TDDC78 Lab 1-2

Anton Sterner
antst719@student.liu.se

Leo Nordling
leono184@student.liu.se

Linköping University

May 24, 2018

Chapter 1

Lab 1 - MPI

1.1 Blurfilter

1.1.1 Description of program

The program reads an image and applies a blur effect on the image. The blur effect is calculated by taking the average pixel value in a NxN window around every pixel. Every pixel in a NxN window affect the middle pixel by a factor depending on its distance to the center pixel. The factor is determined by a matrix of Gaussian weights. This means that to calculate the new weights the program has to visit a NxN square window around every pixel which will be executed in O(n*N*N) time. Where n is the amount of pixels in an image and N is the radius of every square to the averaged pixel value. In this implementation, blurring is applied only row-wise to all image partitions. This has the benefit of not having to use any extra padding in the image partitions. Instead, after the first row-wise blurring, we transpose the image, switching rows and columns, and repeat the blurring on the new rows.

The program code is based on the original serialized sample files provided, although heavily modified. A custom MPI data type is created which contain pixel data, one unsigned char for each value r,g and b. This is required, since the MPI functions only use their own specific data types. The Gaussian weight matrix which is used in the blurring of the image, is computed by the function `get_gauss_weights` provided in the lab. The first process reads the input image and partition it to each available process. All partitions contain only full rows of the original image, the image is never split in the middle of a row. The displacement of memory addresses are calculated at the same time. All partitions are not always equally large, the size of all partitions are saved in an array. The image partitions are scattered to all available processes using `MPI_Scatterv`, and saved to a temporary buffer. Blurring is applied to all partitions, and then gathered and transposed. These steps are then repeated, to apply blurring to the transposed image.

1.1.2 Execution times

Blurring scales well with increased number of processes, with an almost linear decline in execution time when increasing from 1 to 4 processes. When increasing the number of processes further, the execution time decreases, but not quite halving in each step, see figure 1.1 and 1.2 for results on two different images. The overhead cost of communication increases for each additional process, so there must be a limit to where further parallelization is not worth it, but we are not quite there yet with this program.

When increasing the blur radius from 10 to 50 pixels in steps of 10, the speed scales linearly, as can be seen in figure 1.3. Because of the parallelization, the amount of additional work required scales linearly in each process, not quadratic as it would if the calculation were serialized.

1.2 Thresholding

1.2.1 Description of program

The threshold filter calculates a sum of all pixel values and divides it with the amount of pixels. It then compares every pixel with the calculated sum. Pixels that has a value above the calculated sum is set to white and pixels below is set to black. This means that the program need to visit every pixel twice and therefore have an execution time of O(n), where n is the amount of pixels.

The MPI paralleled version uses the `MPI_Scatterv` method to distribute the image equally over all the processors. Initially the first processor reads the image and calculates how many pixels each processor will handle. The dimensions along with displacements are then broadcast to all processors. The first processor then scatters the image evenly to all processors and the image partitions are stored in image buffers. Each processor then calculates the sum of all pixel values in its own image buffer. With the use of `MPI_AllReduce` the sums are reduced to a sum of all the sums from every partition. Each processor then divides the new sum by the amount of pixels in the whole image. The new value becomes the threshold value that each processor uses to filter its own partition of the image with. After the filtering process is done, the partitions are gathered in the first processor that finally writes the whole filtered image to a file. Since reading the image is done on a single processor the execution time will still be O(n). However the filtering is done evenly on all processors which means that the filtering will be executed in O(n/p) time, where p is the amount of processors.

each process can apply the filtering process on its image partitions.

1.2.2 Execution times

Applying a threshold to an image is a simple process, and does not scale well with increasing the number of processes when using small images. The overhead cost of communication becomes more expensive than the actual computational cost when increasing the number of processes. As shown in figure 1.4, using more than 4 processes makes the program perform worse. For larger images, the program still scales fairly well when using more processes, see figure 1.5.

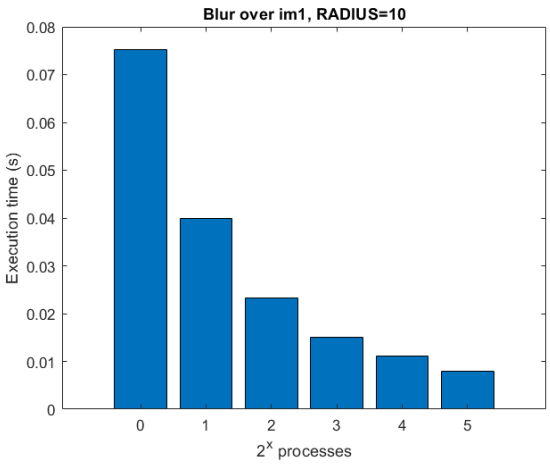

Figure 1.1: MPI blur on im1.

