INF-3201 Parallel programming

Assignment 2 - Parallel Raster Image Filtering using MPI and OpenMP

Magnus Dahl-Hansen

October 2025

1 Introduction

The goal for this assignment was to implement a sequential and parallel algorithm that could apply an Emboss and Sobel filter on an image. The sequential version were to apply the filters using only one process, while the parallel version should make use of several processes using openMP and run on different nodes in a network. This report describes how the different versions was implemented and their performance in detail.

2 Sequential Solution

The sequential version starts by loading an entire image into memory using the provided helper functions. The image is then processed by two very similar functions that apply either the Sobel or the Emboss filter. The main difference between these two functions lies in the kernels they use. A kernel is a 3x3 matrix of numbers that is used to modify the red, green and blue (RGB) values of a pixel. To apply a filter, the values of each pixels surrounding neighbors are multiplied with the corresponding values in the kernel and summed together. The result value is then used to replace the value of the original pixel, effectively producing any kind of filter.

To apply the Sobel and Emboss filters, an image first have to be converted to grayscale. This is done by using a helper function. An output buffer of the same size as the image is then allocated to store the filtered values. The filtering process is performed by using two nested loops - one iterating over the rows of the image, the other iterating over the columns of the image. This way detection of the images edges is made easy by checking if the row or columns are equal to zero or the width/height of the image. Since the edge pixels do not have neighbor values on all sides, the correct Emboss and Sobel filter values cannot be calculated. Their RGB values are therefore set to zero, creating a

small black edge around the image.

For pixels that are not at the edge, the neighboring pixel values are retrieved and multiplied by the corresponding kernel values. In contrast to the Emboss filter, the Sobel filter requires two kernels. So, in the Emboss version the calculated values are summed together and stored in the output buffer, while in the Sobel version, the calculated values for two kernels are combined before being placed in the output buffer. For both filters, a computed pixel value may exceed the maximum intensity level of 255. If this happens, it is clamped to 255.

After both filter values are computed, helper functions are used to write the filtered pixels into a buffer and create new BMP image files, containing the results.

3 Parallel Solution

In the parallel version, multiple processes are used to apply the filters on the image. Here, process zero acts as a *master process* and is responsible for dividing the image into tiles, distribute these tiles to worker processes, receive filtered tiles and assemble them back together to form the final image. Before dividing and distributing the tiles, the master process loads the image into a buffer and converts it to grayscale. When dividing the image, it is not always possible to split it into even tiles. In some cases excess rows remain. To handle this, the master process calculates both the number of tiles each process should receive and the number of excess rows. The first worker process will therefore receive and handle the excess rows including the normal number of rows each worker process should have.

As mentioned earlier, applying filters requires access to neighboring pixels. To ensure that each worker can apply the filters correctly, "ghost" rows and columns are added to the tiles. Depending on which tile number is sent, it will either contain zero padded pixels on edges or copies of pixel values from image rows in the middle of the image. These rows are only used for calculations and will not be part of the actual output they send back. In addition to the pixel data, each worker also receives an information message about each tile, containing its width, height, total number of pixels and the number of padded pixels. This information is later used to help apply the filters.

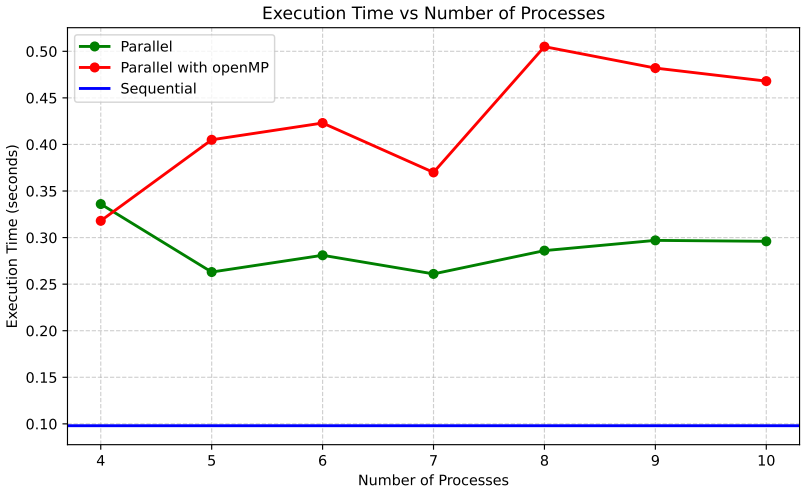
When a worker receives the pixel data and the metadata, it allocates memory for two output buffers - one for Sobel pixels and one for Emboss pixels. It proceeds to apply the Sobel and Emboss filters in the same way as the sequential version, but does not compute values for the ghost rows. After the filters are applied, they are sent back to the master process along with a tag that identifies which kind of filter is applied to each tile.

The master process initiates a loop that continuously receives tiles from the workers until both images are assembled back together. Using MPI's built-in functions, the master process is able to fetch the tag and the size of the message, allowing it to correctly place the received pixels into the appropriate position in the final image buffer. Placing the received pixels into the final buffer by using the tag and message size is necessary because messages may arrive out of order, so the tiles cannot simply be put back together in a sequential way. For each tile the master receives, a counter will increment. If this counter matches the number of tiles that were sent for each filter, the loop will terminate. Finally, the master process writes the assembled image and creates new BMP files with the filters applied.

4 Time Performance analysis

The different implementations and versions have been tested and their execution time measured. The runtime for the sequential version was measured once, while the parallel versions were tested using between 4 and 10 processes, both with and without multi thread processing. Measurements for 1 to 3 processes were excluded because one process would only consist of a master process, which does not apply filters. For processes 2 and 3, errors occured and the program could not complete successfully, which will be discussed in the *discussion* section.

