CMPE 300

Programming Project Fall 2016

Gönül AYCI
2016800003
December 27, 2016

1 Technical Part

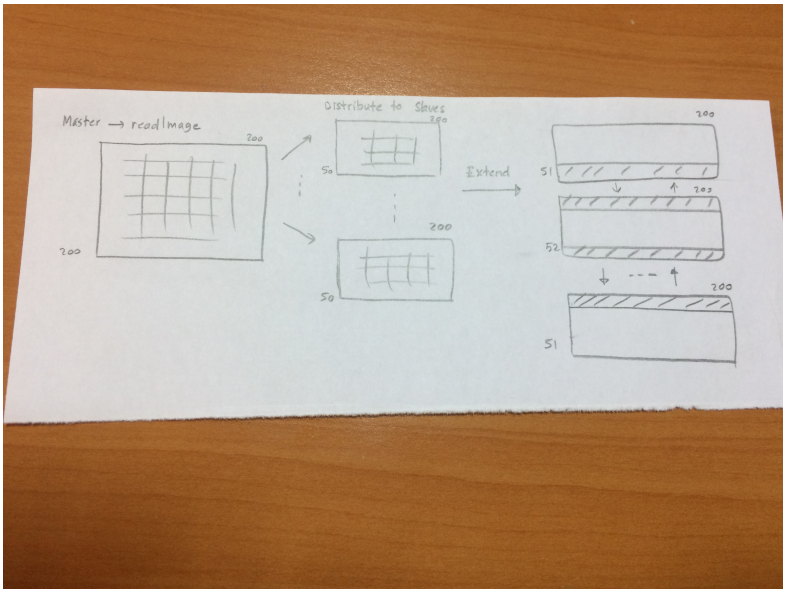
At the beginning of project, I need to install:

- Open MPI
- Python environment

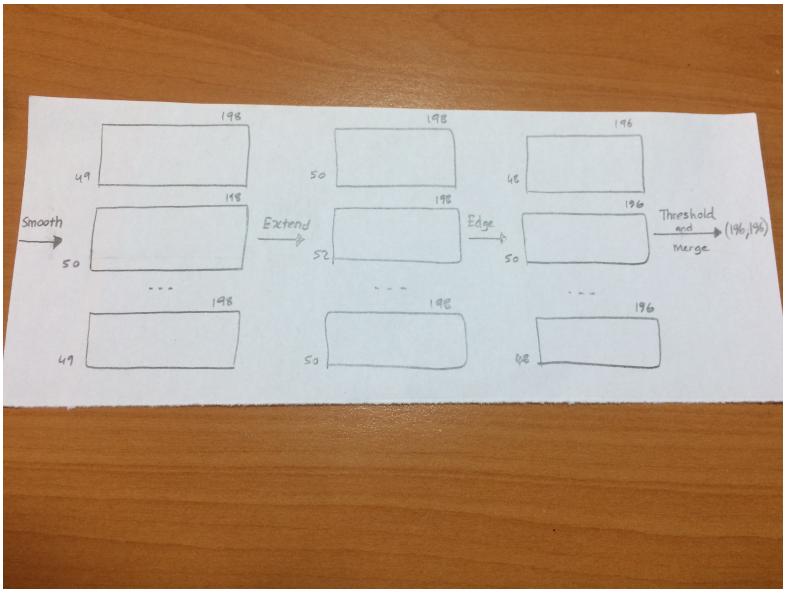
2 Structure

First of all, I divided my project into two parts. The first part of this project is **Smoothing and Line Detecting** and another one is **Parallel Algorithm for Smoothing and Line Detecting**. Parallel algorithm part is an important and a little confusing part of this project. Because, you need to memorize which process send a data and then which one get data. I draw a structure of this problem. I divided parallel algorithm part into seven parts which are:

- Distribute: 50 by 200 for each slave
- Extend: 51 by 200, 52 by 200 (*2), 51 by 200



- Smooth: 49 by 198, 50 by 198 (*2), 49 by 198
- Extend: 50 by 198, 52 by 198 (*2), 50 by 198
- Edge Kernel: 48 by 196, 50 by 196 (*2), 48 by 196
- Apply Threshold
- Merge: 196 by 196



3 IMPLEMENTATION

Implementation part is divided into three parts which are *MPI, Algorithm-C,* and *Functions* parts.

