully compile & run you can see how similar it is to the original.

5. Program Structure

The application is developed with C++ with MPI.
The program first checks the input values. If your input values are different from the desired value, it returns an error. If the values are correct, the master processor assigns the values in the input.txt file to a two-dimensional array. Then calculate the exponential distrubition according to the values around a randomly selected point. Slave processors change the values in rows and columns in parallel. The important point in this correction is the pi and beta values entered by the user. After editing, the values are sent to the main processor and written to the output.txt file.
To make the structure easier to understand, the program is explained with the comments line by line. The program itself is given in the appendices, you can see the related methods from there.

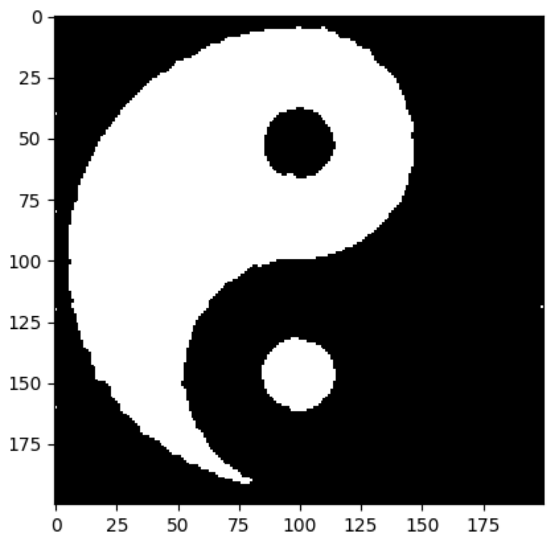
6. Examples

You can try running the program with different number of processors and values of beta & pi . The values of beta & pi are 1 & 0.1 respectively, and it gives the best-looking image result for yinyang_noisy.txt, but you can also try with 0.8 & 0.15 beta, pi values or any other numbers of processors like this:



```
mete@mete-VirtualBox:~/Desktop$ mpiexec -n 6 prog yinyang_noisy.txt output.txt 1 0.1
mete@mete-VirtualBox:~/Desktop$ mpiexec -n 6 prog yinyang_noisy.txt output.txt 0.8 0.15
ete@mete-VirtualBox:~/Desktop$ mpiexec -n 21 prog yinyang_noisy.txt output.txt 0.8 0.15
```

If we type the first line from the above examples in terminal, the program gives us output below. We can also get an output by giving the value in the second row, but the result for these pi&beta values is deviated from the original output. Finally, the number of processors can also be changed, which does not affect the accuracy of the output, but increases the compile time.



7. Improvements and Extensions

The program might be written in more efficient way with respect to time and space complexity. But since the aim was to understand the parallel programming approach, major focus was on learning parallel programming.

Also in this version program does not run if the number of processors matching 200 lines is not given. In that case I think the program should automatically compile for each entered number of processors by making small additions to the values it receives from the input file.

8. Difficulties Encountered

We all learned what MPI is especially in Operating Systems(CMPE322) course. But it was my first time implementing this interface in a program. Also, it took my a while to understand how Open MPI works. I spent quite some time synchronizing the communication between slaves and master , in my opinion, this process is the most difficult and critical part of the project. Because you need to synchronize parallel working process with each other. Otherwise, even if they all work correctly at the same time, the connection problem between processors can be seen clearly when you create the actual image. Additionally, I have added barrier between communication. An absolute difficulty was debugging the program. There are lots of messages being passed both master-to slave and slave-to-slave, and it was hard to keep track of every one of them. This was a very good project to practice parallel programming in my opinion. I was a little bit unsure about the memory leaks. Since we use memory allocation a lot in this project, it could be a good place to learn how to detect and avoid memory leaks. I also learned a library called Boost.MPI to solve this synchronization problem. But I couldn't use this library because I finished most of the project. But if I come across such a project in the future, I think I can work more efficiently by using a different library.

9. Conclusion

This project was a good practice to understand and learn the basic functions regarding parallel programming approach. It’s not only a parallel application, but a concurrently working. Once the concepts are understood, actually it is not that complex project. Indeed it was a hard project to start but also I think it is beneficial to learn the complicated functions. It was a good opportunity to have hands on experience to familiarize with the MPI programming interface, but one needs to practice more to write better programs I finished all tasks that are required successfully. The program works as it should.

