CMPE300

ANALYSIS OF ALGORITHMS

TUNGA GÜNGÖR

PROGRAMMING PROJECT

25/12/2018

HASAN BASRİ BALABAN

2016400297

INTRODUCTION

In this project, we experienced parallel programming with C++ using MPI library. We implemented a parallel algorithm for image denoising with the Ising model using Metropolis – Hastings algorithm.

PROGRAM INTERFACE

User can interact with the program using a Linux terminal.

First thing that is needed to run the program is to download and install Open MPI from its official website. After the installation, the program can be run by following these steps:

1) Copy the main.cpp and the input files to open MPI directory.

2) Open a new terminal and change the directory to open MPI directory.

3) Compile the code by the command:

“*mpic++ -g main.cpp -o program*”

4) Then run the executable “program” by the following command:

“*mpiexec -n NUM_PROC ./program input_file.txt output.txt b_val pi_val*”

-> NUM_PROC stand for the number of processors that the program runs on. This number should divide 200 exactly, because the given input files contains Ising model of a 200x200 image.

-> b_val and pi_val are double numbers.

5) Step 4 will create an output.txt file which contains the denoised version of the image given by input.txt file.

6) In the end, the denoised image can be converted into a image file by using python script that given by the project. To run the script, type the following command:

“*python text_to_image.py ./output.txt out.png*”

This command will create out.png file using output.txt and display the denoised image.

INPUT & OUTPUTS

Input:

The program requires five different inputs to be able to run. These inputs are:

1) Number of processors that the program runs on. This number must divide 200 exactly since the images that we are given as input are in the size 200x200.

2) Input file. This file contains the Ising model of the image that we try to denoise. The entire file composes of +1s and –1s. The black and white pixels are shown as +1s and –1s.

3) Output file. This input is given to program to specify the path and the name of the output file.

4) Beta value. This is the parametrization value of Ising model.

5) Pi value. This value is the probability of a random flip on the original image. As an example, if the pi value is 0.1, this means that ten percent of the pixels of original image are flipped.

Beta and pi values are used later in the program to calculate acceptance probability to flip a random selected pixel.

Output:

The only output of the program is a noise free Ising model of the input file. This file is created in the given path as an input.

PROGRAM STRUCTURE

The program is implemented using parallel programming logic. And it runs on more than one processor simultaneously. The structure of the program is as follows:

-> First, the MPI environment is initialized.

-> Then, an if/else structure is used to determine which part of the program is run by each processor. This if/else structure is the core part of the program. The detailed explanation of this structure is given below.

The part which contains code for the master processor:

```
if(world_rank == 0){          //if it is the master process
    ifstream in(argv[1]);  //Takes input from input file
    freopen(argv[2], "w", stdout);
    for(int i=0; i<200; i++)
      for(int j=0; j<200; j++)
        in >> X[i][j];
    //Sends the necessary parts to each slave
    for(int i=1; i<=N; i++)
      MPI_Send(X[200/N*(i-1)], 200*200/N, MPI_INT, i, 0, MPI_COMM_WORLD);
    //Receives the parts of final array from each slave
    int Z[200][200];
    for(int i=1; i<=N; i++)
      MPI_Recv(Z[200/N*(i-1)], 200*200/N, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

In the part that given above, the master processor takes the input file and the output file as an input from the user and initialize the ifstream and freopen interaction. Then it reads the Ising model from the input file and creates a 200x200 size array to hold the data. After that, it sends all the data to slave processors by using MPI_Send function. The data which send to each processor is in size (200/N)x200. After sending the data, master processors use MPI_Recv function to receive denoised data from each slave processor. At the end, it prints out the elements of its final data array to output file. (For the sake of simplicity, the code for printing out is not included in the documentation.)

The part which contains code for slave processors:

```
else{
    int* temparr = NULL;
    temparr = (int *)malloc(sizeof(int) * 200 * 200 / N);
    MPI_Recv(temparr, 200*200/N, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    int subarr[200/N][200];
    for(int i=0; i<200/N; i++)
      for(int j=0; j<200; j++)
        subarr[i][j] = *(temparr + i * 200 + j);
    free(temparr);
    srand(time(NULL));
    for(int t=0; t<T; t++){
      int* temp = NULL;
      int* temp2 = NULL;
      temp = (int *)malloc(sizeof(int) * 200);
      temp2 = (int *)malloc(sizeof(int) * 200);
      MPI_Recv(temp, 200, MPI_INT, world_rank-1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      MPI_Send(subarr[0], 200, MPI_INT, world_rank-1, 0, MPI_COMM_WORLD);
      MPI_Send(subarr[(200/N)-1], 200, MPI_INT, world_rank+1, 0, MPI_COMM_WORLD);
      MPI_Recv(temp2, 200, MPI_INT, world_rank+1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      int upper[200], lower[200];
      for(int i=0; i<200; i++){
        upper[i] = *(temp + i);
        lower[i] = *(temp2 + i);
      }
      free(temp); //Free the dynamically alocated memory
      free(temp2); //Free the dynamically alocated memory
      int i = rand() % 200/N, j = rand() % 200, sumos = 0;
      for(int k=max(i-1,0); k<=min(i+1,(200/N)-1); k++)
        for(int l=max(j-1,0); l<=min(j+1,199); l++)
```

