INF3201 - Parallel Programming

Assignment 2 - Parallel Raster Image Filtering using MPI and OpenMP

Zulfiqar Ali, Muhammad Nauman Ali

Table of Contents.

Introduction

Raster image represents a two dimensional image as a rectangular matrix of pixels. A raster is technically characterized by the width and height of the image in pixels and by the number of bits per pixel. For RGB, it is 8 bits per pixel with a maximum value of 255. Filtering of images

involves operations on pixels to enhance image or reveals some particular aspects of image. For example removing noise, color enhancement, blurring image etc. image processing/filtering basically involves operation on pixels with convolution matrix which depends upon the choice of filter you are applying to it. Sobel filter or sobel-feldman operator is particularly used for edge detection.

Implementation

Sequential

Firstly, for sobel implementation, the image is loaded into memory and also the same image size memory is allocated for output image after processing. In order to implement the sobel filter it is required to convert the image to grayscale. So the grayscale function is applied on the input image before any processing; however I prefer to process without applying grayscale because from experiments it doesn't affect the image in our case, so I didn't want to increase the time complexity of the solution. Two convolution matrices of 3x3 size, one for horizontal and for vertical changes are convoluted with the 2 dimensional image pixel by pixel.

$$G_x = [[\,1,\,0,-\,1],[\,2,\,0,-\,2],[\,1,\,0,-\,1]] * \mathrm{A}$$

$$G_y = [[\,1,\,2,\,1],[\,0,\,0,\,0],[-\,1,-\,2,-\,1]] * \mathrm{A}$$

Where Gx is the convolution matrix for horizontal changes and Gy is for vertical changes approximation and also A is the two dimensional image. Then gradient approximation is calculated as:

$$G = \sqrt{G_x^2 + G_y^2}$$

After approximating the gradient, if the value has exceeded the range i,e 0 - 255 it is set to 0 or 255 depending upon the direction it exceeded, and is given to each color channel of that pixel.

Pseudo code for sobel filter

```
function ApplySobel(*img, width, height, *out_img)
   sobel_x[3][3] ={ { -1, 0, 1 },{ -2, 0, 2 },{ -1, 0, 1 } }
```

```
   sobel_y[3][3] ={ { -1, -2, -1 },{ 0,  0,  0 },{ 1,2,1 } }
      for i = 0 to width
            for j = 0 height
                  initialize xpixel,ypixel to 0
                  for k=0 to 3
                        for l=0 to 3
                              xpixel += sobel_x[k][l]* sobel[w * (j + k - 1) +
(i + l - 1)].blue;
                              ypixel += sobel_y[k][l] * sobel[w * (j + k - 1)
+ (i + l - 1)].blue;
                        end for
                 end for
                pixel = sqrt((xpixel * xpixel) + (ypixel * ypixel));
                   chcek if pixel < 0 or > 255 set to 0 or 255
                   Put the value to all colors pixel
            end for
      end for
   End function
```

Secondly, the box blur filter blurs the image based on the average value of neighboring pixels. The blurring is based on the size of the kernel, so for the 3x3 kernel, a pixel with its 8 neighbors is averaged and the resulting value is the blurred value. kernel convolution matrix of size 3x3 is applied with each pixel weight the same equal to 1.

Boxblur function call

```
function ApplyBoxBlur(*img,width,height,*out_img)
   set image to new image
   for i=0 to width
      for j = 0 to height
            initialize red blue green to 0
           for k=0 to 3
               for l=0 to 3
                 blue += image[width * (j + k - 1) + (i + l - 1)].blue;
                 green += chunk[width * (j + k - 1) + (i + l - 1)].green;
                 red += image[width * (j + k - 1) + (i + l - 1)].red;
              end for
            end for
       Divide blue,green,red by 3
       If blue,green,red < 0
            set to 0
       else if(blue,green,red >255
             Set to 255
       Put the value to out_img at i,j location
       end for
```

```
    end for
end function
```

So the first step is allocating memory for an output image equal to the original image. Then accessing the image pixel values by iteration. Then for each pixel color the neighbors 8 pixels values are added without any multiplication with the kernel matrix value because of the same 1 weight for all. Then each color pixel is divided by the size of the kernel matrix which in our case is 9, because of the 3x3 kernel and in case the resulting value has exceeded the range of 0-255 for 8 bit pixels it is set to min or max based on the direction of exceeding. And in the last step is putting each color pixel value to the new output image with the same pixel.

Parallel Solution

Applying sobel and box blur filters on an image is done through MPI and openMP for parallel implementation. MPI is implemented using MPI_send and MPI_recv function. This way the specific part of the image is passed to every process, in every process there is a call to the sobel function. So, to divide the image among different processes, image height is divided upon size of MPI i,e number of processes we want to generate and is called chunk.

