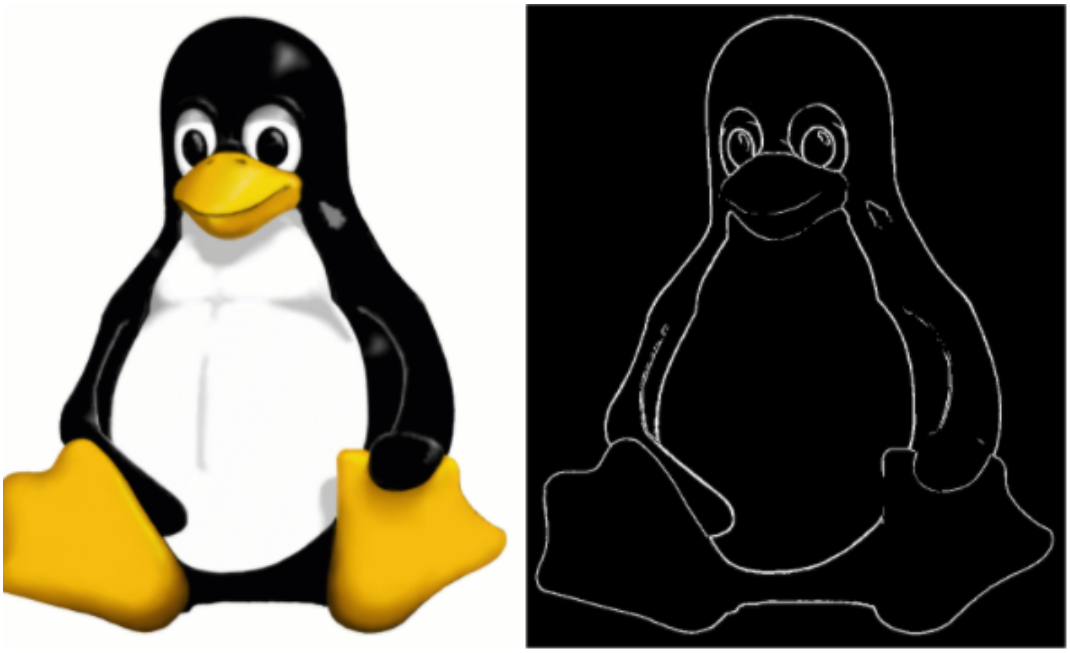
INF560 - Parallel and Distributed Algorithms (2021-2022)

Final Project

Image Filtering

submitted by

Ali Haidar
Marc Dufay



Institut Polytechnique de Paris

Introduction

The goal of the INF560 Parallel and Distributed Algorithms project is to improve the performance of an existing sequential code by using parallel programming paradigme. The target program code takes in input an Image as GIF file and gives in output the filtered gif by applying in a sequential way these 3 differrent filters : (Grey Filter, Blur Filter and Sobel Filter). To speed up the execution of the code, we will explore several ways to parallize the code and we will see the result of each method and at the end we will represent the best way that gave the best result.
In this kind of application, speedup may be achieved by parallelizing on the different subimages that compose the GIF file or on the pixels of each subimage.

In order to do so, different technologies have been used:

- MPI to distribute the computation over different nodes.
- OpenMP, to parallelize loops over different threads in a process
- Nvidia CUDA to exploit the GPU for extremely parallel computations.

1 Merge Load Data and Grey Filter

The pipeline of the program was :

1. Load a gif file.
2. Apply Grey Filter, Blur Filter then Sobel Filter.
3. Store the gif file.

The first optimization was to execute the loading and the storing in a parallel way by using Open-MP. It helped us to improve the execution time of the code.

To improve further the execution time of the filter phase, we need to take a look on the objective of each filer :

- **Grey filter**: converts a colored 3-channels gif to a grey gif by averaging rgb value at each pixel and saving it a 3-channels gif.
- **Blur filter**: blurs the upper and the bottom part of the image until every pixel is processed enough compared to the previous signal by a certain threshold. It needs a two-dimension kernel to see neighboring pixels, meaning that it has four nested for loops. In addition, it uses a do-while loop with a flag, which needs a careful attention.
- **Sobel filter**: It is an edge-detection algorithm that outputs an image whose background is black and whose edges are white. It also uses a kernel like the blur filter and the size is 3x3.The function is supposed to have four nested for loops, but it directly fetches adjacent pixels only with two for loops because the kernel size is small enough.