10. References

I've watched several youtube videos to comprehend and compile the MPI library, and I've used the codes given by the assistant. I also got help from https://stackoverflow.com/ for compiling error, installing a new library, missing packages, python updates and many other problems. In addition to these, the following links were very helpful for me.

http://mpitutorial.com/tutorials/
https://www.open-mpi.org/
https://www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/node114.html

11. Appendices

```
/*
Student Name: METE HAN KURT
Student Number: 2016400339
Compile Status: Compiling
Program Status: Working
Notes: The program first checks the input values. If your input values
are different from the desired value, it returns an error.
       If the values are correct, the master processor assigns the
values in the input.txt file to a two-dimensional array.
       Then calculate the exponential distrubition according to the
values around a randomly selected pixel.
       Slave processors change the values in rows and columns in
parallel.
       The important point in this correction is the beta and pi values
entered by the user.
       After editing, the values are sent to the master processor and
written to the output.txt file.
*/
#include <iostream>
#include <fstream>
#include <math.h>
#include <cstring>
#include <mpi.h>
using namespace std;

// 200x200 is fixed size, so they are defined as fixed.
const int array_rows = 200;
const int array_columns = 200;

// The method which creates 2D array dynamically is called.
int** matrix(int, int);
```

```
// Main function
int main(int argc, char *argv[])
{
    if(argc < 5) //  Returns an error if at least one of the required
values is missing.
    {
        cout << "Invalid args\n";
        return 0;
    }
    int data_order, data_size, slave_count, row_count, i, j, k, val,
ii, jj, total, count = 0, T, arc, ran; //  Define the variables we will
use.
    int** temp_array, **temp_array2, receive_data1[200],
receive_data2[200]; // Temporary arrays to keep the values to be copied
are defined.

    MPI_Request req;

    string input_data = argv[1], output_data = argv[2]; // Input and
output values are initialized.

    double beta_value = atof(argv[3]), pi = atof(argv[4]), gamma_value
= 0.5 * (log10((1 - pi) / pi)); // Variables to be used for gamma
detection.

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &data_order); // Rank of processors
    MPI_Comm_size(MPI_COMM_WORLD, &data_size); // Total number of
processors(n)
    slave_count = data_size - 1; //   Number of slave processors
    row_count = 200 / slave_count; // There are 200 rows and number of
rows are shared to slave processors equally.


    if(data_order == 0) // For the master processor..
    {

        ifstream reader;
        ofstream writer;

        reader.open(input_data); //  Opened file to be read
        writer.open(output_data); // Opened file to be written

        temp_array = matrix(array_rows,array_columns); //  temp_array
was assigned a two-dimensional location of 200x200

        if(reader)
        {   // We read from the file and assigned the values to
temp_array.
            for(i = 0; i < array_rows; i++) // For each row..
            {
                for(j = 0; j < array_columns; j++) // For each coulmn..
                {
                    if(reader >> val)
                    {
                        temp_array[i][j] = val; // The value read is
thrown into the matrix of the temp_array.
                    }
                }
            }
        }

        for(i = 1; i < data_size; i++)// Sending to slave processors
         {
            MPI_Send(&(temp_array[(i - 1) * row_count][0]), row_count *
array_rows, MPI_INT, i, 0, MPI_COMM_WORLD);
        }

        temp_array2 = matrix(array_rows,array_columns);// temp_array2
was assigned a two-dimensional location of 200x200

        for(i = 1; i < data_size; i++) // Receiving to slave processors
           {
            MPI_Recv(&(temp_array2[(i - 1) * row_count][0]), row_count
* array_rows, MPI_INT, i, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

            }


        for(i = 0; i < array_rows; i++)  // The values assigned above
are written to the file.
        {
            for(j = 0; j < array_columns; j++)
            {
                writer << temp_array2[i][j] << " " ; // Writing the
assigned values in temp_array2
            }
            writer << "\n"; // If line end is reached, move to next
line.
        }

        reader.close();
        writer.close();
    } else { //  For slave processors..
        T = 250000; // Number of Iteration

         // Two two-dimensional dynamic arrays are created.
        temp_array2 = matrix(row_count,array_columns);
        temp_array = matrix(row_count,array_columns);

        MPI_Recv(&(temp_array[0][0]), row_count * array_rows, MPI_INT,
0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);



        for(i = 0; i < row_count; i++)// We just copy temp_array to
temp_array2.
        {
            for(j = 0; j < array_columns; j++)
            {
                temp_array2[i][j] = temp_array[i][j];
            }
        }

        for(i= 0; i < T; i++){
            ii = rand() % row_count, jj = rand() % (array_columns) ,
total = 0; // Assigning a random row and column value to ii and jj.
            if(ii == 0 || ii == row_count - 1) //For the top rows or
bottom rows
            {
                switch(data_order)// Processes to be performed in
processor order.
         {

                case 1: //  For the 1st slave processor..

                    if(ii != 0) // For the last line of the first slave
processor.
                    {
                        for(j = ii - 1; j < ii + 1; j++)
                        {
                            for(k = jj - 1; k < jj + 2; k++)
                            {
                                total += temp_array2[j][k];
                            }
                        }
                        total += receive_data2[jj - 1] +
receive_data2[jj] + receive_data2[jj + 1] - temp_array2[ii][jj];
                    }