All nodes will receive the distributed image through MPI Send and MPI Receive. The size of MPI determines how the image is actually distributed. Every node will add two more adjacent rows, one from top and other one from the bottom, to address the border pixel issue. The filter functions then will begin processing the image block's second row and continue until excluding the last row. One thing to keep in mind about the implementation specifics is that the rank 0's will only read and distribute the picture before arranging and writing back to the file system once it has received all of the parts.

And one thing to be noted is that sobel and boxblur functions are only producing the result or output image equal to size of an image based on the chunk size.

OpenMP is implemented to parallelize the loop's iterations inside both functions before the first loop. Basically the functions are now processing a chunk of image and putting the result back into the output array, and also that chunk of image processing is divided into different threads to run in parallel. So OpenMP threads are dividing the processing of the chunk of an image. And in this way both MPI and OpenMP are implemented.

Time Performance Analysis

After performing different tests on the different image sizes, the results we got from sequential and parallel solutions are given below. The speed up, efficiency is calculated through time taken by computation of different sizes. Time performance is analyzed for both sobel and box blur filter collectively

$$speed = t_s \div t_p$$

$$efficiency = speed \div No.\ of\ cores/\text{processe}$$

Time taken by sequential solution

The graph below shows the overall time taken by three different sizes of images for sobel and box blur filters combined. As expected with the increase in size of image the time it takes has also increased, because of higher computations involved.

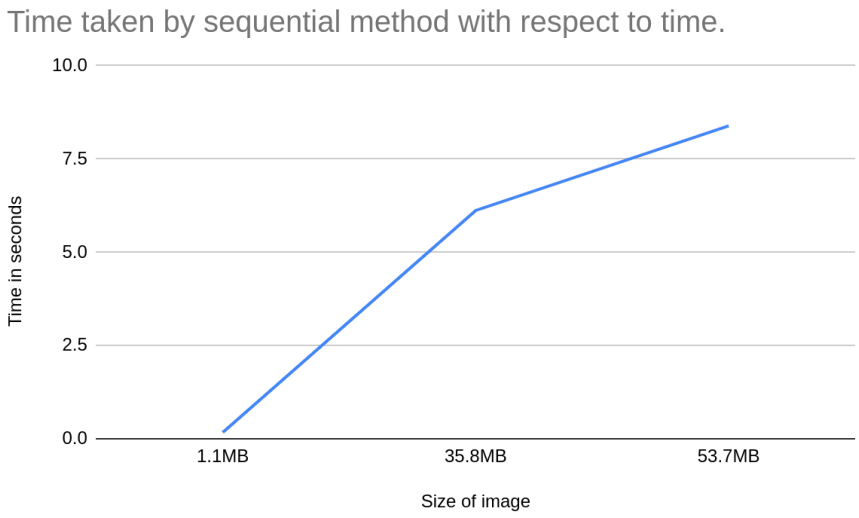

Fig 1, time taken by different image size in sequential solution

Time taken by parallel solution

The below graphs represent the time with different numbers of cores and their speedup ratio and efficiency regarding different sizes of images. So in a parallel solution MPI and openMP are combined and in below graphs the calculations are based on a fixed number of threads i,e 8. Because of best run time as compared to other sizes of threads from experimentation.

