LAB1: Image filter (MPI)

Date: 24/04/2016

Group A1:

Antonio Jimenez (antji996)
Martin Fontanet (marfo044)

The goal of the lab1 is to parallelize two filter (average filter and Thresholding Filter). We have to apply MPI in the algorithm of each filter to optimize it.

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

   -Define the new type MPI for the structure pixel.
   -Calculate the number of processors and the id of each processor.
   -Share the variable size by BroadCast.
   -Create a local array of pixels for every processor, called src_local.
   -We have the original array of pixels. Then we divide it between all the processors by Scatter with lsize (size/np) and save in src_local.
   -Every processors calculate the local_sum of all the pixels in the src_loca.
   -Calculate the TotalSum of the sum of local_Sum of each processor by AllReduce.
   -Each processor checks if the pixel is black or white in the own buffer. src_local.
   -Combine all the local_scr of each processor in the final array of pixel by Gather.

   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:

-Define the new type_MPI for the Pixel structure.

-Calculate number of processors and id of each processor.

-Share the variable ysize, xsize and radius by BroadCast.

-Share the array weight by BroadCast. All the processor access to it.

-Create 2 local buffer of pixel: src_local and dst_local for each processors.

-We divide the original array of pixel between all the processors by Scatter with lsize (xsize*ysize/np) and save it in the src_local.

-Every processor calculates the new value of each pixel with the array of weight and saves it in dst_local.

-Combine all the dst_local in dst for all the processors by ALLgather.

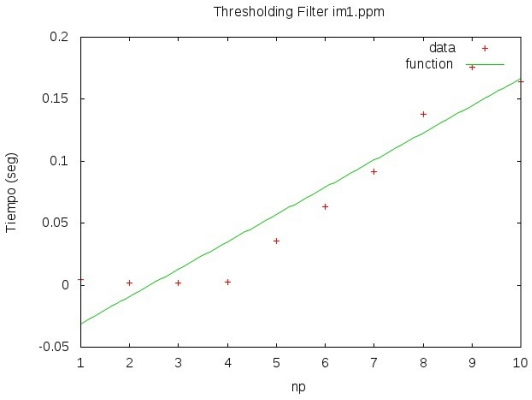
-Every processor calculates the new value of every pixel with the array of weight and save in src_local.

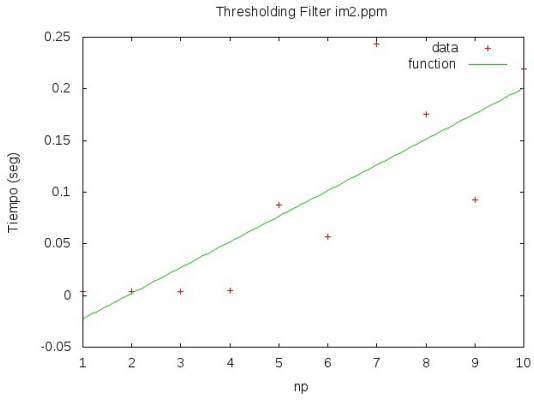
-Combine all the src_local of each processor in the final array of pixels by gather.

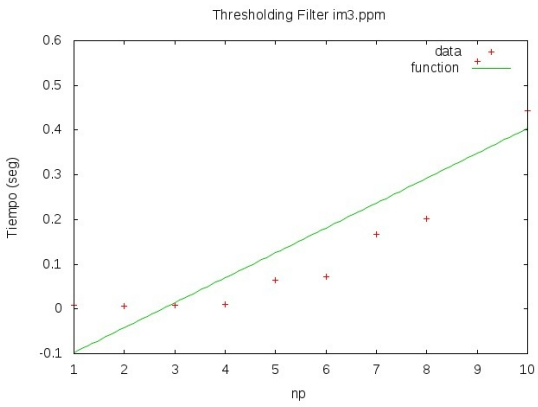
2. Draw graphs that show the variations in the execution time.

