LAB2: Image filter (MPI)

Date: 27/04/2016

Group A1:

Antonio Jimenez (antji996)
Martin Fontanet (marfo044)

The goal of the lab2 is to parallelize two filter (average filter and Thresholding Filter). In order to do it, we will use the *pthread* library.

1. **Description of your program and how you have parallelized it.**
   a. **Thresholding Filter:**
   The filter computes the average intensity of the image and the result is an image containing only black and white pixels. White for the pixels in the input image that are lighter than the threshold and black for the pixels in the input image that are darker than the threshold.
   To parallelize the program with the pthread library, we had to follow some steps, which are:

   -Create a structure *thread_data* which will contain the data for each threads (we can't directly pass arguments to the function of the thread, and so we have to do it with a pointer to a structure).
   -Create an array of size NUM_THREADS (the number of threads), which is an array of type *thread_data*.
   -Create an array of NUM_THREADS *pthreads*.
   -Split the work between the threads. In order to do it, we give a “beginning pixel” and an “ending pixel” on the picture to each thread.
   -Start all the threads.
   -Each thread will calculate the sum of every RGB values of the pixels he was given, which will be stored in a local sum (an attribute of the *thread_data* structure).
   -Add every local sum to the total sum and wait for all the threads to be done. This step has to be protected by a mutex.
   -Each thread calculates the average of the total sum.
   -Each thread checks if the pixel of the part of the picture he was given should be black or white and modifies it.
   -Join all the threads.

b. Averaging Filter:

The value for a pixel (x, y) in the output image is the normalized weighted sum of all the pixels in a rectangle in the input image centred around (x, y).

We have applied MPI to parallelize this program, here are the steps that we are following:

-Create a structure *thread_data* which will contain the data for each threads (we can't directly pass arguments to the function of the thread, and so we have to do it with a pointer to a structure).

-Create an array of size NUM_THREADS (the number of threads), which is an array of type *thread_data*.

-Create an array of NUM_THREADS *pthreads*.

-Split the work between the threads. In order to do it, we give a “beginning pixel” and an “ending pixel” on the picture to each thread.

-We set the weight array as static so every thread can access it.

-Start all the threads.

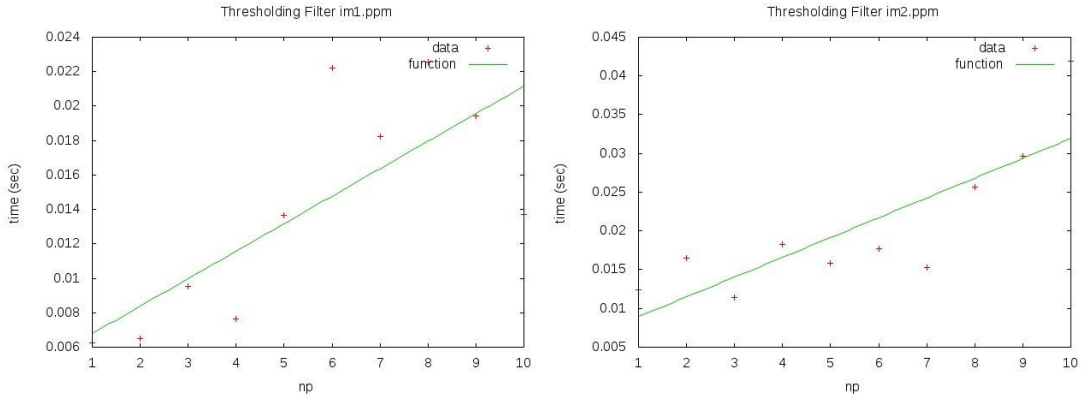
-Every thread calculates the new value of the pixels he was given with the array of weight and saves it in dst (which is a static array of pixels).

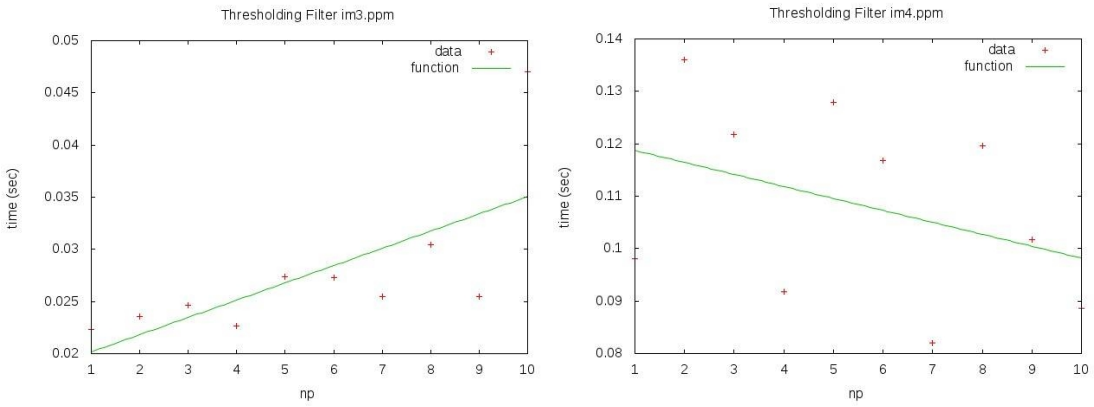
-Join all the threads.