3.1 MPI

Firstly, we have an image data file which is 200 by 200. I distributed these data from *Master* to *Slaves.* In MPI programming, I have *rank* and *size. size* is the number of processes and *rank* points that only one process. I use MPI environment. Now, I want to explain using MPI issues.

- `MPI_Init(&argc, &argv)`: Initialize the MPI execution environment
- `MPI_Comm_rank(MPI_COMM_WORLD, &rank)`: Determines the rank of the calling process in the communicator
- `MPI_Comm_size(MPI_COMM_WORLD, &size)`: Determines the size of the group associated with a communicator
- `MPI_Send(&array[(i-1)*partSize+j][0],colnCount,MPI_INT,i,0,MPI_COMM_WORLD)`
- `MPI_Recv(&recArr[i][0],200,MPI_INT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE)`
- `MPI_Barrier(MPI_COMM_WORLD)`: Blocks until all processes in the communicator have reached this routine.
- `MPI_Finalize`: Terminates MPI execution environment

MPI_Init: This has two input parameters which are `argc` and `argv`. `argc` is a pointer to the number of arguments, and `argv` is a pointer to the argument vector.
The MPI standard does not say what a program can do before an `MPI_Init` or after an `MPI_Finalize`.
MPI_Comm_rank: This defines the rank of the calling processes in the communicator. `comm` means communicator.
The rank is different from the size.

- if rank == 0, Master process
- if rank == 1, First slave process
- if rank == size-1, Last slave process
- if rank != 0, Slave processes between first and last processes

MPI_Comm_size: Size is the total number of processes in the communicator. At the end of this report, I give an example to compile my project. I use `-n 5` which means there are 5 processes in the communicator.
MPI_Send: In this project, I need to use *MPI_Send* many times. Processes send data each

other using by MPI_Send. First of all, I need to distribute my input data to slave processes under favour of master process. So, our input data size is 200 by 200, the data can distributed into 4 slaves such as between (0,0)-(49,0)–>Slave1(by 200), (50,0)-(99,0)–>Slave2(by 200) and so on. It supports the communication between processes using with MPI_Recv. MPI_Send parameteres include destination rank number. As I mention before, in this project;
MPI_Recv: So, MPI_Recv is the inverse operation of MPI_Send. In this step, processes receive data from their neighbours.
I use Send and Receive in Distribution step, Extention step, Smoothing step.
MPI_Barrier(MPI_COMM_WORLD): It is also important for communication in parallel programming. This is beneficial for synchronization of processes. It can avoid any potential deadlocks. It gives us to guarantee that processes can share their data safely. The routine is not interrupt safe. Typically, this is due to the use of memory allocation routines such as `malloc`.
MPI_Finalize(): It is like an opposite of MPI_Init, this supplies the termination of MPI execution.

3.2 ALGORITHM-C

Related to the programming part, I use:

- `int **array = malloc(200 * sizeof(int *))`
- `free(array)`

These are C programming part. As we know that, malloc function is beneficial for memory allocation. I use it to create arrays such as given input data array (array), receice data array (recArr), smooth image array (outSmooth), edge image arrays (edgeImage-i), output result image array (result).
In additions to `malloc`, I should use `free`. I learn these C language two functions during this project. I need to use `free` function, because when I allocate the memory, then need to delete allocation space from memory.

