IMAGE FILTERS WITH MPI AND PTHREADS

Santiago Pagola

Hao-Hsiang Liao

Version 1.1

PROJECT IDENTITY

04/22/2018, ISY, LiU

Participants of the group

Name	Phone	E-mail
Hao-Hsiang Liao	0734845666	haoli436@student.liu.se
Santiago Pagola	0727239400	sanpa993@student.liu.se

Table of Contents

1 FIRST LAB – BLURRY FILTER

1.1 Question

The purpose of this first assignment is to make a blurry filter with a parallel implementation by using MPI and POSIX Threads.

1.2 Solution for MPI

First of all, we assume that an image has M rows and M columns, and we have N cores to process the image. We assign M/N rows of the image to each core for processing. Fig. 1-1 is an example where M/N is 3, so each core processes three rows of data. This example shows that every core can process different parts of the same image in parallel.

After that, we need to process vertical pixel columns. To make this problem easier, we convert Fig. 1-1 to Fig. 1-2 by calculating the adjoint matrix of the original image array. By doing so, we can get the data array shown in Fig. 1-3. Then, we apply the same horizontal filter again to process horizontal data in the image shown in Fig. 1-2.

Finally, we transpose back the calculated adjoint matrix and convert it back to a pixel array in order for the root process to write the image to disk.

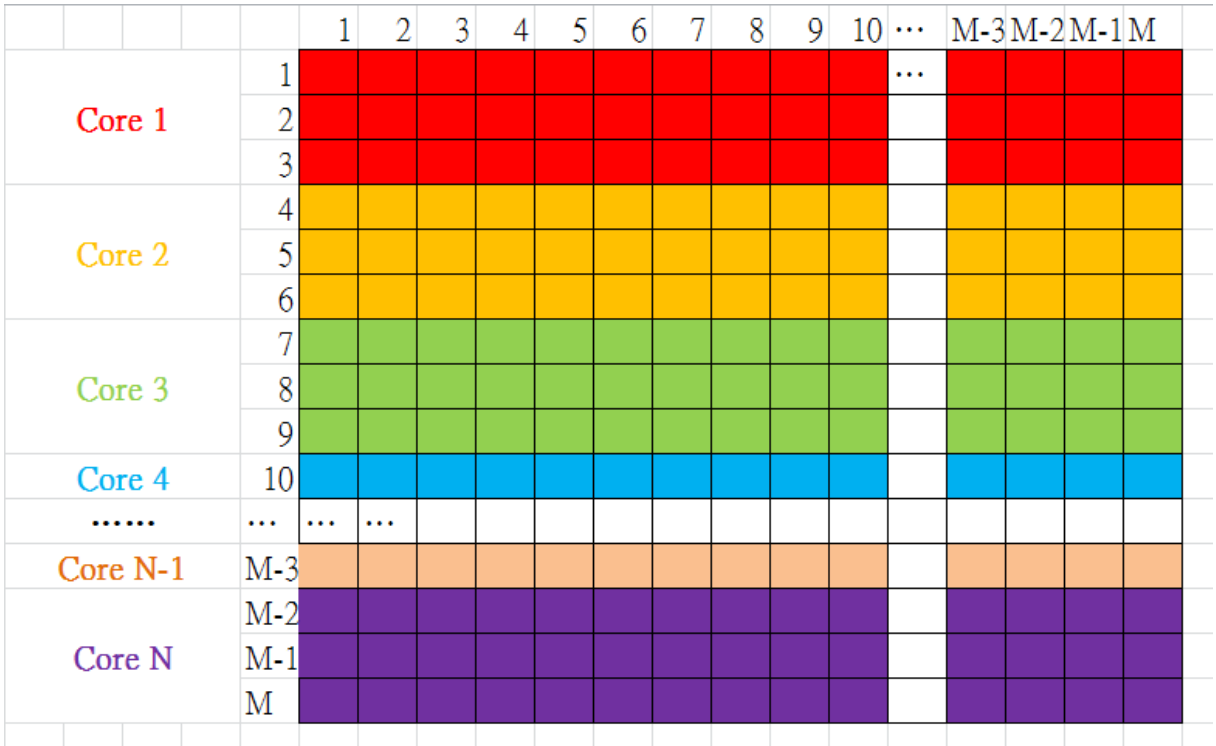

Figure 1-1 An example of horizontal processing.

