

# TDDC78: Lab Report

Lab 1: MPI

| Name            | PIN         | Email                   |
|-----------------|-------------|-------------------------|
| Alexander Yngve | 930320-6651 | aleyn573@student.liu.se |
| Pål Kastman     | 851212-7575 | palka285@student.liu.se |

# Contents

|          |                            |          |
|----------|----------------------------|----------|
| <b>1</b> | <b>Introduction</b>        | <b>3</b> |
| 1.1      | Blur filter . . . . .      | 3        |
| 1.2      | Threshold filter . . . . . | 3        |
| <b>2</b> | <b>Our implementation</b>  | <b>3</b> |
| 2.1      | Blur filter . . . . .      | 3        |
| 2.2      | Threshold filter . . . . . | 4        |
| <b>3</b> | <b>Execution times</b>     | <b>4</b> |
| 3.1      | Blur filter . . . . .      | 4        |
| 3.2      | Threshold filter . . . . . | 6        |

## 1 Introduction

This lab consists of two image filters: blur filter and threshold filter.

The goal of the lab is to distribute the workload to tasks running on different processes with the help of MPI. The tasks won't have access to the same data, so this needs to be sent.

### 1.1 Blur filter

The blur filter uses a normal distribution together with a given radius to calculate the mean value for every pixel in an image, this will create a blur the given image.

### 1.2 Threshold filter

The threshold filter first calculates a mean value for every pixel in the image, it will then go through the image one more time and either set every pixel to black or white depending if the pixel value is over or under the calculated mean value.

## 2 Our implementation

This section will describe how we used MPI to parallelize the execution of the programs.

It might be worth mentioning that we use `MPI_Barrier` in both filters just for the sake of timing the filters, so that the timing starts and stops when all tasks are done with the filter.

### 2.1 Blur filter

For this filter we define an MPI type called `mpi_pixel_type` which contains a pixel, this will make it easier for us to send the data as we can send whole pixels instead of sending the r, b and g doubles of the pixel one by one.

We started by using `MPI_Send` and `MPI_Recv` to send parts of the image to the tasks. These parts were overlapping regions depending on how big the radius were, the size of them will also depend on how many tasks we have. This worked on our own laptops but unfortunately not on triolith for whatever reason we don't know. This made us instead use `MPI_Scatterv` and `MPI_Gatherv` as we had done for the threshold filter.

To be able to do this we must first broadcast the size in x and y, along with the radius, we do this with `MPI_Broadcast`.

When the tasks have received their data, they will first run the blurfilter and then send the data back, without the overlapping regions.

After sending the data back, the tasks free their memory and runs `MPI_Finalize`, which will kill them, except the first one which will save the result and free the memory before quitting.

## 2.2 Threshold filter

In this filter we also define the `mpi_pixel_type` just as for the blur filter. But because every pixel is just dependant on the actual value of the pixel, we won't need any overlap of the regions, and thus the `MPI_Scatterv` and `MPI_Gatherv` are instead used here.

We first broadcast the sizes in `y` and `x` along with the radius just as in the previous filter, then we send the regions to the different tasks and let them calculate the average value of their areas. When all tasks are done with this, we run `MPI_Allgather` which will make all tasks send their average value to all other tasks. So now all the tasks will calculate the average value independantly and then run the filter with this value.

When done, all tasks will send their data back to the first task and run `MPI_Finalize`. The first task will save the result and free memory before quitting, just as in the previous filter.

## 3 Execution times

In this section we will present the execution times of the filters.

It is worth mentioning that the times for all graphs below are in seconds.

The tests were performed on four images with sizes as can be seen in 1.

**Table 1** – The sizes for the images used in the tests.

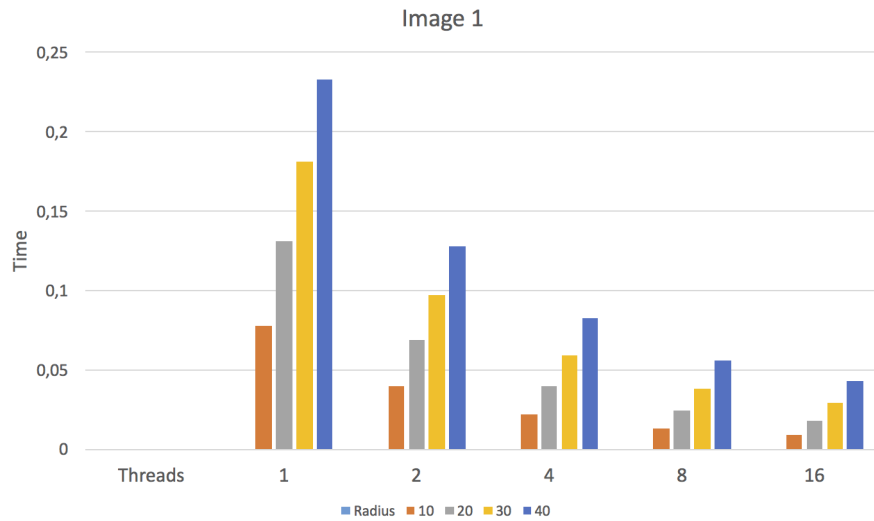
| Image name | x pixels | y pixels |
|------------|----------|----------|
| im1.ppm    | 676      | 763      |
| im2.ppm    | 1024     | 1024     |
| im3.ppm    | 1600     | 1200     |
| im4.ppm    | 3000     | 3000     |

