Parallel Image Denoising

Introduction

In this project I implemented an image denoising program that runs on multiple number of processors at once, concurrently. The image is a black – white image and can be assumed as a binary 2D array. I assumed that the image

- Generated by the Ising model. (explained below)
- Image is noised means that some random pixels of the image is flipped to the other bit (from white to black or from black to white)

So my goal was finding which pixels were flipped and flip them back to the original bit. To calculate how likely the pixel are noise ones and needs to be flipped I imlemented a probablistic formula.

I also choose the pixels randomly, but in my implementation there are lots of iteration of this choosing and altering before conclude that the image denoised so we can assume that each pixel is choosen multiple times or at least once.

Ising Model

Ising model is actually a mathematical model of ferromagnetism in statistical mechanics. In Ising model the particles has 2 states, 1 and -1. In a very basic point of view to this model, his model shows how these particles interact with each other. This model says, if there are lots of -1 valued particles around a particle, this particle usually also valued as -1.

Program Interface

The program is implemented using C++ using MPI so to compile it the user will need to download the mpi library first. Then the code can be compiled from linux terminal in the directory of the C++ file using the

"mpic++ main.cpp -o main.o"

and can be executed using

"mpiexec -n N ./main.o inputfile.txt outputfile.txt beta pi"

and turned into image file by using the command

"python3 ./text_to_image.py result.txt output.png"

Program Execution

Argugments are as follows:

- inputfile.txt is the 2D array model of the noised image
- outputfile.txt is the 2D array model of the denoised image and can be translated into a .png file using the text_to_image.py program that has been shared to all students.
- Beta is a double that indicating how much the image is consistent with the ising model.
- Pi is a double indicating how much the image is noised(pi). The result will be generated in the same directory.

```
~/Desktop/cmpe300$ mpic++ main.cpp -o  main.o
~/Desktop/cmpe300$ mpiexec -n 3 ./main.o lena200_noisy.txt result.txt 0.4 0.15
~/Desktop/cmpe300$ python3 ./text_to_image.py result.txt output.png
```

The original image:

The noised image:

The denoised image:

Input And Output

The input should be given to the program via terminal arguments. The arguments should be as follows with the strict order:

-n N :
this argument indicates the number of processors that the program will be executed on ans since its not given with the same format with the other arguments its explained separately. N is the number of processors given to the program.

1. Inputfile.txt
2. Outputfile.txt
3. Beta
4. Pi

Inputfile.txt :
inputfile.txt is the 2D array file containst 1 for the white pixel and -1 for the black pixel. Its name should be passed to the program using arguments. The 2D array should be in size 200 x 200. A picture can be translated into such file using image_to_text.py

Outputfile.txt :
outputfile.txt will be generated by the program so there is no need to put such file into the directory. If there exist such file in the directory, it will be overwritten. But its name should be passed to the program.

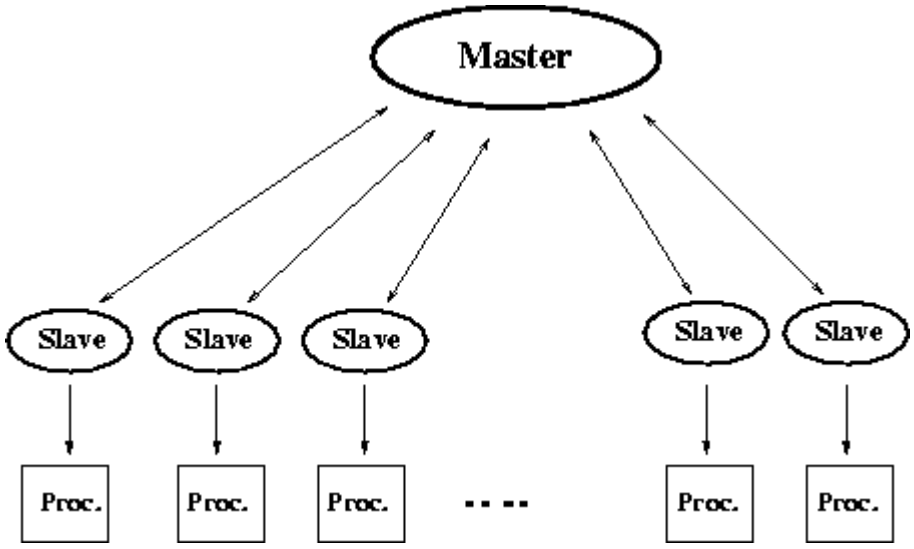
Beta and pi :
Actually these are parameters to the given input image to indicate its status. (if the image fits the ising model or in another words how much does it fits and the level of the noise in the image)

Program Structure

In this program the number of the processors is given as an argument to the program. 1 processor with the ID 0 is used as Master processor and the others are used as slave processors.

