**Documentation**

**Title Page**

**Course ID: 300**

**Course Name: Analysis of Algorithms**

**Student's name: Sait Talha Nisanci**

**Type of The Project: Programming Project**

**Name of The Submitted Person: Metehan Doyran**

**Submission Date: 12/15/2016**

**Introduction**

**We are given a 200\*200 image and we are supposed to determiine horizontal, vertical, y=x and y=-x lines in the image. In order to make this faster we are supposed to use parallel programming approach while solving the problem which will make it faster. Very detailed of the problem description is already given in the project description, therefore we will not give so much details in this part.**

**Program Interface**

**In order to run the program, you should go to program's directory and open terminal. You can compile the program or you can directly use the executable file to run the program. In order to compile the program to produce the executable by yourself you have to write "mpicc main.c -o prog.exe", which simply means that it will give prog.exe as output. You can this name to anything for example "mpicc main.c -p weirdo.exe" would produce the exactly same program as previous one without any errors. After that in order to run the program there are some parameters that you have to use. The general syntax of how you can run the program is as follows : "mpiexec -n amount\_of\_processors ./prog.exe input.txt output.txt threshold". Here mpiexec is necessary to run the program as it is an mpi program. -n means it will be followed by the number of processors. The number you choose will have one master processor and the rest will be slave processors, where the amount of slave processors must divide 200 without remaining. An example of a valid number of processors to be passed as a parameter is 3,5,11,21,51,41 etc. There is a special case for 201, when you pass 201 as number of processors the program will give bus error sometimes because 201 is a very high number for number of processors, so we advise you not to try the program with 201 processors. The number of processors is followed by the name of the program. After that we have name of the input file. You should pass the file that contains the initial image as the input file. The passed image must be 200\*200. After this parameter you have to pass output file name. This output file will have the values 0 or 255, black and white pixels relatively which will be found after executing the algorthm, the details about the execution of the algorithm will be given in the next section. After this parameter threshold parameter follows. This parameter will be used as threshold while filtering the array. The greater this value the more black the output image will be, because it will be harder to put 255, white pixels, to the output. An example command to run the program is "mpirun -n 3 ./prog.exe input.txt output.txt 10" . For threshold you should choose one of the 3 values, 10, 25, 40.**

**Program Execution**



**When you run the program with the given example input and with threshold 10. You will get an output like this. As you can see there are a lot of white pixels because the threshold is kept low. By this way the chance of detecting lines is more and you are not that strict about detecting lines.**



**When you run the program with the given example input and with threshold 25, you will get an output like this. This output image has less white pixels compared to the previous one with threshold 10 because it is harder to pass the threshold. This way you are more strict about detecting lines.**



**This is the output when you run the program with threshold 40. As you can see white pixels are not that much because you held the threshold very high which makes it harder to find values that will be greater than the threshold.**

**By looking at these images one can infer that the higher the threshold, the less white pixels the image has. While trying different images it is important to choose threshold as one desires so that they will get the desired output.**

**Input and Output**

**Input should be 200\*200 matrix where each value of the input will be an integer and the range for each value is [0-255]. It is preferable that after 200 integers there will be a break line.**

**For output there will be a 196\*196 matrix, where each entry of the matrix is either 0 or 255. The output file will have line breaks at each row.**

**Program Structure**

**Getting Input:**

**In the first part of the program we get the input from input.txt. We store the array in original\_arr as 1D, since this will make it much easier to scatter for slave processors. The master processor is responsible for getting the input since we need to get the input only once. We also hold a counter for indexing of original\_arr. We used fsancf here since it is a faster way of reading when we know the type which is always an integer in this project. We have a MPI\_Barrier at the end since we shoudnt start scattering before completing getting the input.**

**Scattering the Image:**

**We have a "local" array for each processor which will get the input from original\_arr of master processor. We also have a sub\_arr for each processor which will hold the local array as 2D array to make smoothing process easier without having to deal with indexing.**