3.3 FUNCTIONS

In the implementation part of this project, I totally use five functions because of simplicity which are:

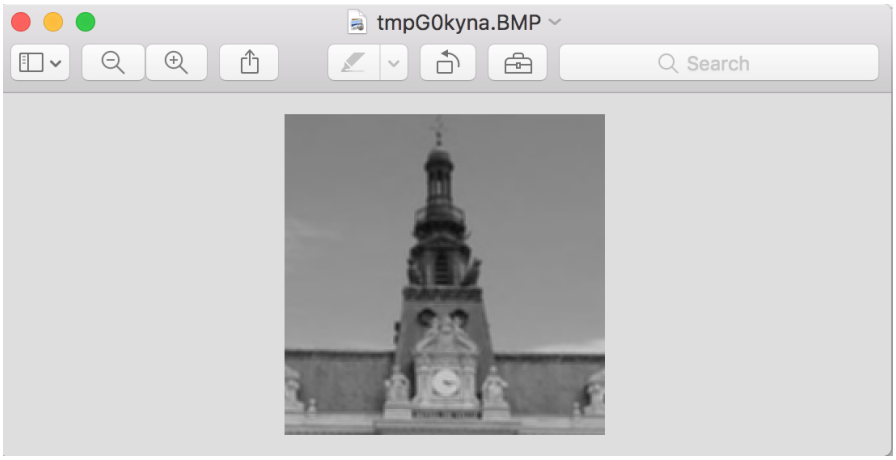
- readImage
- applySmooth
- applyKernel
- applyThreshold
- applyWrite

readImage: In this function, I read a text file with dynamically. That is, the size of array can not be defined at the beginning part.
applySmooth: The aim of this function is that to smooth the given image by using a 3 by 3 mean filter (which takes the average of 9 numbers). I convert my function Metehan Doyran's function after his public e-mail. The function do convolution steps of the 200 by 200 integer array with a 3 by 3 double array. This function gets four parameters which are input image, output image, number of input rows, and number of input columns. There is a reduction in here. When we have an integer input image with size N by N and apply this function, the output image array becomes (N-2) by (N-2).
applyKernel: In this function, I have five parameters which are input image array, output image array, filter matrix double array, the number of rows, and the number of columns. Note that, I do not mention about four filters for detecting lines in a given image.
applyThreshold: This function has eight parameters. Four of them four filters(edge1, edge2, edge3, and edge4), one of them is output, one of them is threshold value, and others are the number of rows, the number of columns. Information given in project description, all the values lesser than or equal to the threshold will be set to 0(black), and all the values greater than the threshold will be set to 255(white). So, I use an *if statement* which includes `edge1[i][j] > threshold || edge2[i][j] > threshold || edge3[i][j] > threshold || edge4[i][j] > threshold`
applyWrite: In this function, I write my results to a text file. This function has four parameters which are matrix for input image, result file name, the number of rows, and the number of columns.

4 RESULTS

At the end of this project, this is the time for showing results. First of all, I want to show smoothed image, so I call from terminal

- `mpicc smooth.c -o smooth`
- `mpiexec -n 5 smooth`
- `python visualize.py write.txt`