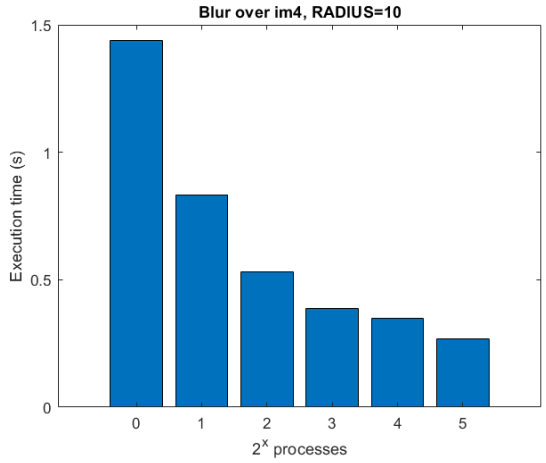

Figure 1.2: MPI blur on im4.

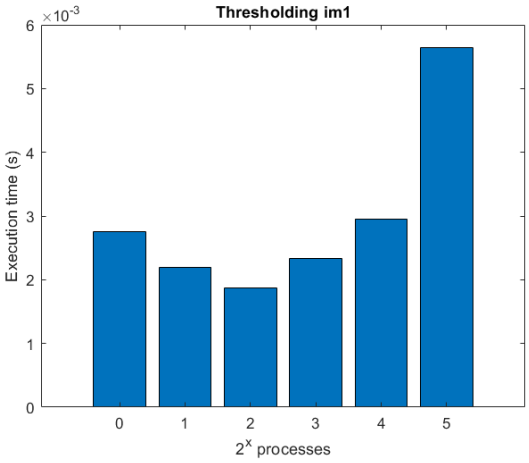

Figure 1.4: MPI threshold on im4.

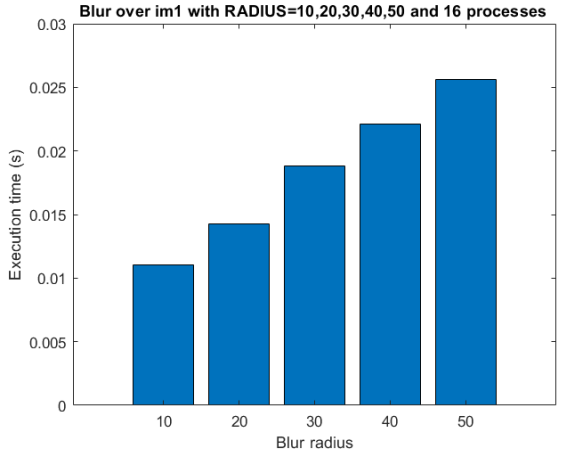

Figure 1.3: MPI blur on im1 with varying blur radius.

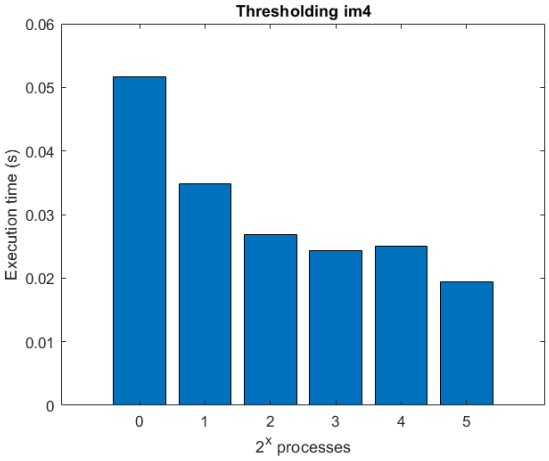

Figure 1.5: MPI threshold on im4.

Chapter 2

Lab 2 - Threading

2.1 Blurfilter

2.1.1 Description of program

All data relevant to the blur computations are saved together in a struct `arg_struct`. Threads are only created once. For each thread, the function `myThreadFun` is applied to the segmented data. The partitioning of the image data is done the same way as in the MPI implementation. Blurring is first applied on the rows of the image, writing the result to a temporary buffer. To avoid read/write conflicts, a thread barrier is used here. Blurring is then applied on the columns, writing the result to the original image buffer. The threads are then joined, and the program ends.

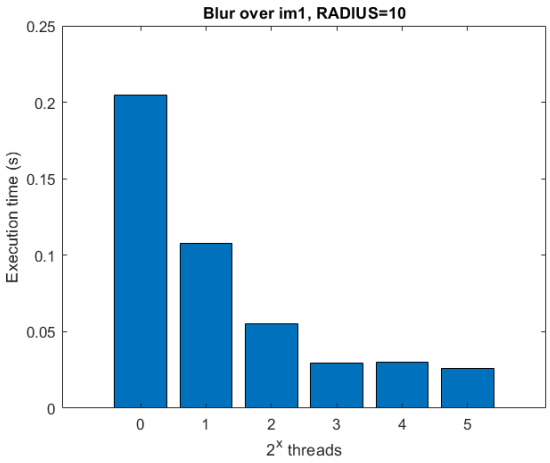

Figure 2.1: pthreads blur on im1.