Using this information, we improved the performance of our program by applying the grey filter while loading the data and storing the pixels in a array of integers instead of an array of struct pixel that contains r, g and b. Using this optimization, we were able to reduce our memory size by 3 and execute the grey filter in the best optimal way and this will lead to reduce the data transfered through MPI or to GPU by 3.

2 Simpler export

The `store_pixels` function is very generic (can be used to store any colored set of pixels into a gif) and therefore not really adapted for our specific algorithm. Indeed one major inconvenient (and advantage looking at the size) of the gif format is that images can have only 256 different colors. This is problematic for a colored image where there can be up to 256^3 different colors but absolutely not for a black and white image with exactly at most 256 different shades of grey.

So the algorithm in `store_pixels` was replaced: instead of looking for each pixel at the colormap in order to find if its color is present or not, we create a colormap of size 256 with all black and white shades then the index in the colormap of a color is itself.

This new algorithm is up to 100 faster. Because we only need to do a simple lookup for each pixel, offloading this work in another node is not worth it.

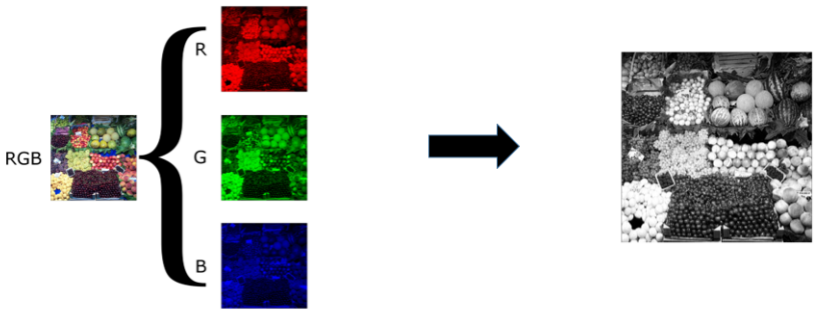


Figure 1: Transforming 3 channels image to 1 channel

3 Multi Thread Approach - Open MP

Our approach was to distribute the threads among the iteration of for loops in each filter. This was done by collapsing some loops and sometime change the order of loops to get better performance.

We tested two ways to schedule threads :

- **Static scheduling** by using default option, it tries to assign an equal number of iterations to every threads at the beginning.
- **Dynamic scheduling** it assigns 1 iteration to every thread.

id	n_images	OMP_Static	OMP_Dynamic	SEQ
1	1	0.0324	0.400932	0.143254
2	10	0.033673	0.707133	0.180107
3	1	0.000247	0.004922	0.001195
4	10	0.014991	0.360147	0.086596
5	1	0.015084	0.185749	0.067841
6	1	1.922356	17.042047	7.230838
7	1	2.002311	18.194207	7.108456
8	33	0.000633	0.006851	0.000953
9	1	0.002261	0.031654	0.009029
10	5	0.195735	2.50322	0.783461
11	20	0.413938	6.154721	1.670568
12	1	0.000527	0.010798	0.002742
13	1	0.013868	0.158816	0.056715
14	19	0.03765	0.916724	0.205066
15	10	0.035294	0.80337	0.187691

Table 1: Table shows the difference between the execution time of versions OMP-Static, OMP-Dynamic and the sequential version

We can see clearly from the table that dynamic scheduling version is slower than the static scheduling and than the sequential version. The dynamic scheduling assigns a thread at each iteration which can cause some overhead in the program. In another hand, the static scheduling version seems to be faster than the sequential one as expected.

