TDDC78 - Programming of Parallel Computers

Lab1 & Lab2 - Image filters with MPI and pThreads

Authors:
David Tran - Davtr766
Jakob Bertlin - Jakbe457

I. Introduction

In the course TDDC78, Programming of Parallel Computers, the first two labs focuses on Message Passing Interface (MPI) and POSIX Threads (Pthreads) and how to write efficient parallelized algorithms in order to solve a problem. For the labs presented in this report, the available images was read into the program and then converted to a one-dimensional unsigned char array. Therefore, the images are represented as a one-dimensional array with three channels *r,g* and *b*.

II. Lab 1 (MPI)

In the first lab, *MPI* was initialized and the root process reads the input image and divides it into chunks depending on the amount of available processes and then scatters the chunks to the other processes in the communication world. An illustration of the function *MPI_Scatter* can be seen in figure 1. However, the chunks were sent to the other processes differently depending on which filter was used.

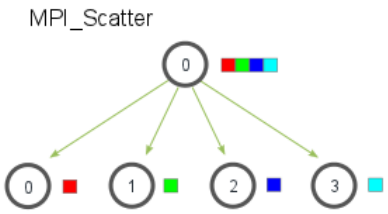

Fig. 1: A visual representation of how MPI_Scatter works. The root process divides the image into chunks and then send each chunk to each available process.

A. Blur filter

The image is divided into p number of chunks with y-size y/p where y is the vertical image size of the image and p is the total amount of processes available for use. An example can be seen in figure 2. Given a radius, the blur filter works by taking the average of one pixel in the rectangle-shaped neighborhood whose size depends on the radius.

A problem arises when the processes receives the chunks of the original image from the root processor. When calculating the average pixel of the top and bottom part of the image chunk with a radius greater than one will render errors because you need pixels from the previous or next chunk from the other processes. An illustration can be seen in figure 2. To solve this problem, depending on which process handles which chunk, the blur filter function will receive an array of overlapping pixel of the top, bottom or both parts of the image chunk as shown in figure 2 where the blue lines in the image

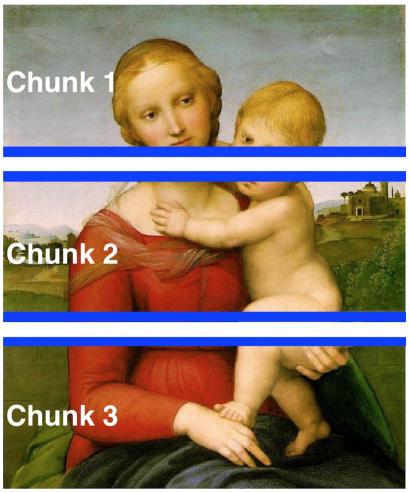

Fig. 2: An illustration of how the image is divided into chunks, in this case three chunks because the number of processes used is three. The blue area in the respective image chunks shows overlapping pixels that are needed in order to successfully calculate the pixels at the top or bottom part of a chunk.

indicates the overlapping pixels that are needed to be sent or received with MPI.

After that, the blur filter function was applied to each chunk and finally the root process gathers the results and render the output file.

B. Threshold filter

In this filter, there will be no need to send and receive chunks from other processes as overlapping pixels won't be necessary. The image was as previously, divided into chunks of y-size y/p where y is the vertical image size of the image and p is the total amount of processes available for use.

The threshold filter works by computing the average intensity of the whole image which is then used as a threshold value. This is done by using *MPI_Allreduce* which is used to compute the total sum of pixels in a chunk and after that compute the average by dividing with the image size. The mean value is then distributed to all the other processes.

The mean value will be the threshold value and the resulting image will be black and white depending on the threshold value. A pixel value that is greater than the threshold value will be black and a pixel value smaller than the threshold value will be white.

C. Result

The result section is divided into two subsections. The first section is Performance results where the filters have been applied with different number of processors or problem size to measure scalability and performance speed. The last section covers the resulting images after applying the filter.

