CMPE 300 - PROGRAMMING PROJECT

PARALLEL LINE DETECTION USING MPI ENVIRONMENT

December 27, 2016

Student Name: Rahmetullah VAROL
Student ID: 2016700159
Bogazici University
Computer Engineering Department

1 Introduction

In this project a simple smoothing and line detection algorithm is implemented under a parallel programming scheme, using the MPI environment. A line detection algorithm is used to detect and highlight the boundaries between different regions on an image. These different regions might be different objects in a scene as well as different parts of an object. Edge detection is a fundamental concept in image processing and may be used as a preparatory step in many algorithms for problems like feature extraction or object detection.

The method used in this project is based on the usage of convolution kernels throughout the input image. At first, the input image is convoluted with a kernel tuned for blurring the image and the with kernels tuned for detecting horizontal, vertical and oblique lines in the image. The blurring step is for reducing the noisy edges and obtaining a smoother result. After the line detection kernels are applied a different line image is obtained for each kernel. The results are then gathered into a single image, in which a pixel is highlighted if and only if any of the corresponding pixels on the line images exceed a given threshold value.

This algorithm is highly suitable for parallel programming because we need to do many independent but identical operations on different parts of the image. Also, the 2D structure of images and the uniform application of the algorithms make it easy to distribute the work load into balanced segments.

2 Program Interface

The program can be run from a command line in a system with MPI capability (i.e. the 'mpiexec' program can be run), using the following command:

```
mpiexec –n $<$Number of processors$>$ lineDetection $<$Input file name$>$ $<
    $Output file name$>$ $<$Threshold value$>$
```

The threshold value is used in merging the intermediate line images into a single final line image.

3 Program Execution

This program is used to apply line detection on a given gray-scale image file. The input image should be provided as a text file of intensity values of the pixels, in which each line represents a row in the image. The output is a likewise text file of a black and white image which is resulting line map of the algorithm. The options available for the program the number of processors to be used and threshold value to use in merging the intermediate line images. These parameters are set using the execution command of the program stated in the previous section. The <Number of processors> and <Threshold value> fields correspond to these parameters respectively.

4 Input and Output

As stated in the previous section the input image should be provided as a text file representation of a gray-scale image. In this text file line breaks are used to identify the ends of rows in the image and spaces are used to identify distinct pixels throughout a row. The width of the image is detected using the first line in the input file, and naturally, it is expected that each row contains a uniform number of pixels. The row count of the input image should be divisible by the number of slave processors. Also, for the convolution operation to be defined each slave processor should be able operate on at least 3 rows. So, the number $\frac{\text{Height of the image}}{\text{Number of slave processors}}$ should be at least 3. Failing to provide a valid input file will result in an undefined behaviour.

The output of the image is a black and white image that is the resulting line map of the line detection algorithm. So, the output file will only contain the intensity values of 0 and 255. The output image width and height will be 4 pixels smaller than the input image. This is due to the convolution operations applied on the image.

5 Program Structure

The structure of the program can be thought to consist of two separate operations. One set of operations are used to divide the input image into slave processors and communicate rows of the divided images between slave processors and the other set of operations are used to apply image processing algorithms on the divided images. The image processing operations are only used on the slave processors. The master processor is used only for distributing and gathering the input and output images and for synchronizing the progress of the slave processors. The program execution can be divided into 8 following steps

1. Reading the input image from the given input text file.
2. Distributing the input image to the slave processors.
3. Extending the partial images in the slave processors.
4. Applying blurring kernel into the extended partial images.
5. Extending the resulting blurred images.
6. Applying line detection kernels into the extended blurred images.
7. Merging the intermediate line images into a single line image using thresholding operation on each of them and taking their union.
8. Gathering the partial line images in the master processor and writing the final image into the given output text file.

The extension operation used in the 3rd and 5th steps should defined more clearly. After the initial image is distributed to the slave processors, special care must be taken when the topmost and

bottommost rows are being convoluted. Because for the topmost rows the the pixels corresponding to the upper elements of the kernel does not exist in the current processor, because they are on the previous processor. Similarly the lower pixels are on the next processor for the bottommost pixels. This problem is shown in [Figure x]. In order to circumvent this problem, we can retrieve the last row of the previous processor and the first row of the next processors and append them in to the rows of the current processors accordingly. This will result in the desired partial blurred image for the current processors. Naturally, for the first and the last slave processor, there will be no retrieval from the previous and next processors (because they don't exist) respectively.

The parts of the program will be explained in more detail in the subsections.