2. Draw graphs that show the variations in the execution time.

In this step we do measurements by the execution the program using various number of processors and sizes in each filter:

a. Thresholding Filter:





Here you can see 4 graphs, on each one there is a different input size according to the number of pixels of every picture. And, on every graph, the time is analized with different numbers of threads.

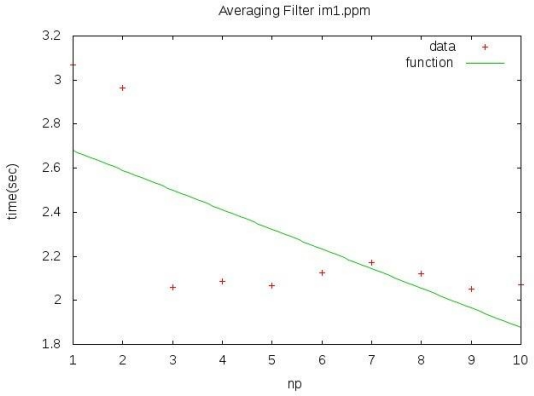
We can see that the optimal number of thread is between 3 and 4 for every test.

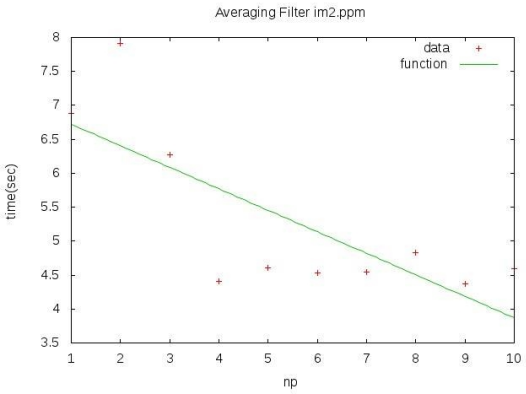
As in the first lab, we can see that if the size is smaller, then the necessary time is smaller too.

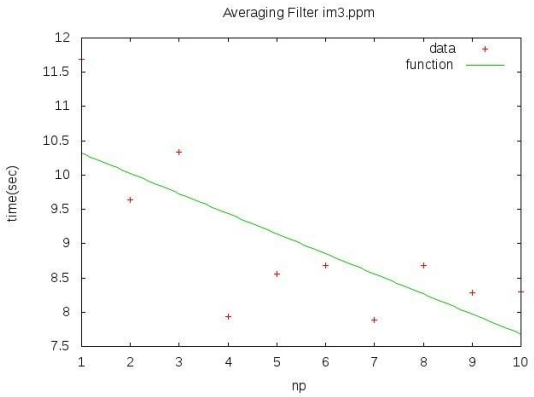
Also, we can check that, in every example, it needs more time when the number of threads is greater, because it needs more communication and has to wait.

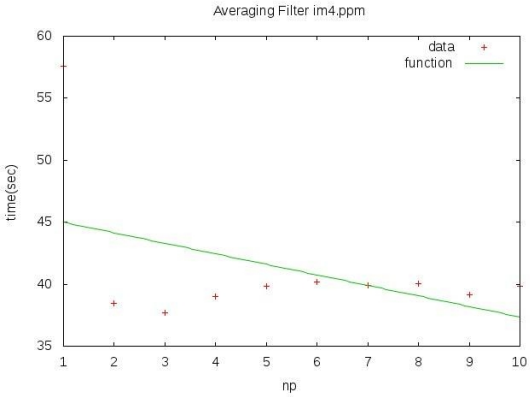
However, it's different for the *im4.ppm* picture. We can see that it's better to have a greater number of threads, because the picture is very large.

b. **Averaging Filter**:









Here you can see 4 graphs, on each one there is a different input size according to the number of pixels of every picture. And, on every graph, the time is analyzed with different numbers of threads and with a default radius of 100.

We can see that the optimal number of processors is either 3 or 4 for every test.

Again, we can see that it takes less time to apply the filter on a small picture than on a big one.

We can also see that the the time decreases when the number of threads increases.

3. Source-code of our program

a. Thresholding Filter:

You can find the code in the attached document.

b. Averaging Filter:

You can find the code in the attached document.

Build the project:

Compile: *make*

Run: *bash run.s*