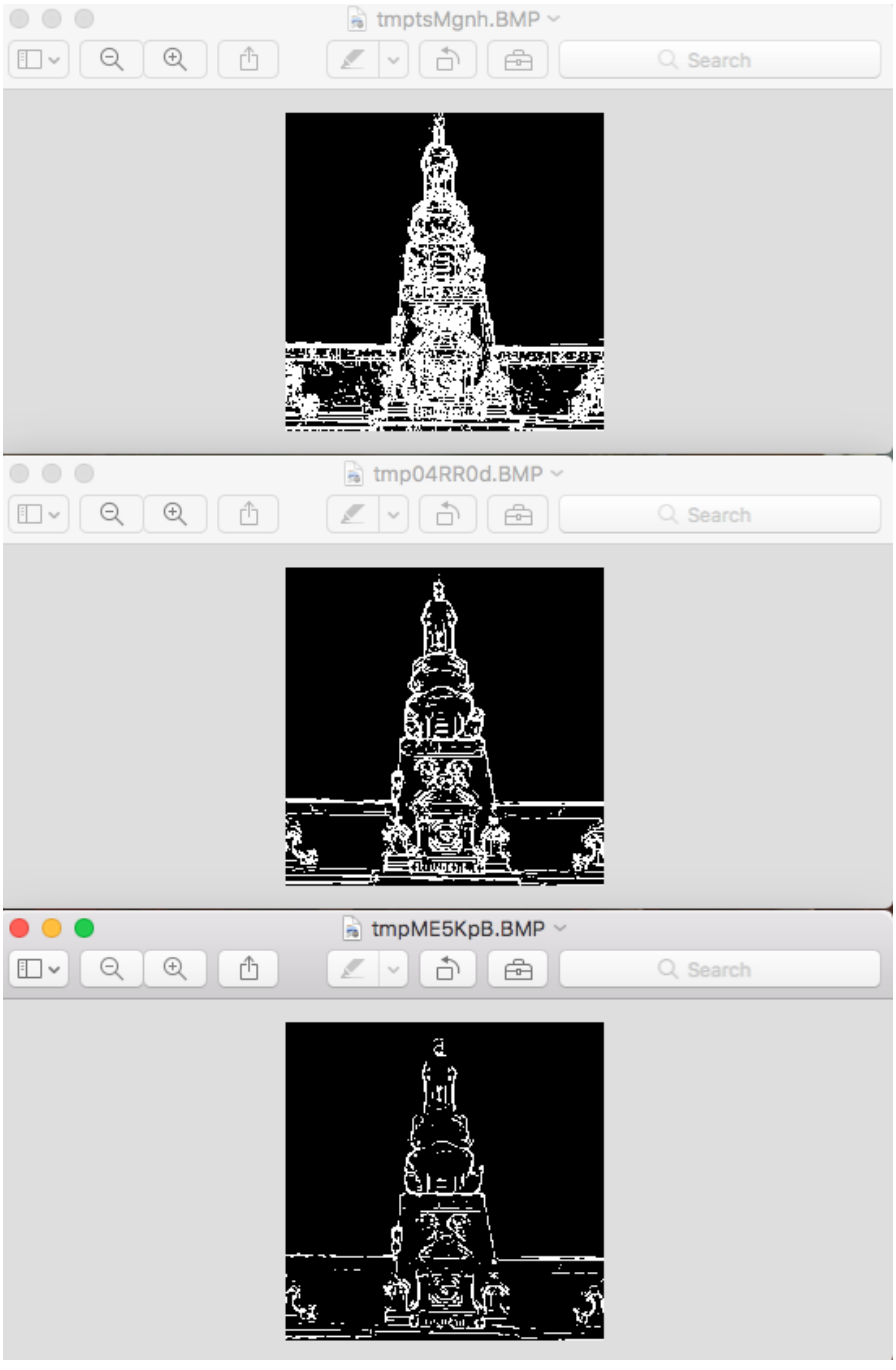
This is the first part of my project.

As we show that, my program is smoothing the given image successfully. We can observe the difference and similarities between two images (input.txt, and smooth.txt).

Then, in the second part of my MPI project, I need to show the result images which is applied some different thresholds. So, I need to compile my main.c file and call python script which is given from project descriptor.

- `mpicc main.c -o main`
- `mpiexec -n 5 main`
- `python visualize.py output10.txt`

First and second lines equal to `mpiexec -n <Processors> <executable> <input> <output>` from project description. As an example of compilation, I use 5 processors.
I apply three different thresholds (10, 25, 40), and visualize these outputs by using the python script as requested in description of project. The results for threshold 10, 25, 40, respectively are below:


tmptsMgnh.BMP
Search
tmp04RR0d.BMP
Search
tmpME5KpB.BMP
Search