5.1 Data Structures

There are two structures that should be that should be taken into consideration for this problem. Namely, the image and kernel structures. Both structures are represented in the program using two dimensional arrays, and the arrays are implemented using double pointers so that their sizes can be set in runtime. This allows for processing of variable sized input images. The array representing the image and kernels are enclosed within a structure along with width and height variables for the images and size variable for the kernel. This method allows to implicitly keep track of the image size and kernel size throughout the program. Thus, generic methods can be used to apply on images without calculating their sizes explicitly.

5.2 Reading the Input File

The image is read only in the master processor into an image structure as described in the previous section. As a first step the image width and height is detected from the input text file trivially, by counting the spaces in the first row (image width) and counting the newlines throughout the text file (image height). Next, sufficient memory is allocated in the array of the image structure to store widthheight number of pixels. Subsequently, the pixels are read from the input text file.

5.3 Communication Between Processors

The image data is communicated between the processors throughout the execution of the program. First, communication occurs right after the image is read, through which the image is distributed to the slave processors. In order to achieve this, the slave processor count is detected using the functionality provided by the MPI environment, the number of rows each slave processor must operate is calculated and that many rows are sent to each processor in order of increasing rank. Simultaneously, the slave processors read the rows sent by the master processor. The two dimensional array structure is very convenient for this scheme, since a consecutive memory block can be communicated between processors using the 'MPI_Send()' function. After the images are distributed, the partial images in the slave processors must be extended as described in 4. This is done by sending the last row to the next processor in each slave except the last one and sending the first row to the previous processor in each slave except the first one. The slave processors then

receive these lines from their adjacent processors and append them to the partial images they possess. The convenience of the image structure is highlighted in the appending operation, since this can be achieved by reallocating the memory of the image array to accommodate one more row than it already holds and the necessary allocation can be made using the size information kept with the image data. One more extension operation is performed after the blurring step because a similar dependency between the slave processor arise in the application of the line detection kernels. After the line detection step, the partial images are sent to the master processor. The final image is constructed by appending the received image rows consecutively in the master processor.

5.4 Image Processing

The main reason of using the MPI environment is to parallelize the image processing operations in the slave processors. So the image processing is only done in slave processors on the partial images. The first image processing operation is the blurring step. This is achieved by convoluting the images with the following kernel:

$$\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

This kernel has an averaging effect on the pixels so that the small distortions throughout the image are suppressed. This operation is also called box blurring and is one of the simplest blurring operations. The next image processing operation is the application of line detection kernels. These kernels are as follows:

$$\begin{bmatrix} -1 & -1 & -1 \\ 2 & 2 & 2 \\ -1 & -1 & -1 \end{bmatrix}, \begin{bmatrix} -1 & 2 & -1 \\ -1 & 2 & -1 \\ -1 & 2 & -1 \end{bmatrix}, \begin{bmatrix} -1 & -1 & 2 \\ -1 & 2 & -1 \\ 2 & -1 & -1 \end{bmatrix}, \begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix}$$

These kernels are used to detect horizontal, vertical and diagonal (+45 and -45 degrees) line respectively. As a result of these convolutions four line images will be created, each image consisting of lines of corresponding nature. These images then need to be merged into a single line image which contains all the horizontal, vertical and diagonal lines. In order to achieve this, a threshold value is determined (from the user input) and every pixel on each of the line images are examined. If in any line image the threshold value is exceeded in a pixel, the corresponding pixel in the final image is highlighted by setting its value to 255. In other words, the threshold value acts as filter which highlights the pixels with a dominant line characteristic.

Threshold values of 10, 25 and 40 are tested with the program. As the threshold value increases fewer number of pixels are accepted as a line because a higher intensity difference is needed in order for a kernel window to be able to produce a value that will exceed the threshold. This reduces the number noisy pixels in the final image, but if the threshold value is too high the algorithm fails to detect lines in the lower contrast parts of the image.

6 Examples

The execution steps and the result of the program for a threshold value of 10 is exemplified in Figure 1 for two different inputs. In this figure; the input images are given in Part (a), the smoothed partial images are shown in Part (b), the edge detected partial images are shown in Part (c) and the final images are shown in Part (d). As can be seen from this figure this algorithm works well for images with straight lines and even though the used kernel are tuned for horizontal, vertical and diagonal lines, the algorithm also works well for round edges.