4 Distributed Memory approach - MPI

MPI is used to share the computations among different nodes with nonshared memory. The data is shared in 2 different ways depending on the number of images in the gif file and the number of nodes available:

- Even distribution: the images of the gif file are evenly distributed among all nodes. Node 0 load the gif, shares it and then all nodes (including 0) apply the sobelf filter to their images and then send them back to node 0 which stores it.

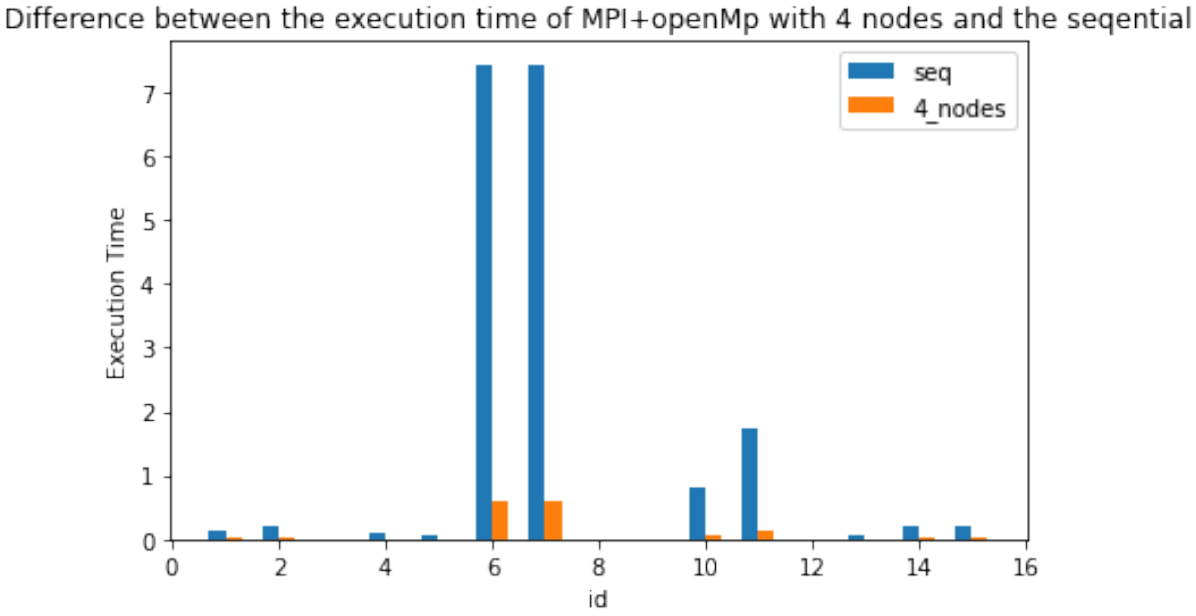
- Server-client model: Node 0 is the server, it loads the gif file then distributes parts of the images to all the clients for them to process it. Every time a client has finished processing its part, it sends it back and gets a new one.

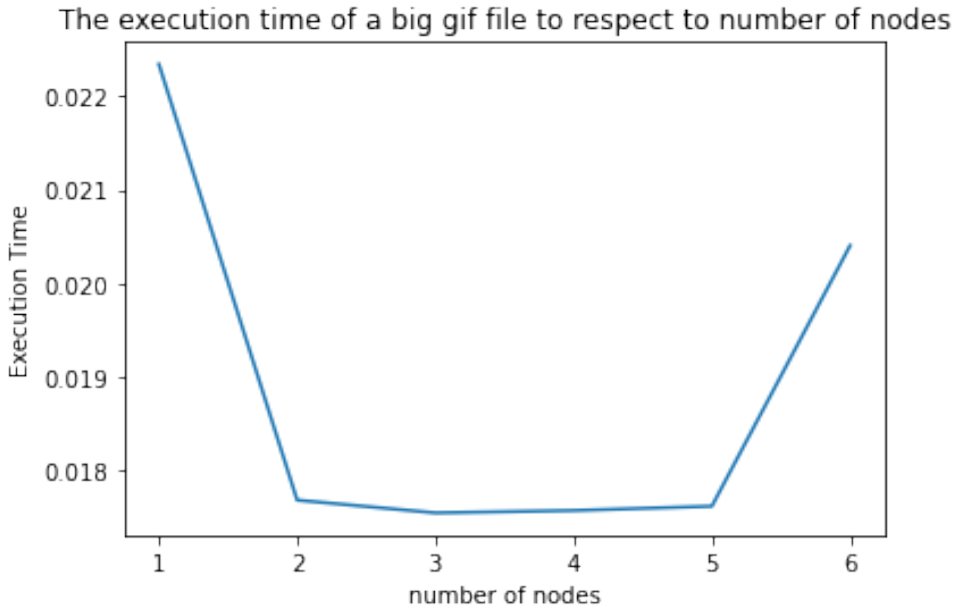
We will focus on the server-client model which is by far more complex. The following points were considered:

- Image cutting: for each image, we needed to decide how to cut it in parts in order to distribute it to all nodes. First, the top and the bottom of the image are sent. Indeed, only these parts need to have the blur filter applied on them and each pixel needs data on pixels that can be very far away (but still on

the top or bottom part) to compute the blur. Therefore it is not possible to cut them more without spending far more time in transmission sending pixels when needed. Except the top and the bottom part, the middle part was then cut into pieces along the horizontal line to preserve data locality and then sent.

- Unprocessed lines: In order to compute the sobelf filter on a pixel, we need to know the color of its surrounding pixels. Therefore when sending a block to be treated, we need to send its surrounding pixels even though they are not modified.
- Asynchronous sending: the server sends pixel data in an asynchronous way in order not to block the sending to the next client.





5 GPU Approach - CUDA

CUDA: it's the API that Nvidia develops to provide primitives to use their GPUs for computational purposes. It consists in a custom compiler (nvcc) built upon GCC, that takes as input .cu files, which are C programs that use CUDA functions to transfer data (with cudamemcpy()) and to offload code (by running kernels) to the GPU.

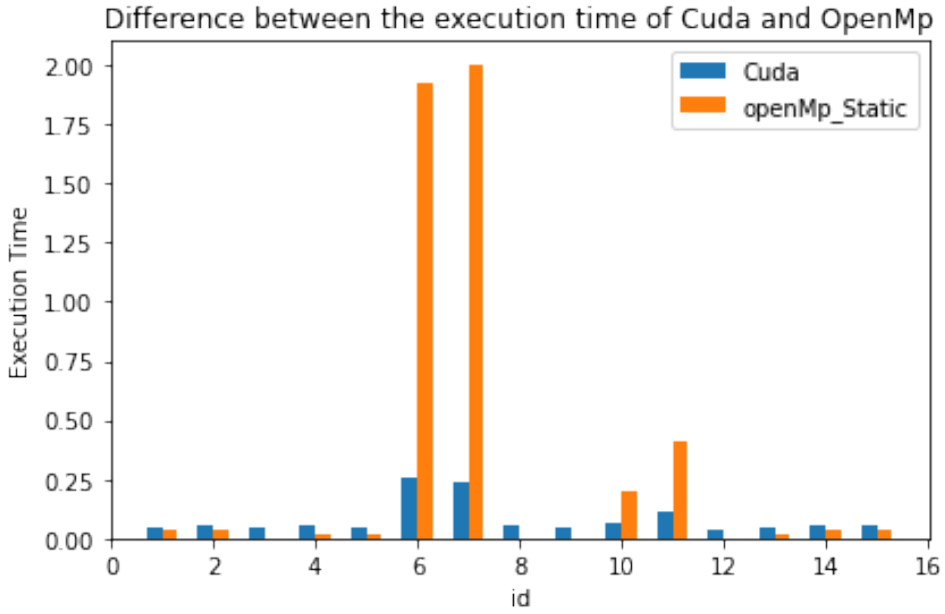
Our expectation before starting the project was that Cuda solution should perform better than OpenMp,MPI and sequential. After testing the cuda solution and running several experiments, we deduced that it is true that Cuda solution is better than the sequential solution and the MPI solution but in several cases OpenMp solution can perform better especially when the size of image is small but when the size of image becomes big, Cuda performs better. This due of the time that cpu takes to transfer the data from the CPU to the GPU.