```
                sumos += subarr[k][l];
            sumos -= subarr[i][j];
            if(i==0)
                sumos = sumos + upper[j-1] + upper[j] + upper[j+1];
            else if(i==(200/N)-1)
                sumos = sumos + lower[j-1] + lower[j] + lower[j+1];
            double delta_E = -2*gamma*X[i][j]*subarr[i][j] - 2*beta*subarr[i][j]*sumos;
            if((log((double) rand()/RAND_MAX + 1.)) < delta_E)
                subarr[i][j] = -subarr[i][j];
        }
        MPI_Send(subarr[0], 200*200/N, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
```

The above code is for intermediate slave processors. These processors should be able to communicate with their neighbors. Initially, each processor takes it data from master processor and keep it locally on a (200/N)x200 size array. And then, processors perform several iterations on this local data to denoise the image part that given to them. The iteration number is constant and determined at the beginning of the program. (Iteration number for each processor: 500000 / N, where N is the number of slave processors)

At each iteration, the processors perform the following operations:

-> Receive and send border rows of data array to upper and lower neighbor. (MPI_Send and MPI_Recv are used for this communication.)

-> Create two random integers between 0 (included) and 200 (excluded) to select a random coordinate on the array. Then calculate the sum of the pixels around that random selected pixel.

-> Then, calculate the acceptance probability which formula is given in the project description.

-> At the end, take a uniformly distributed random double number between 0.0 and 1.0 (say x) and flip the randomly selected pixel if the acceptance probability is higher than x.

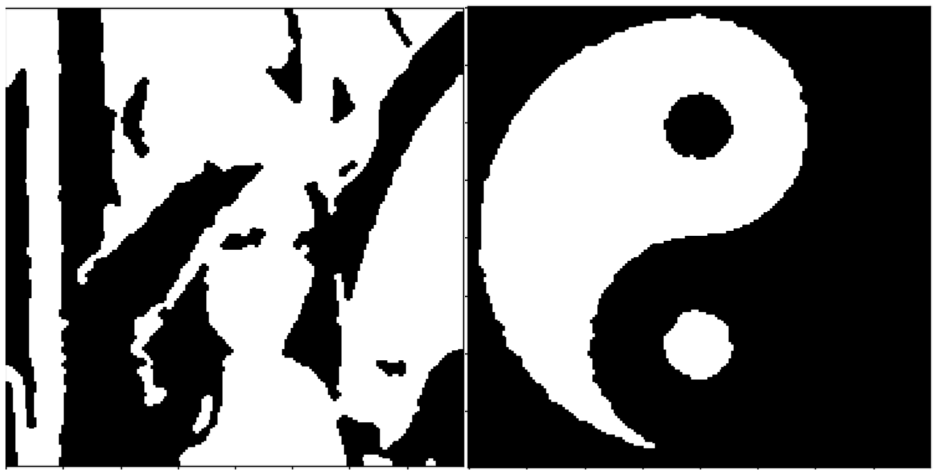
After the iterations end, the processor sends its final array to master processor using MPI_Send.

To keep this documentation shorter, I didn't include the code parts for the first and last processors. In the code given above, the intermediate processors are communicating with two other processors (upper and lower), but in the first and last processors, the communication is done only with one other processor. First processor only communicates with the second processor, and the last processor only communicates with the processor which has a world rank N-1. The only difference (both logically and as implementation) between intermediate and these first and last processors is the number of communications.

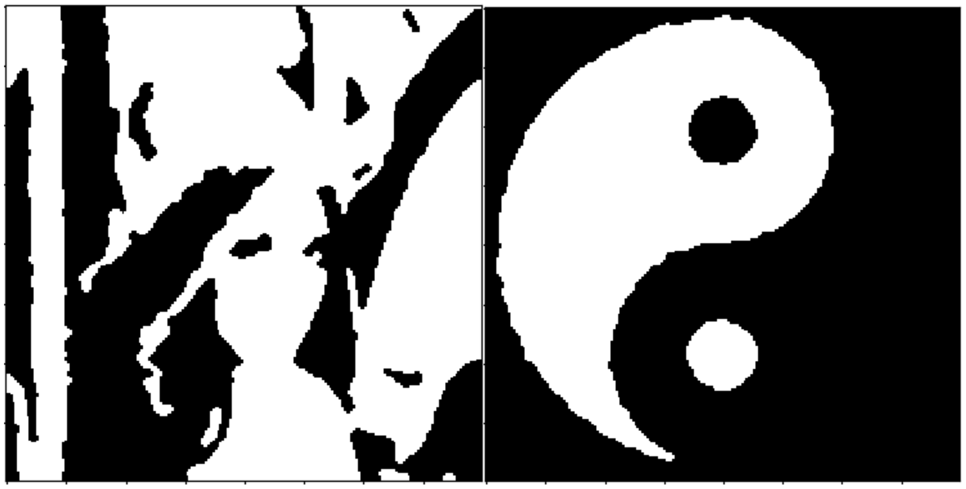
EXAMPLES

Several examples of the program outputs for the different number of processors, pi and beta values are given below.

Number of processors: 5, Beta: 0.6, Pi: 0.2



Number of processors: 11, Beta: 0.8, Pi: 0.15



DIFFICULTIES ENCOUNTERED

While implementing this project, the only difficulty that I think about was to prevent the deadlocks that can occur while processors are communicating.

CONCLUSION

In this project, our aim was to implement a parallel programming algorithm with C++ and MPI library. Project helped me to understand how parallelization is done in real code environment and gave me a new insight about developing parallel algorithms.