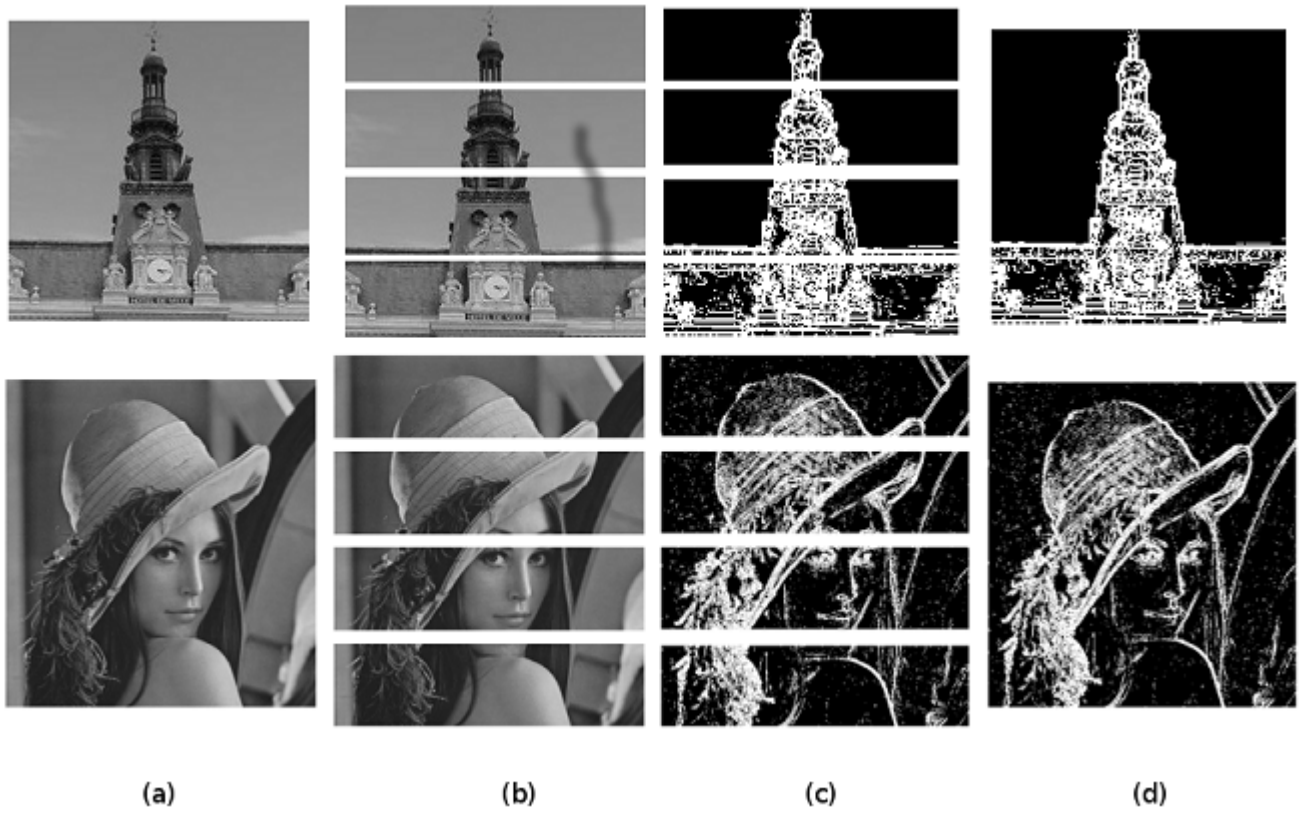

Figure 1: The intermediate and final results of the program for some example inputs

7 Improvements and Extensions

One problem with this algorithm is that, since the threshold value is fixed and received from the user, it may not give satisfactory results for images which have parts with different contrast levels. For low contrast parts of the image a lower threshold value is required but this will result in noisy edges in the high contrast parts of the image. For higher values of threshold the algorithm may not yield the edges at the low contrast parts of the image. In order to circumvent this problem, we can set add a histogram equilization step to the process before applying the line detection kernel. We can also use an adaptive threshold value which detects the contrast value of the neighbourhood of the operated pixel and changes the threshold value accordingly.

Another improvement to the program would be to get rid of the requirement that the row count of the image should be evenly divisible by the number of slave processors. This can be done by finding the greatest value evenly divisible by the slave number count that is smaller than the row

count of the image. If we denote the number of remaining rows by m, we can then increase the number of rows that will be distributed to the first m processors by one. Using this method we can use an arbitrary number of slave processors.

8 Difficulties Encountered

The most prominent difficulty encountered during the project was to keep track of the memory leakages. When doing garbage collection in the slave processors, we should be careful not to try to free any memories that we have not allocated. The Image structure used in the program makes things easier because we are relieved from the burden of keeping track of the image sizes.

9 Conclusion

In this project we have implemented a trivial image processing algorithm using the MPI environment. We have seen, how a problem can be divided into independent subproblems in order to be solved using parallel architectures. The parallelization method is especially suited for some problems which are highly parallelizable. Some image processing algorithms are among these problems since we can operate on different parts of the image independently of other parts. Spotting such problems and constructing suitable parallel algorithms for them allows us to utilize parallel architectures optimally. This project also demonstrates the importance of writing explicitly parallel software in order to maximize the efficiency of parallel architectures.