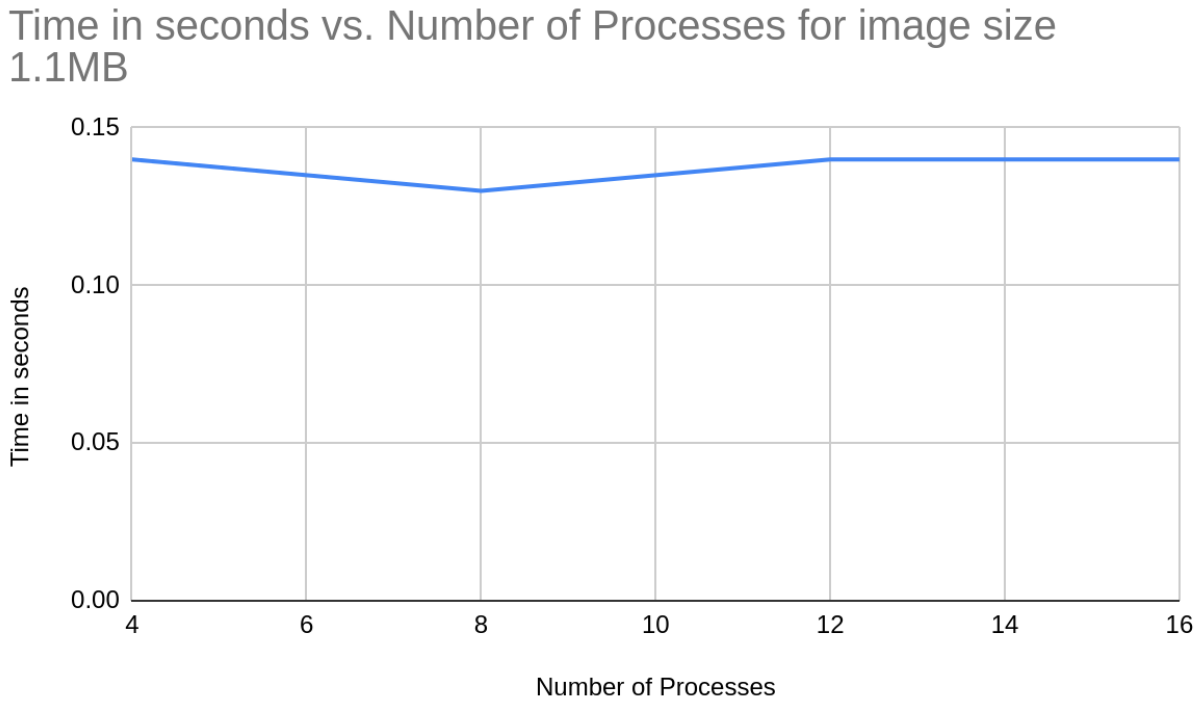

Fig 2, time for increasing number of cores

1.1MB

Number of Processes	Speedup	Efficiency
4	1.214	0.303
8	1.307	0.163
12	1.214	0.101
16	1.214	0.075

In comparison to the sequential solution with the image size 1.1 MB, it hasn't performed well in parallel with the increasing number of cores, because of increasing network overhead.

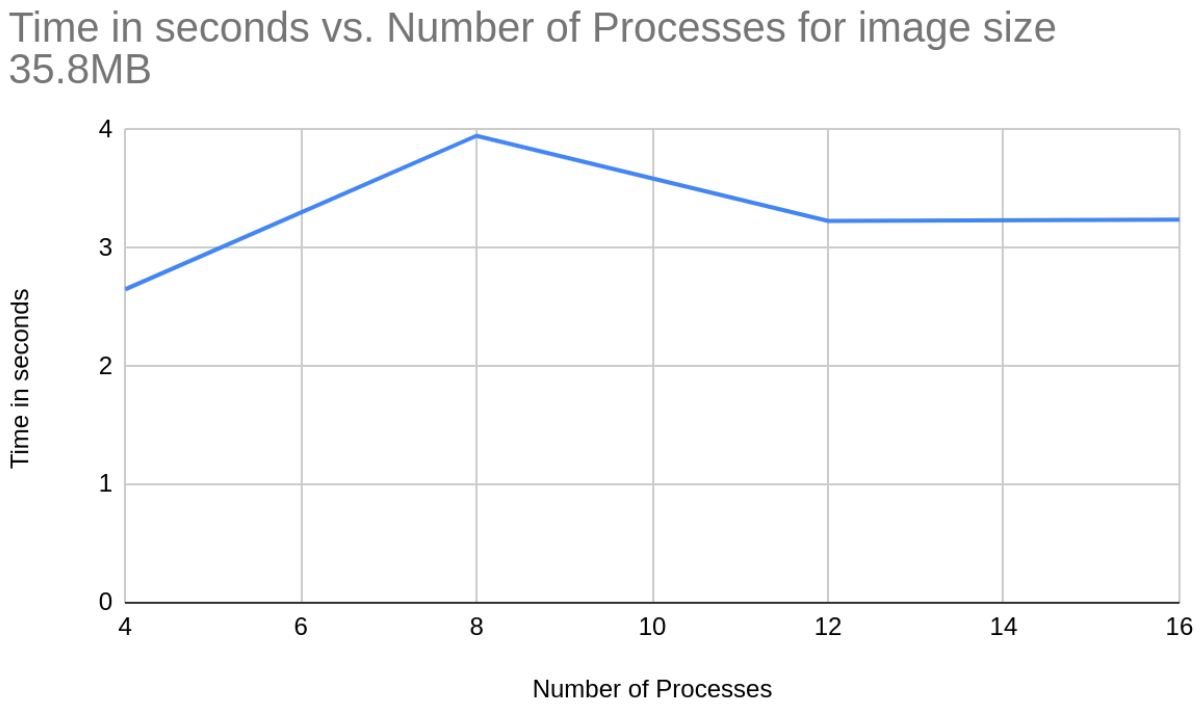

Fig 3, time vs number of processes with image size 35.8 MB

Number of Processes	Speedup	Efficiency
4	2.309	0.577
8	1.549	0.193
12	1.894	0.157
16	1.888	0.118

The parallel solution with input image of size 35.8 MB has dropped down to half of its time as compared to sequential solution on 4 cores and 8 threads, which has the highest efficiency.