**Send and Receive:**

**In order to start smoothing we need to get boundary rows of neighbor slave processors. In order to make things easier we are going to get one array for last row of (rank -1) th processor and we will store it in received\_arr which is a 1D array. And we will get first row of (rank+1) th slave processor and we will store it in received\_arr2 which is a 1D array. We will update sub\_arr with those rows and now we have all the values for smoothing. While sending and receiving data we are doing it for odd values and even values seperately to prevent any situation that could lead to a deadlock.**

**Smoothing:**

**While smoothing since we have established everything to make things easier, we will just sum 9\*9 matrix for each value of sub\_arr and we will divide this by 9 and get only the integer part. After smoothing we will store each smoothed value in smoothed\_arr for each process. We add MPI\_Barrier to prevent any incorrect results. After this stage we are going to lose 2 rows and 2 colomns.**

**Send and Receive:**

**After smoothing process we need to send last row of (n-1) th slave processor and first row of (n+1) th slave processor. Again we are going to store these values in receive\_arr and receive\_arr2 relatively. We add MPI\_Barrier to prevent any incorrect result. We do the exchange in two steps. For each step we do it for odds and evens seperately to prevent any deadlock.**

**Filtering:**

**After getting necessary rows from the last and first rows of neighbors, we are going to update sub\_arr with these values and with smoothed\_arr.**

**We are given 4 different matrix to determine vertical, horizontal, y=x or y=-x lines. So we will store create a variable isAbove and we are going to iterate over sub\_arr and we are going to apply the filter and for any of the 4 different filters if there is a value greater than the chosen threshold then we are going to make our isAbove variable 1, which indicates that this value should be 255, or white, in pixel at the output file so we create an array filtered\_arr and we put 255 to this part or if there is no value that is greater than threshold after applying 4 different filters we put 0. We are going to put MPI\_Barrier at the end to wait for all slave processors to be done with their parts.**

**Gathering:**

**After finishing filtering in each slave processor we need to gather those values in the master processor. In order to accomplish this we are going to create an array called final\_arr which will store the filtered array for the master processor. We will use MPI\_Gather for this and we are going to put MPI\_Barrier at the end to wait for the process to be completed after continuing.**

**Outputting:**

**After gathering all the filtered arrays in the final array of the master processor we need to output this to a file which was passed in the parameter while running the program. We are simply going to iterate over the final array and we will hold a counter which will put a break line when it reaches to 196 since the output is going to be 196\*196. After this we put a Barrier and we finish our program with the instruction MPI\_Finalize which is necessary to end the program.**

**Improvements and Extensions**

**The program is designed for only 200\*200 pixel images and it is designed for only certain number of processors. This can be improved by making the program more generic which is a good idea because the images will not always have 200\*200 pixels. It is also good to calculate the optimum number of processors so that we will not waste the energy. It can also be given as parameter whether they the user care about power or not or whether they care about speed or not. If they care about speed more processors can be used to run the program so that it will give result faster technically.**

**Difficulities Encountered**

**The project was not a hard project but it was a little bit hard to send 2D array with MPI\_Send. In order to solve this difficulity it was a good idea to convert it to a 1D array and send it like that. By this way we only needed to adjust indeces, which wasn't a big issue.**

**Conclusion**

**This was a nice projet, and it is a useful program to determine lines in a 200\*200 image. It needs to be generalized though for any size of input.**

**Importance of The Threshold**

**It is important to note the importance of the threshold. We only consider 3 different thresholds here. 10,25 and 40. When we choose 10 there will be a lot of white pixels as we are holding the threshold very low. There will be a little bit less white pixels for 25, and there will be even less for 40.**

**Benefits of Parallel Programming**

**In this project parallel programming is a good approach to handle this problem when we have a lot of processors because the problem is independent in itself. We can use a lot of processors and they can all work at the same time. They will only need to wait for each other while sending and receiving data which is not a big amount of time. On the other hand when we are using sequential algorithm it will need to iterate all the image by itself. Even though the execution times will not be that different since we are talking about 200\*200 which is 40000, which takes a very little amount of time in terms of modern CPU's, with an image greater than this it can be very useful to use parallel algorithm to solve this problem.**

**Appendices**

**The source code is included in the zip file.**

**The different images and results are included in the zip file for 3 different thresholds.**