1) Performance results: The following result is for image 1 which is the smallest image of size 676763 for blur filter and threshold filter:

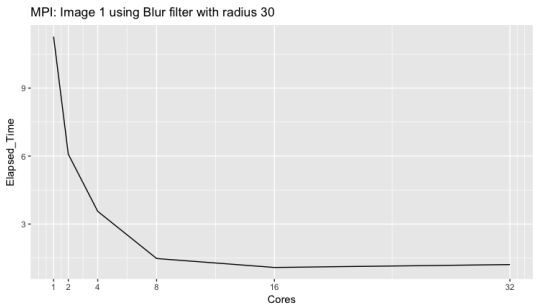

Fig. 3: Linear plot showing dependence between number of processors and elapsed time for image 1.

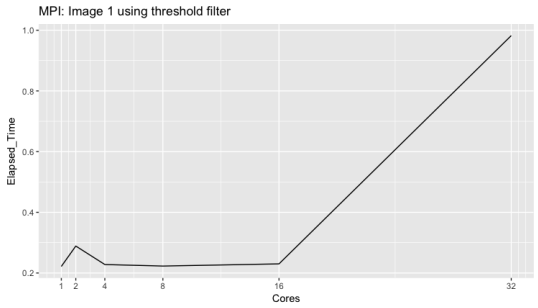

Fig. 4: Linear plot showing dependence between number of processors and elapsed time for image 1.

The following result is for image 4 which is the largest image of size 3000x3000 for blur filter and threshold filter:

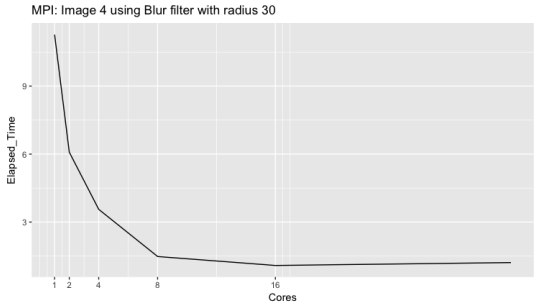

Fig. 5: Linear plot showing dependence between number of processors and elapsed time for image 4.

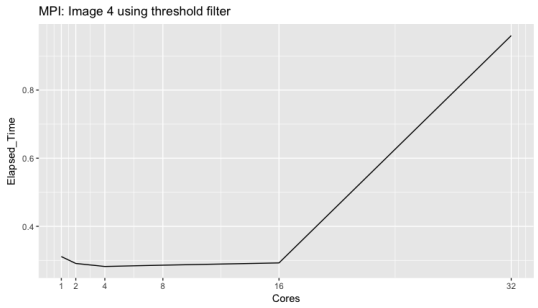

Fig. 6: Linear plot showing dependence between number of processors and elapsed time for image 4.

2) Image results after applying filter: Blur filter can be seen in figure 7:

Threshold filter can be seen in figure 8:

III. Lab 2 (Pthreads)

In the second lab, the same images and filters done in the previous lab had to be written with

Fig. 7: Blur filter with radius value of 30 and number of processors = 4.

Fig. 8: Each pixel in the image is set to white or black depending on the pixel value is above or below the mean image pixel value.

pthreads. The same approach had been done in lab 2 where the image is divided into chunks based on the number of available processors. After that, each thread will operate on their own individual part in which the filters will be applied concurrently.

A. Blur filter

The implementation of blur filter with *pthreads* is done in a different way compared to the implementation with *MPI*. Each thread will work different parts of the image and then blur filter is applied to each part. However, there is no need to send and receive overlapping parts of the image in order to compute the average pixel value with a given radius because each thread already has access to the whole image. Therefore, the only thing each thread has to worry about is if tries to operate outside of the image boundary.

B. Threshold filter

For threshold filter, each thread will compute the local sum of pixels in their respective part of the image. After that, the average intensity value of the whole image is computed then each thread will apply the threshold filter on their respective part of the image.