Master and Slave Approach

Master/slave is a model of communication where one processor has unidirectional control over one or more other processors.The controlling processor is called the master, the processors doing the actual work are called slaves. Usually, slaves are generated as necessary to solve the problem.



Program's code is seperated into 3 parts, main part, master processor, slave processor. And in the Slave processor there is another part called Send and receive part.

Main part

This part is executed by all processors (Master and the slave processors). Arguments are assigned to the variables, the variables that all or most of the processors should initialize are also initialized here.

Each processors rank is assigned in the main part of the program and MPI environment is initialized as well.

Master Processor

The file streams are initialized here since all reading and writing will be done my the master processor. The 2D array is read into the 1D array arr[]. This file is divided into pieces and one piece of the array sent to the one slave processor.

Then after the slave processors done their job on this pieces of the information, they send their data to the master processor and master processor writes this data to the output file.

For more detailed information please follow the comments in the c++ code corresponding to the master processor part from the program.

Slave processors

The arrays and variables that each slave processor needs during their processes are initialized here.

Also the denoising is done here. A slave processors picks a pixel randomly and looks at the neighbor pixels. If the neighbor pixels is in another processor, the information that received from that processor is used. The processors send their first and last rows of data with the other processors that might need this info (the processors that denoising the image part just above or under). After determining what these neighbors values are the acceptance probability for the pixel flip is calculated by using the formula :

$$\left(-2\gamma Z_{ij}^{(t)} X_{ij} - 2\beta \sum_{(i,j)\sim(k,l)} Z_{ij}^{(t)} Z_{kl}^{(t)} \right)$$

actually in the original formula the exponential of this formula but in the code, instead of taking the exponantial of this code, the logarithm of the “random acceptance limit” is taken. This way is more stable.

Then if the flipping is accepted, the pixel of the result image is flipped. This process is done 500.000 times in each slave processor.

Communication Part

This part is where the slave processors send each other the data that might be needed by their neigbours, and the data received from neigbours to the corresponding arrays within each slave processor.

Firstly the odd numbered processors sends their first row to even numbered slave processors which are above them and the even numbered processors receive that data to their bottom array

Secondly the odd numbered processors sends their last row data to even numbered processors below them and the even numbered processors receive data to their top array

Thirdly the even numbered processors sends their first row to the odd numbered processor above them and the corresponding odd numbered processor receives the data in their bottom array

Lastly the even numbered processors send their last row to the odd numbered processors below them and the odd numbered processors receive the data to their top array.

This part is implemented carefully to avoid any deadlock situation. The slave processors. The detailed explanation of this part can be found in the code

Examples

Here you can see the different outputs of the program under different inputs and parameters.

Examples of Lena.png:

original image:

noisy image:

with parameters beta = 0.4 and pi = 0.15 :

beta: 0.4 pi = 0.45

beta = 0.3 pi = 0.01

beta = 0.8 pi = 0.15

here you can see if we tell the program its more noised, it will do more denoising. The closer we get to the original parameters of the image the more it will look like the original image.

Examples of the yinyang.png

the original image:

the noised image:

beta = 0.4 pi = 0.15

beta = 0.4 pi = 0.45

beta = 0.8 pi = 0.15

beta = 0.8 pi = 0.45

Improvements and Extensions

With using more processors and more iterations and making the image resolution better, we can reach better results but that will take more cpu time and memory. There is a trade of between the iteration count and the cpu time and with growing image size we need to make more iterations to make sure that each pixel is choosen randomly at least once. Also with using bigger array and reading and writing to bigger amounts of data the memory usage also will raise a lot. Taking these considerationsinto account, improvements can be done according to the needs of the user.

Difficulties Encountered

The most difficult part in the project was to avoiding deadlocks when sending and receiving data and because of the difficulty to handle the corner cases while determining the neigbours of a pixel, i needed to abandon my approach to determine them by using the arrays and i had to assign them to integers while taking into account all the corner cases which took lots of if/else statements and quite the time.

Conclusion

This was the first time i have coded something with MPI and used more then one processor while executing my code. It has its own difficulties of course, up it speeds up the things a lot. While the sequential code does about 5 million loops my code does 500.000 loops in different processors concurrently.

This project helped me a lot in terms of understanding the paralel programming and what can be done by using paralel programming.