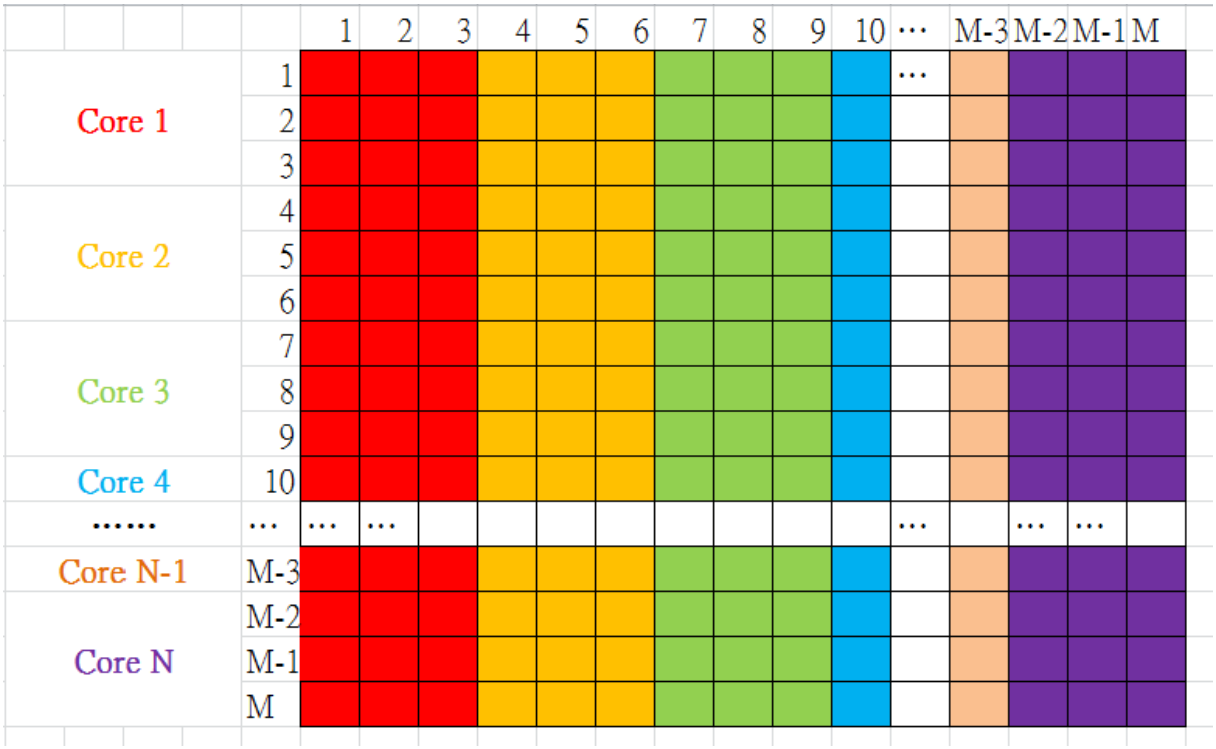

Figure 1-2 An example of horizontal processing

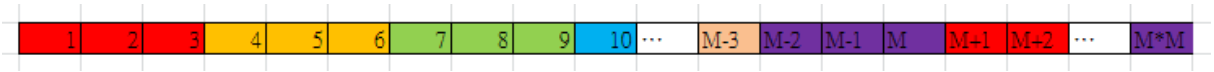

Figure 1-3 The arrangement of the image in the platform of MPI

1.3 Results for MPI

Fig. 1-4 demonstrates execution time for different number of cores. We can find that while the number of cores is increasing, then execution time decreases. We use 2^k cores, with k = 0, 1, 2, 3, 4, 5.