2.1.2 Execution times

Blurring scales well with increasing the number of threads, but with only small performance gains when using more than 8 threads, see figure 2.1 and 2.2. The speed increase is canceled out by the additional cost of communication. Increasing the blur radius scales more or less linearly, as can be seen in figure 2.3.

2.2 Thresholding

2.2.1 Description of program

This implementation is very similar to the blurring version, the main difference is in the `myThreadFun` function. The calculations made to find the threshold are the same as the MPI version. When summing the threshold, a mutex lock is used for writing to a single variable. This can be optimized by instead keeping the partial sums in an array, and then sum the array elements, removing the need of a mutex lock.

2.2.2 Execution times

Thresholding the small image `im1` does not scale well, with execution times remaining constant or increasing with additional threads, see figure 2.4. Performance for larger images only improves up to 4 threads, with constant execution times when using 4-32 threads, see figure 2.5.

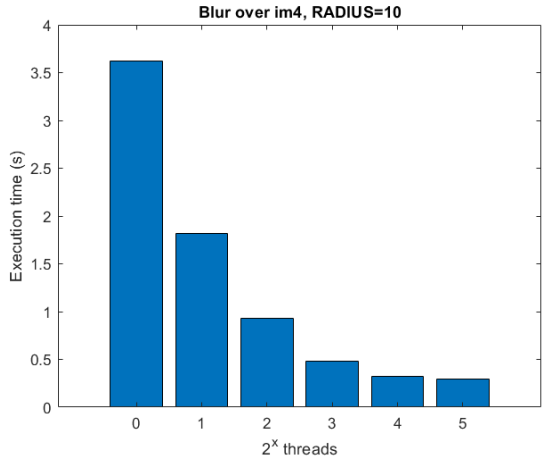

Figure 2.2: pthreads blur on im4.

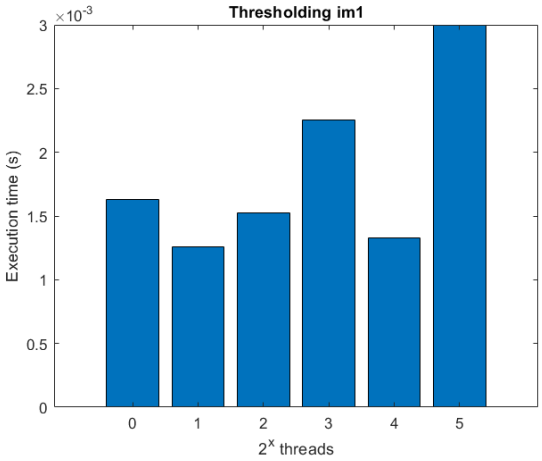

Figure 2.4: pthreads threshold on im1.

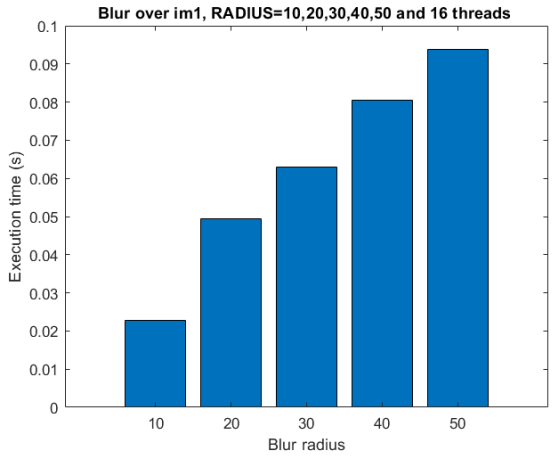

Figure 2.3: pthreads blur on im1.

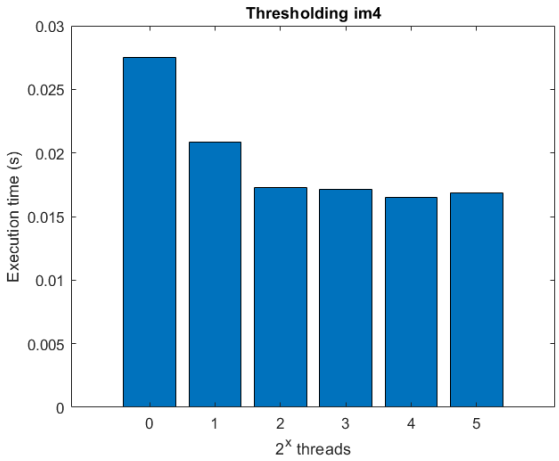

Figure 2.5: pthreads threshold on im4.