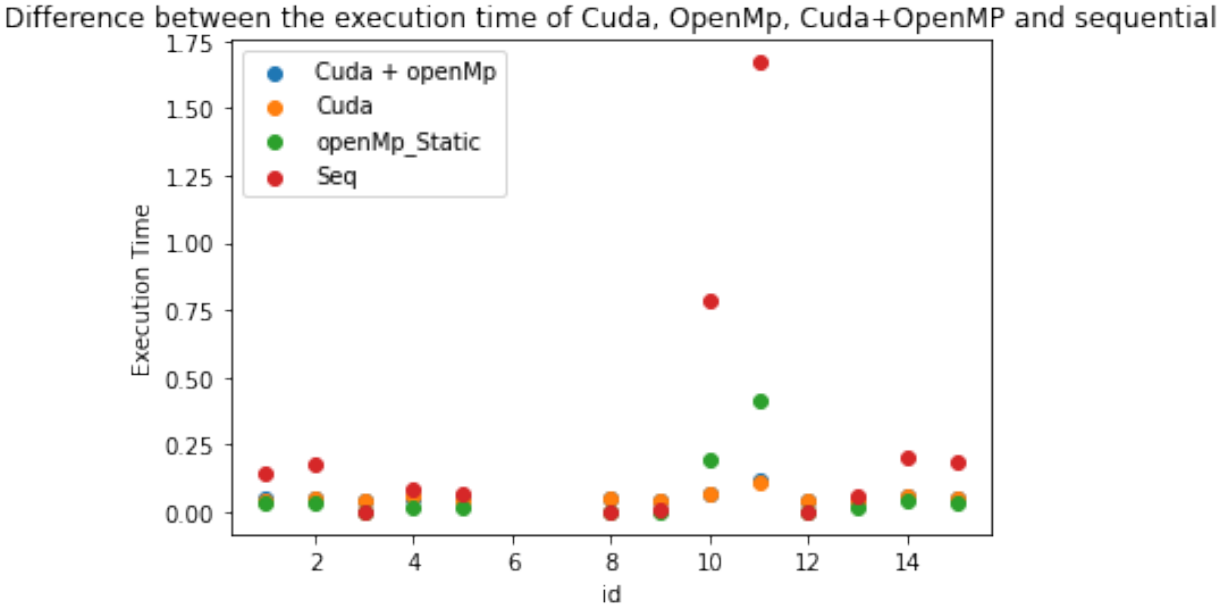
We tested a solution that combines Cuda and OpenMP by distibuting the images of a gif to multiple thread and than each thread compute the filtering of each image using Cuda but it appears that the performance of this solution was close the only Cuda solution.

id	n_images	Seq	OpenMp_Static	Cuda	Cuda+OpenMp
1	1	0.143254	0.0324	0.045637	0.053857
2	10	0.180107	0.033673	0.052293	0.053865
3	1	0.001195	0.000247	0.042893	0.041675
4	10	0.086596	0.014991	0.056975	0.047679
5	1	0.067841	0.015084	0.042653	0.046317
6	1	7.230838	1.922356	0.258823	0.319672
7	1	7.108456	2.002311	0.236163	0.296838
8	33	0.000953	0.000633	0.052356	0.047611
9	1	0.009029	0.002261	0.041276	0.040312
10	5	0.783461	0.195735	0.067535	0.070368
11	20	1.670568	0.413938	0.109564	0.119752
12	1	0.002742	0.000527	0.039763	0.041629
13	1	0.056715	0.013868	0.043716	0.043057
14	19	0.205066	0.03765	0.058831	0.061963
15	10	0.187691	0.035294	0.050937	0.05318

Table 2: Table shows the difference between the execution time of versions OMP-Static, Cuda, Cuda+OpenMp and the sequential version

For multiple graphics card support, the graphics cards are distributed in a round-robin way among all nodes running on the same computer.





6 The Hybrid Model

Our algorithm uses MPI, OpenMP and CUDA on different parts of the code and depending on the available hardware.