The graph below presents the execution times in seconds on the y-axis and the number of processes on the x-axis. The sequential version, which does not use multiple processes are shown as a flat blue line for reference.



From the graph its clear that both the parallel version and the parallel version with openMP are significantly slower than the sequential version. This is to be expected since the size of the input image is relatively small. The overhead of dividing the image into tiles, creating metadata, sending data between processes and reassembling them is considered to much work for a simple task like this. with an image width of 737 pixels and a height of 480, the total number of pixels are roughly 353000. When divided between 4-10 processes, each process handles only 35,000-88000 pixels, which is a very small workload. A result of this is that the worker processes finishes their work quickly and the main run time is dominated by communication between processes.

The parallel version without openMP (shown in green) also shows that this solution does not scale well for this workload. The performance stabilizes around 5 to 7 processes and looks like it worsens with 8 or more processes, showing that communication overhead between processes takes over and reduces the efficiency of the program.

The parallel version using openMP (shown in red) is performing even worse than the parallel version not using it. This is surprising cause one would expect multi-threading to improve performance. The cause of this is probably poor usage of openMP in the implementation and/or "thread contention" - meaning when multiple threads run in parallel they all want to access the same memory, but they can not do so at the same time. Therefore some threads are forced to wait to access some memory and as a result they wil run even slower compared to if there were no other threads. Another cause could be that the use of multiple threads are only used in loops. The workload for each thread might therefore not be divided equally and some threads will have more work than others. This combined with poor implementation and thread contention could significantly reduce performance. In summary, it looks like the workload is too small to benefit from parallelization.

5 Theoretical maximum speedup using Amdahl's law

Amdahl's law is a principle used in parallel programming to measure the theoretical maximum speedup of a program when a given portion of the program can be parallelized. It provides a way to estimate the maximum improvement in performance when parts of a task are run in parallel while other parts remain sequential. Amdahl's law can be expressed in the following way:

$$S = \frac{1}{(1-P)+\frac{P}{N}}$$

Where:

- "S" is the potential speedup relative to a sequential version of the same program.
- "P" is how much of the program that can be parallelized
- "N" is the number of processes that can run in parallel

In this implementation the master process is the only process that does not run in parallel. So, to estimate how much of the program can be run in parallel I measured both the total runtime of all worker processes and the master process. When running the program with 5 processes applying filters to the image tiles I measured that their total work time was 0.177 seconds, and the master process worked 0.269 seconds. Based on these measurements, approximately 60.4% of the time was used by the master process while 39.6% of the time was used by workers. Applying these numbers to Amdahl's law tells us that the maximum achievable speedup of this version 1,46x.

To further test potential improvements I estimated the potential improvement if the master process also applied filters to the pixels. The total measured run time for the current implementation is 0.448 seconds, where the master worked 0.269 seconds and the workers together worked 0.177 seconds. Since each worker approximately contributed with $0.177/5 = 0.0354$ seconds, allowing the master to process one additional tile would roughly bring the total parallel work from 0.177 to 0.212 seconds. With these new numbers the parallel work increase from 39.6% to 47.5% and a potential maximum speedup would be 1.65x faster resulting in a 13% increase, which I believe is a solid improvement.

6 Discussion

During testing, some errors occured when running the parallel versions using 2,3 and 4 processes. In some cases the program completed successfully, while in other cases it produced the following error message:

```
--------------------------------------------------------------
A process or daemon was unable to complete a TCP connection
to another process:
  Local host:    c9-1
  Remote host:   c7-20
This is usually caused by a firewall on the remote host. Please
check that any firewall (e.g., iptables) has been disabled and
try again.
--------------------------------------------------------------
```

What causes this error is currently unknown and it is particularly interesting that the error only occurs when running 2-4 processes, while running the pro-

gram with 5 or more processes completes with no errors. Even more interesting is the fact that even though the program is not able to exit normally it still produces a fully filtered and assembled output images. This behavior indicates that there might be some underlying issues related to MPI, but this has not been fully identified at this point.

During implementation I chose to create a version that did not use openMP, as I had no experience with this. Initially I assumed it would be as simple as adding the *"#pragma omp parallel for"* directive in front of loops to make the program run with multiple threads. However, this was not the case, because careful consideration of which variables to use and where to modify them was necessary to make it work properly. As a result, the openMP version is able to fully apply filters to the images, but will not assemble the final images in the correct way.

Regarding potential optimizations to my code, my initial goal was to implement a simple version of the parallel program, where the master process only were responsible for sending data and assembling tiles, while each worker process applied the filters. Once this version worked I planned to improve it gradually. However, this became more challenging than I expected so I was unable to include all the optimizations I had planned.

What I would have liked to add to improve my program was to allow the master process to also apply filters on tiles, create a system where worker processes could ask for more work once they were finished processing their tile, and making it possible to change tile sizes for each process. Lastly, the master and worker processes currently uses *MPI_SEND* and *MPI_RECV* to send and receive tiles, and I have learned that this causes blocks in communication, because messages can only be sent or received where they are expected to. I also learned that MPI provides non-blocking communication functions that makes it possible to let the master process receive tiles while still sending messages to other processes, effectively improving performance. This is something that could have significantly improved performance.

7 Summary

In this assignment a raster image filtering program was implemented, capable of applying a Sobel and Emboss filter in both a sequential and parallel way. In this implementation the parallel versions did not perform better than the sequential version. The likely reason for this is that the computational workload per process was too small compared to the communication overhead used to communicate between processes. The parallel version using only MPI was slightly better than the version using MPI and openMP, possibly because of thread contention. For the small input sizes used in this assignment the extra time spent on making processes work in parallel did not prove to be benefitial

compared to doing the work in a sequential way, further suggesting that the implementation is relying more on communication than actual computing.