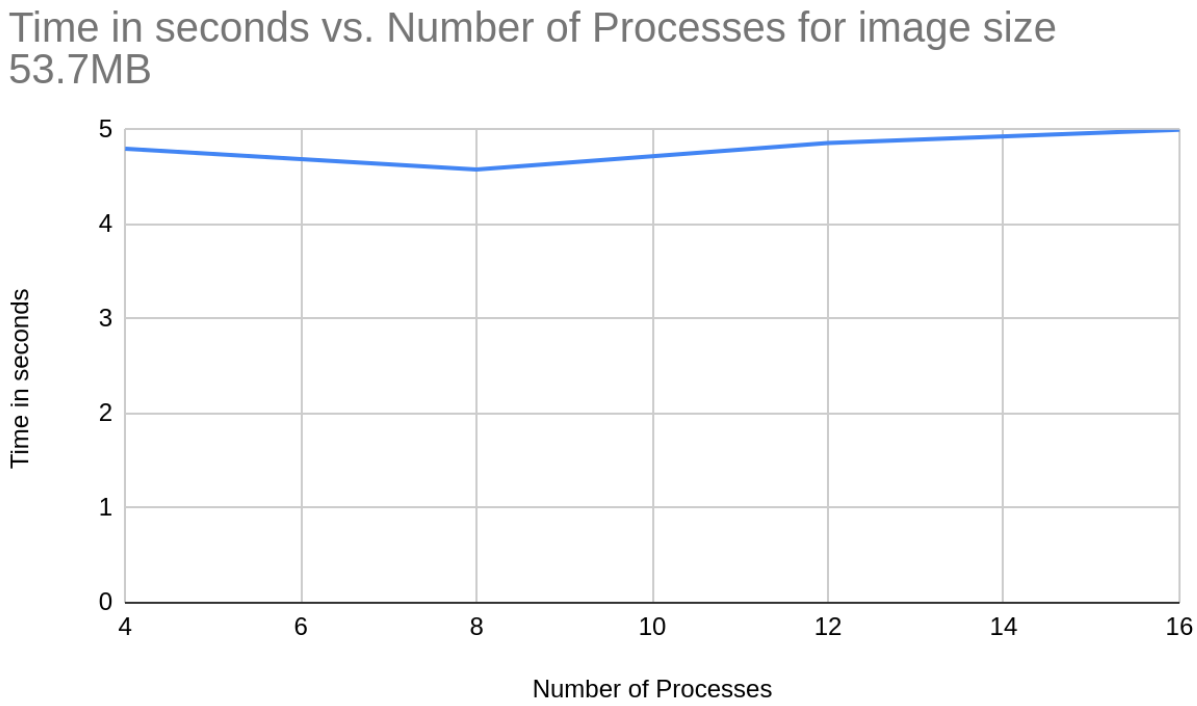

Fig 4, time vs number of processes with image size 53.7 MB

Number of Processes	Speedup	Efficiency
4	1.747	0.436
8	1.831	0.228
12	1.726	0.143
16	1.678	0.104

The parallel implementation of box blur and sobel filter has good performance with efficiency of 0.43 with number of processes of 4.

So the general observation from those graphs is that performance of parallel solution would be much better with higher image size.

Note: The hardware details of the system used in these experiments are given below

Kernel name	Linux

Processor	x86_64
Operating system	GNU/Linux

Discussion

From the experiments performed on different sizes of images. For the larger photos, the parallel solution has significantly reduced runtime. Due to the way the load is distributed among the cores, this method scales well for large photos. The splitting of an image and processing of the image will depend on the number of nodes chosen for processing. Smaller inputs cause the solution to take a long time. Using MPI, runtime can be significantly improved for greater input. We can observe certain yields in the runtime in addition to OpenMP, but not many. For the additional iterative operations on the filter functions, we can simply combine this with OpenMP. The smaller image pieces make it simpler to layer other optimization techniques on top of this approach. The solution's main drawback is that it isn't designed for ideal load distribution. The last node processes any additional rows or pixels that were left out after the algorithm divides the image. For getting the optimal solution of dividing the image across multiple processes and chores, in a way that if a process needed some extra row for processing it could get or request that part from another process.

Conclusion

Sobel and box blur image processing techniques were implemented using both sequential and parallel techniques. And then performing experiments with different sizes of images with different number of chores and results showed that the higher computation time can be reduced significantly using parallel techniques. But this kind of solution can perform better in a significantly higher size of computation. Because in calculating the time it took you also have to include the network overhead in case of significantly smaller size of computation accounts more for the time complexity than the actual computations performed.