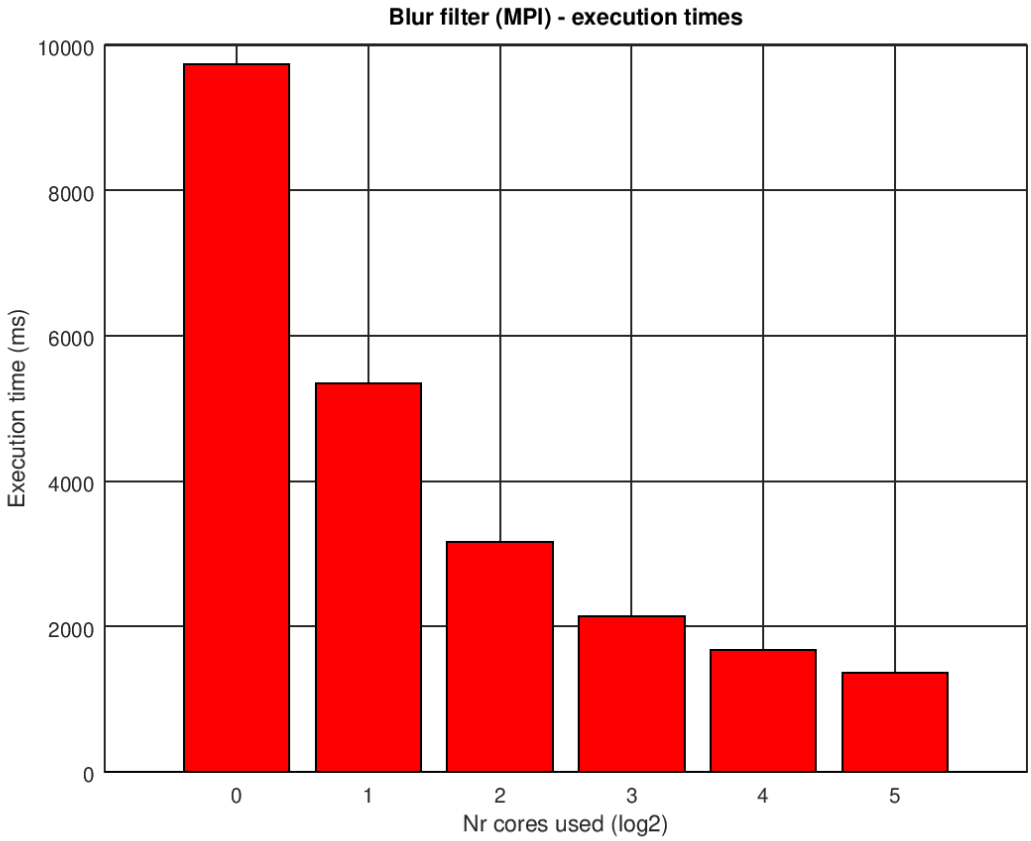

Figure 1-4 Execution time for different number of cores in exponential unit using the largest image, i.e., im4.ppm

1.4 Solution for PThreads

To process the image via PThreads, we split the filter into two parts, a blurfilter_x and a blurfilter_y, to process horizontal data and vertical data. In order for each thread to know where to start processing its portion of the image, starting points in both x and y axis are provided to the thread start routines in a *tdata_t* structure, which contains, among other fields, the ID of each thread, a pointer to the image to process and these starting points.

We use a critical section to protect `printf()` to avoid race conditions on printouts to *stdout*. For example, assume that thread 0 finishes its task and it is writing its results with `printf()`. In the meantime, thread 5 finishes its task as well, so it may start writing to the standard output while thread 0 is doing the same thing, thus resulting in messy interleaved printouts. That's why we needed protect the critical section with mutexes to prevent collisions from happening.

To maximize the usage of PThreads, we divide the number of rows, i.e. *y_size,* by the number of threads so that every thread can process an analogous number of horizontal pixels. The same applies for the "vertical" filter, where each thread processes a group of pixel specified by the number of threads and *x_size*, that is, the number of columns.