                    else if(ii==0)  // For the first line of the first
slave processor.
                    {
                        for(j = ii ; j < ii + 2; j++)
                        {
                            for(k = jj - 1; k < jj + 2; k++)
                            {
                                total += temp_array2[j][k];
                            }
                        }
                        total=total-temp_array2[ii][jj];
                    }
                   break;

               default:
                  if (data_order == data_size-1)    //  For the last
slave processor..
                  {

                   if(ii!= row_count-1) //  For the first line of the
last slave processor.
                    {
                        for(j = ii; j < ii + 2; j++)
                        {
                            for(k = jj - 1 ; k < jj + 2; k++) //
Controls from the previous to the next
                            {
                                total += temp_array2[j][k];
                            }
                        }
                        total += receive_data1[jj - 1] +
receive_data1[jj] + receive_data1[jj + 1] - temp_array2[ii][jj];
                    }
                    else if(ii==row_count-1)  // For the last line of
the last slave processor.
                    {
                        for(j = ii-1 ; j < ii + 1; j++)
                        {
                            for(k = jj - 1; k < jj + 2; k++) //
Controls from the previous to the next
                            {
                                total += temp_array2[j][k];
                            }
                        }
                        total=total-temp_array2[ii][jj];
                    }
             }
             else //  For the slave processors in middle..
             {

                    if(ii == 0)// For the first line of a slave
processor in the middle.
                    {
                        for(j = ii; j < ii + 2; j++)
                        {
                            for(k = jj - 1; k < jj + 2; k++) //
Controls from the previous to the next
                            {
                                total += temp_array2[j][k];
                            }
                        }
                        total += receive_data1[jj - 1] +
receive_data1[jj] + receive_data1[jj + 1] - temp_array2[ii][jj];
                    }
                    else // For the last line of a slave processor in
the middle.
                    {
                        for(j = ii - 1; j < ii + 1; j++)
                        {
                            for(k = jj - 1; k < jj + 2; k++) //
Controls from the previous to the next
                            {
                                total += temp_array2[j][k];
                            }
                        }
                        total += receive_data2[jj - 1] +
receive_data2[jj] + receive_data2[jj + 1] - temp_array2[ii][jj];
                    }
             }

        }


            }
            else //For neither the first row nor the last row of any
slave processor.
            {
                for(j = ii - 1; j < ii + 2; j++)
                {
                    for(k = jj - 1; k < jj + 2; k++)
                    {
                        total += temp_array2[j][k];
                    }
                }
                total = total - temp_array2[ii][jj];
            }
            // Calculates the exponential distribution according to the
values around one point
            arc = (-2) * gamma_value * temp_array[ii][jj] *
temp_array2[ii][jj] + (-2) * beta_value * temp_array2[ii][jj] * total;
            ran = rand(); //  A random number is assigned to ran

            if(log(ran) < exp(arc))
```

```
            {
                temp_array2[ii][jj] = -1 * temp_array2[ii][jj]; // The
pixel is flipped from 1 to -1
            }


            switch(data_order) //  Processes to be performed in
processor order.
            {
            case 1:   // If it is 1st slave..

                MPI_Send((temp_array2[row_count - 1]), 1 * array_rows,
MPI_INT, 2, 0, MPI_COMM_WORLD);    //Last row of P1 was sent by
temp_array2
                MPI_Recv(receive_data2, array_rows, MPI_INT, 2, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);   //Last row of P1 was received by
receive_data2
                break;


            default:
             if (data_order == data_size-1)  // If it is the last
slave..
             {

                MPI_Send((temp_array2[0]), 1 * array_rows, MPI_INT,
data_size - 2, 0, MPI_COMM_WORLD);    //First row of P(n-1) was sent by
temp_array2
                MPI_Recv(receive_data1, array_rows, MPI_INT, data_size
- 2, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE); //First row of P1(n-1) was
received by receive_data1
             }
             else //  If they are middle slaves..
             {

                MPI_Send((temp_array2[0]), 1 * array_rows, MPI_INT,
data_order - 1, 0, MPI_COMM_WORLD);     //First row of P(n) was sent by
temp_array2
                MPI_Recv(receive_data1, array_rows, MPI_INT, data_order
- 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);  //First row of P(n) was
received by receive_data1
                MPI_Send((temp_array2[row_count - 1]), 1 * array_rows,
MPI_INT, data_order + 1, 0, MPI_COMM_WORLD);   //Last row of P(n) was
sent by temp_array2
                MPI_Recv(receive_data2, array_rows, MPI_INT,
data_order+1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);  //Last row of
P(n) was received by receive_data2
             }
         }


        }
            //  Slave processors are sent to master processor
        MPI_Send(&(temp_array2[0][0]), row_count * array_rows, MPI_INT,
0, 1, MPI_COMM_WORLD);
    }
    MPI_Finalize(); //Terminates the calling MPI process’s execution
environment.
}
```

```
//  Two-dimensional dynamic array generating method
int** matrix(int r, int c)
{
    int i;
    int* d = (int *)malloc(r * c * sizeof(int));
    int** new_map= (int **)malloc(r * sizeof(int *));
    for (i = 0; i < r; i++)
    {
        new_map[i] = &(d[c * i]);
    }

    return new_map;
}
```