### 3.1 Blur filter

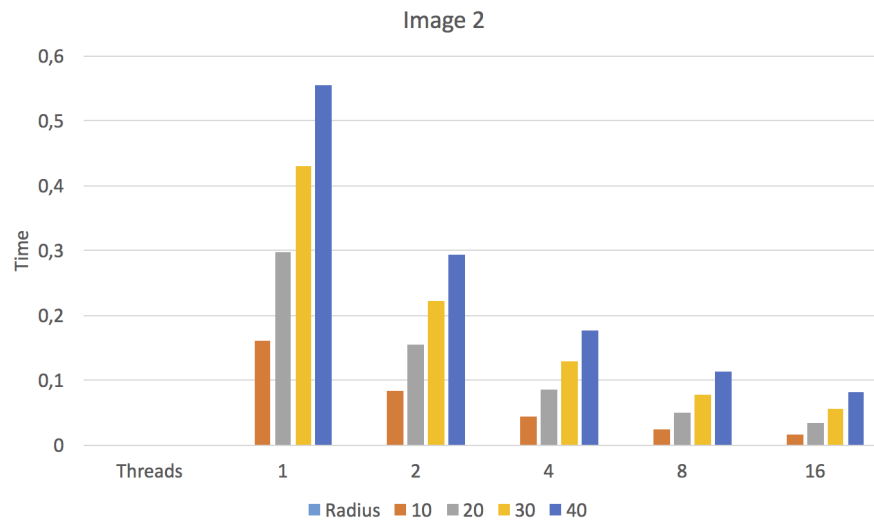
Here we saw that with an increase in the number of threads we get a decrease in time, which we wanted to attain.

Our result show that if we increase the number of threads, we will get a decrease in time. But this relationship is not linear due to deminishing returns, since the problem is not perfectly parallell the serial section sections start to dominate the execution time.

The results for image 1 and image 2 can be seen in figure 1 and figure 2 respectively.

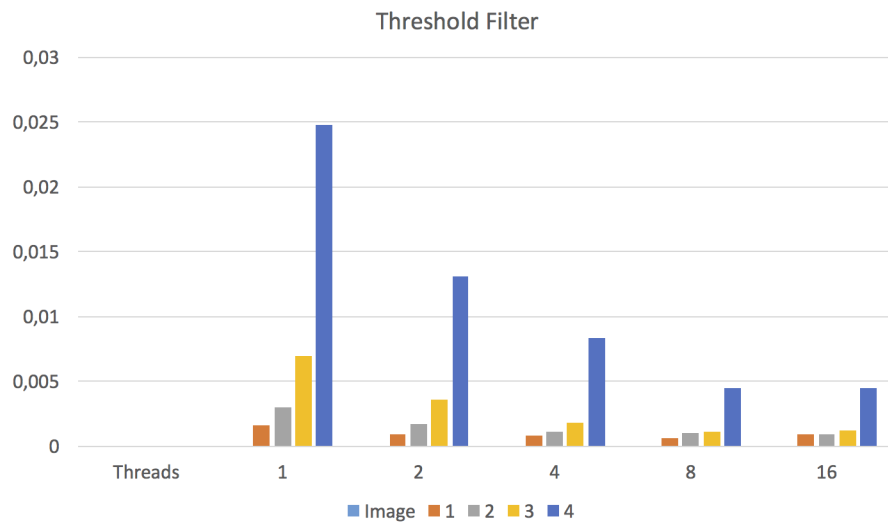


**Figure 1** – Result for the blurfilter run on image 1.



**Figure 2** – Result for the blurfilter run on image 2.

### 3.2 Threshold filter



**Figure 3** – Result for the thresholdfilter.