1.5 Results for PThreads

Fig. 1-4 shows execution time for different number of threads. We can see that if we run the filter sequentially (i.e. with the main thread of the current process), then it is going to take a long time to execute. However, we can decrease time consumption via increase the number of threads. We use 2^k threads, with k = 0, 1, 2, 3, 4, 5. It is interesting to point out that since the execution of this program is going to run on a single Triolith node, where there are 16 cores, that means that we will have a maximum of 16 hardware threads available, so a number of threads above 16 will only add overhead to the execution. This can be seen in Fig 1-5, were the exeuction times when using 2^4 and 2^5 is almost the same (~800ms).

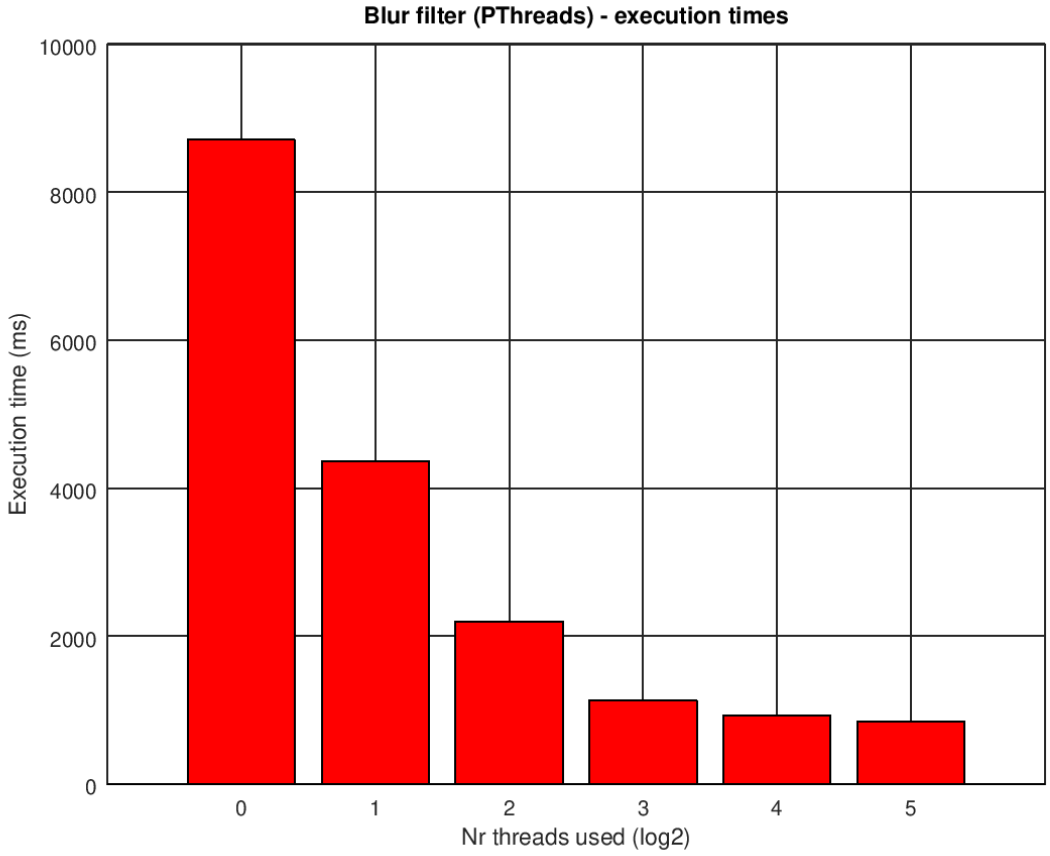

Figure 1-5 Execution time for different number of threads in exponential unit using the largest image, i.e., im4.ppm

2 SECOND LAB – THRESHOLDING FILTER

2.1 Question

The purpose of this first assignment is to make a thresholding filter with a parallel implementation by using MPI and POSIX Threads.

2.2 Solution for MPI

In this case, we split this task into two parts. First one is to sum every pixel in order to calculate an average, and the other subtask is to do the comparison between image pixels and the average.