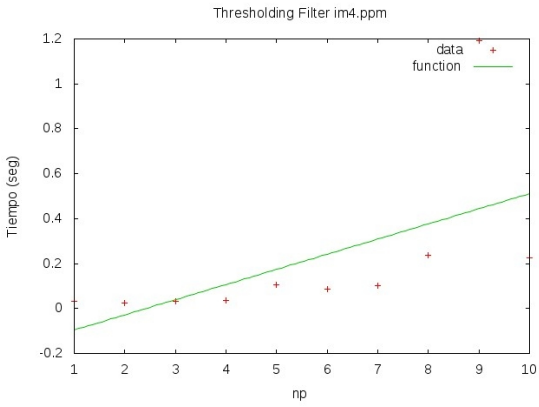
In this step we do measurements by the execution the program using various number of processors and sizes in each filter:

a. Thresholding Filter:









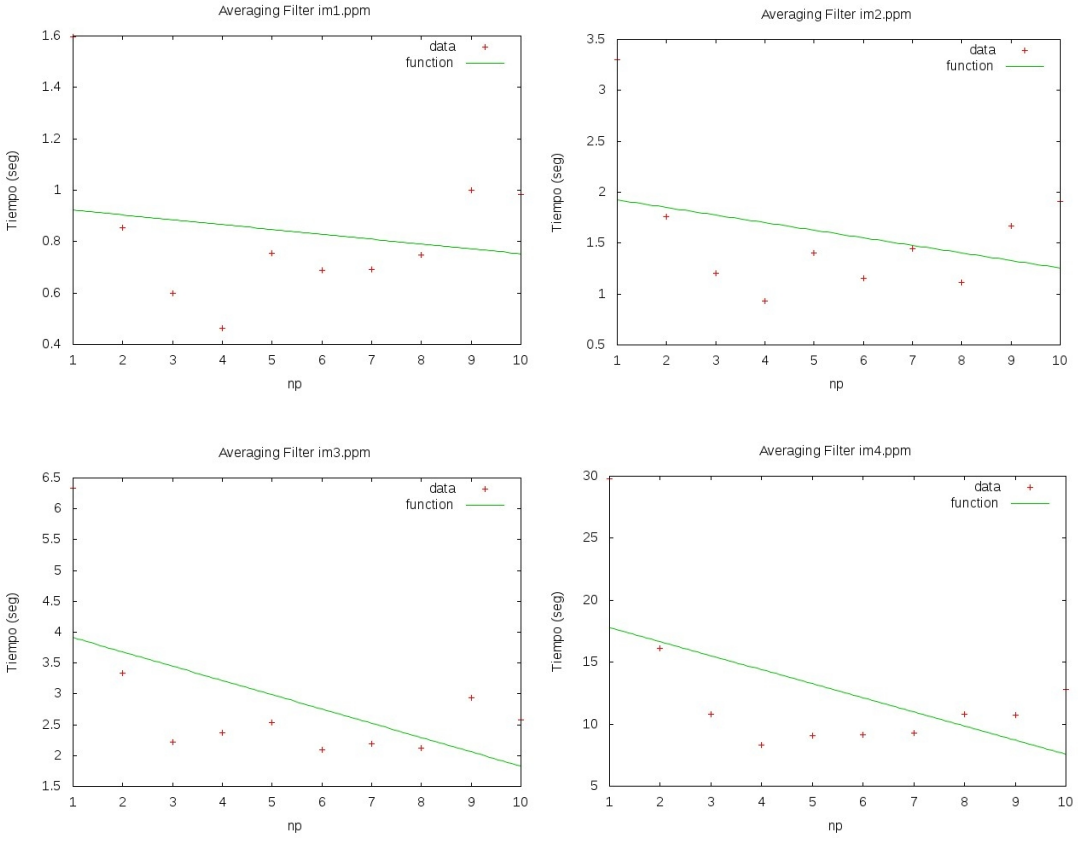
Here you can see 4 graphs, on each one there is a different input size according to the number of pixels of every picture. And, on every graph, the time is analized with different numbers of processors.

We can see that the optimal number of processor is 4 for all the tests.

Besides we can examine 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 processors is greater, because it needs more communication and has to wait.

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 processors and with a default radius 100

We can see that the optimal number of processors is 4 for the majority of the tests.

Besides we can examine that if the size is smaller then the necessary time is smaller too. Because it is around 1sec for smallest size and it is around 20sec for the biggest.

According to both filters, we can see on all the graphs that the parallelization improves the time of execution.
In addition we can detect that with only 1 processor the necessary time is bigger than with more than one.

3. **Source-code of our program**
    a. **Thresholding Filter:**
        You can find the code the attached document.

    b. Averaging Filter:
        You can find the code the attached document.

Build the project:

Install: *sudo apt-get install libcr-dev mpich2 mpich2-doc*

Compile: *make*

Run: *bash run.s*