C. Result

The result section is divided into two subsections. The first section is Performance results where the filters have been applied with different number of processors or problem size to measure scalability and performance speed. The last section covers the resulting images after applying the filter.

1) Performance results: The following result is for image 1 which is the smallest image of size 676763 for blur filter and threshold filter:

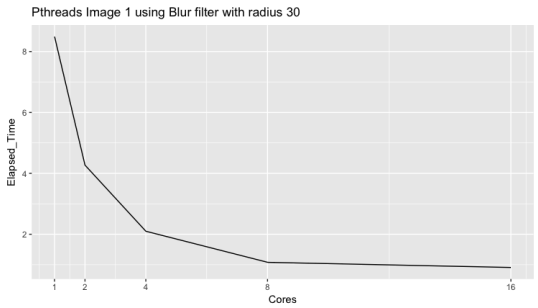

Fig. 9: Linear plot showing dependence between number of processors and elapsed time for image 1 using MPI.

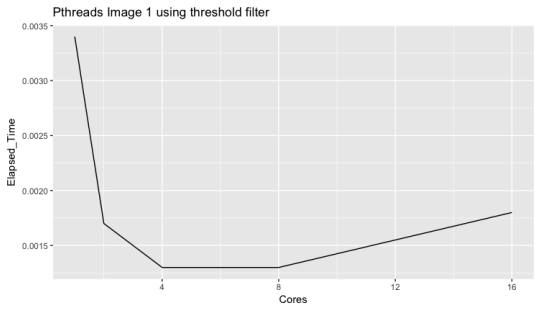

Fig. 10: Linear plot showing dependence between number of processors and elapsed time for image 1 using MPI.

The following result is for image 4 which is the largest image of size 3000x3000 for blur filter and threshold filter:

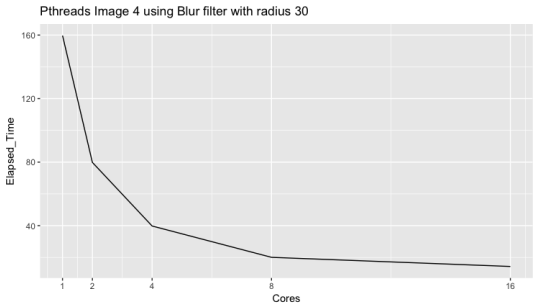

Fig. 11: Linear plot showing dependence between number of processors and elapsed time using pthreads.

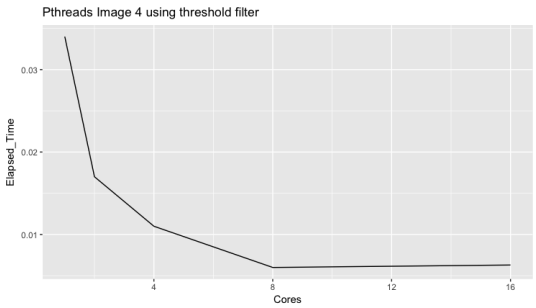

Fig. 12: Linear plot showing dependence between number of processors and elapsed time for image 4 using pthreads.

2) Image results after applying filter: Blur filter can be seen in figure 7 Threshold filter can be seen in figure 8.

IV. Discussion

A. MPI

In the results of blur filter we clearly see a good scaling factor even up to 32 processes on both images. But we see the exact opposite for the threshold filter, and elapsed time increases dramatically from 16 to 32 processes. The reason for this is the very different computations needed for thresholding compared to blurring. Blur filter requires much more computations than threshold filter but they roughly do the same amount of communication. So for the case of threshold filter the communication costs much more than the computation, and when we use 32 processes it also needs to communicate over two separate nodes/computers, further increasing the cost of communication.

B. Pthreads

For pthreads we see a similar results as MPI, blur filter scales very well, almost completely linearly. But threshold filter scales negatively after four cores. The cost of creating threads exceeds the cost of dividing the computation.