Assume a picture has N pixels. To sum every pixel value, we apply the `MPI_Scatter` function to assign N/M pixels to each core, where M is the available number of cores. Then, we use `MPI_Reduce`[1] to sum these values up. By doing so we get the average, or threshold.

We then apply `MPI_Scatter` to send image data and average to all other processes. The input image pixels become white or black if they are lighter or darker than the threshold. After that, we apply `MPI_Gather` again to gather new pixel data and get the filtered image.

2.3 Results for MPI

Fig. 2-6 shows execution time for different number of cores. We can see that when the number of cores increases, then the execution time decreases. We use 2^k cores, with k = 0, 1, 2, 3, 4, 5.

[1]We could have used `MPI_AllReduce` in order to skip a third blocking `MPI_Send` to all non-root processes with the calculated average.

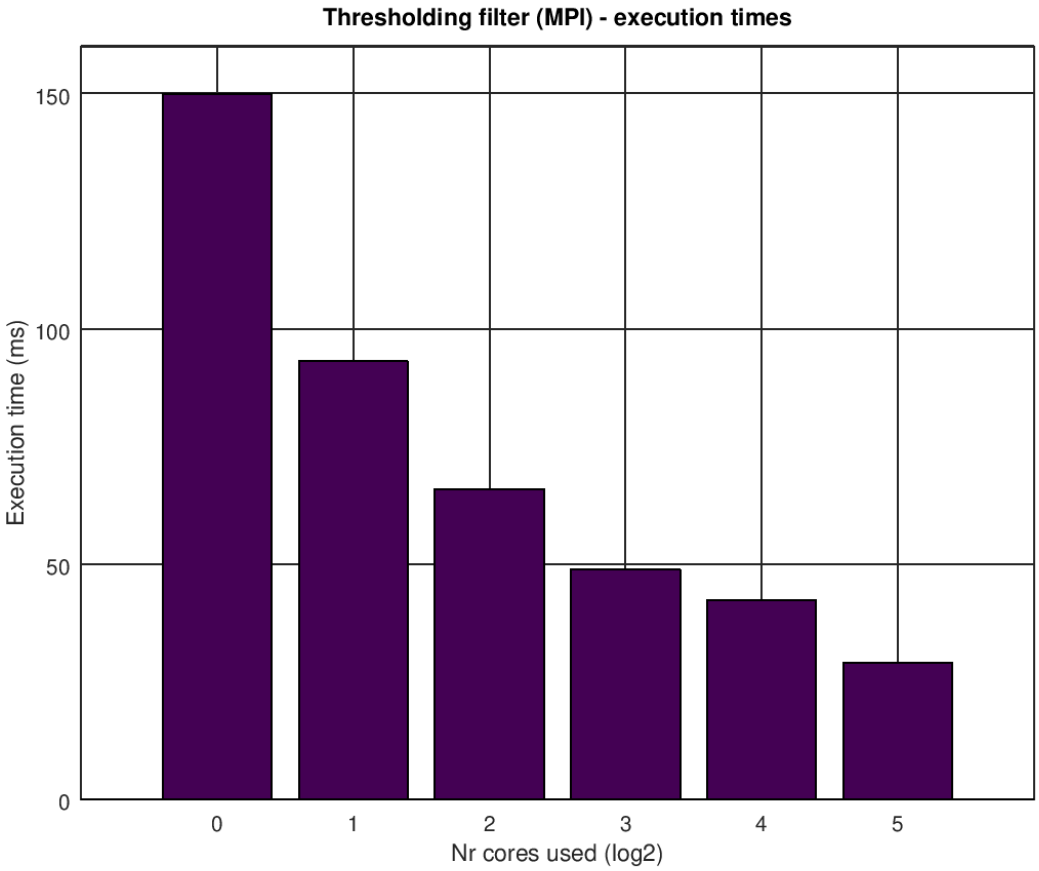

Figure 2-6 Execution time for different number of cores in exponential unit using the largest image, i.e., im4.ppm

2.4 Solution for PThreads

To run the program in parallel, we set some threads to compute partial sums. Assume *nr_elems* means the number of elements that will be processed by each thread, and the number of threads is *nr_threads*. Thus, we can know that the number of all image pixels = *nr_elems* * *nr_threads*. Besides, it is worth mentioning that we assume that thread 0, i.e. the main thread, or ROOT, won't be performing any filtering at all. It will only be responsible of creating the child threads and making sure they return to the main thread when calling `pthread_join` routine. Moreover, we use ROOT to calculate the average from the partial sums.

Similarly as in the blurry filter, we set a critical section to protect calls to `printf()` and *g_sum*, where *g_sum* is global sum. The reason that why we have to protect `printf()` has been discussed before, or Ch. 1.4. The variable *g_sum* is necessary to protect since its value would be wrong if two or more threads write their partial sum into *g_sum* at the same time. Thus, we need to lock the mutex to ensure that every thread writes its result into *g_sum* orderly.

After we get the average, we create threads again to filter their portion of the image.

2.5 Results for PThreads

Fig. 2-2 shows execution time for different number of threads. We can find that 2, 4, and 8 threads' execution times are similar.

We use 2^k threads, with k = 0, 1, 2, 3, 4, 5. As explained above, by using 32 threads we are only adding overhead to the execution, as it is shown in Fig 2-7.

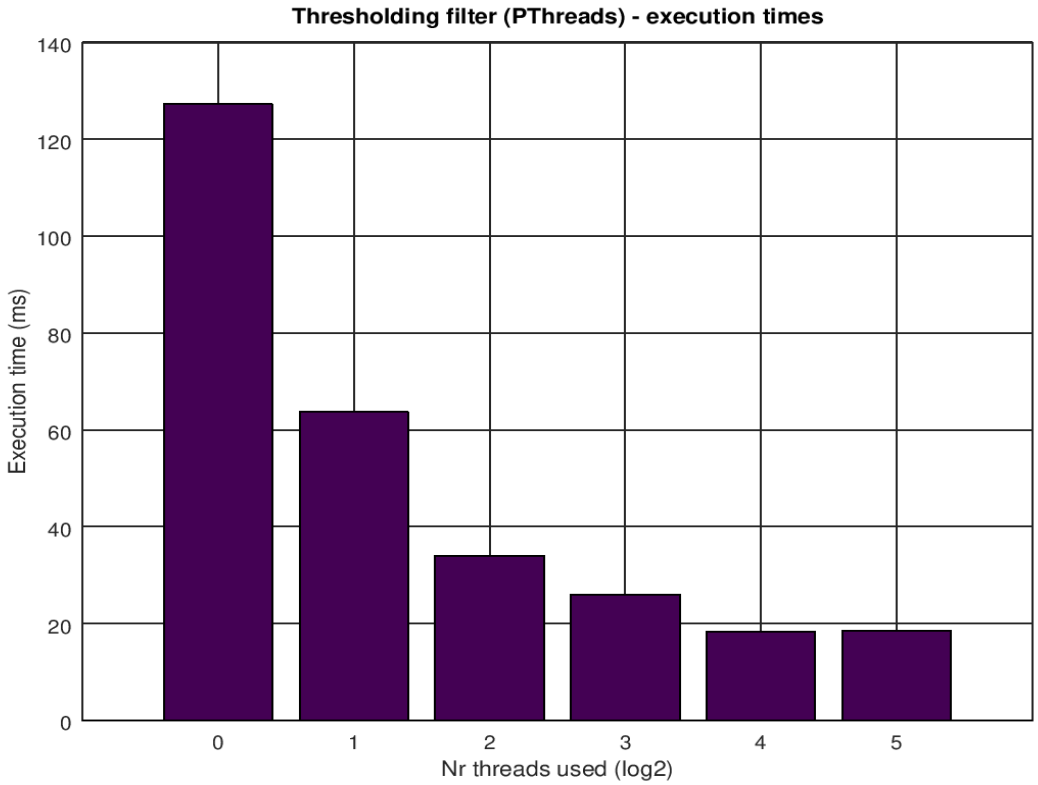

Figure 2-7 Execution time for different number of threads in exponential unit using the largest image, i.e., im4.ppm