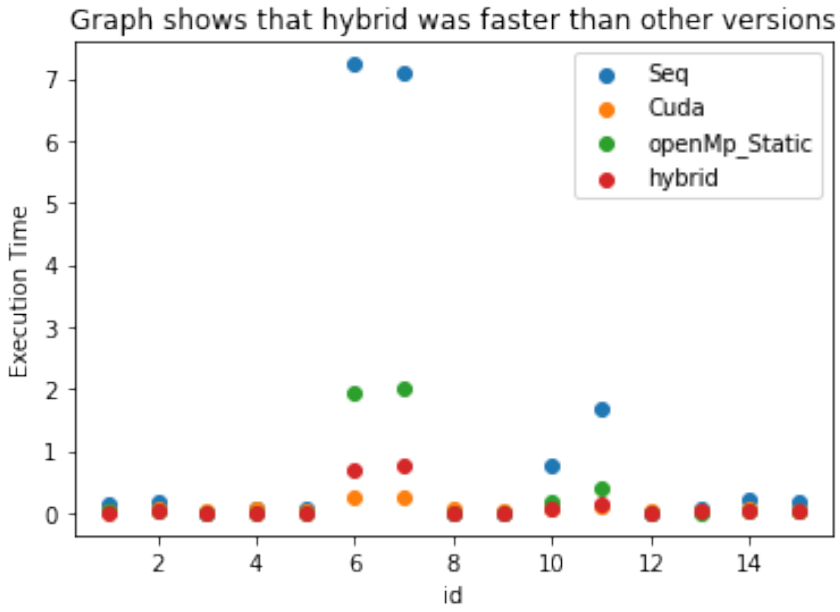
- First, the gif picture is loaded and converted to grey on the first node. Pixel loading and conversion is made accelerated using OpenMP to use all the cores available.
- Then, images or parts of images are distributed to all the nodes using MPI. If there are enough nodes, a server/clients model is used. Otherwise images are scattered evenly among all nodes.
- When receiving a block to process, if a CUDA compatible graphic card is available, the blur and sobelf filter are applied on the graphic card. If there are

multiple graphic cards, they are distributed among available the nodes running on the same computer. If no graphic card is available, the computation is done on the CPU using OpenMP.

- All processed blocks are sent back to the first node which convert them back to a gif file using a fast store_pixel function.

id	n_images	Seq	Cuda	openMp_Static	hybrid
1	1	0.143254	0.045637	0.0324	0.012004
2	10	0.180107	0.052293	0.033673	0.018665
3	1	0.001195	0.042893	0.000247	0.002143
4	10	0.086596	0.056975	0.014991	0.007053
5	1	0.067841	0.042653	0.015084	0.004221
6	1	7.230838	0.258823	1.922356	0.676522
7	1	7.108456	0.236163	2.002311	0.783289
8	33	0.000953	0.052356	0.000633	0.003535
9	1	0.009029	0.041276	0.002261	0.003347
10	5	0.783461	0.067535	0.195735	0.072146
11	20	1.670568	0.109564	0.413938	0.150349
12	1	0.002742	0.039763	0.000527	0.000267
13	1	0.056715	0.043716	0.013868	0.017533
14	19	0.205066	0.058831	0.03765	0.020039
15	10	0.187691	0.050937	0.035294	0.018029

Table 3: Table shows the improvement of hybrid version over all other versions



We can see that the performance of the hybrid model was better than others.

7 Conclusion

To conclude, We improved the existing Sobelf filter algorithm in order to allow it to run on multiple cores, computers and graphic cards. This algorithm scales relatively well but there are a few limitations:

- The gif loading process is done by an external librairy which loads all images. If it were possible for each done to load and save only its part of the whole file, this would allow the algorithm to be much faster on a large number of nodes.
- A relatively small number of operations are done on each pixel (the most being done for the blur filter where a $(2\text{blur\_size})^2$ pixel much be read for each pixel during an iteration. But this can also be reduced using better data structures. Therefore, we can easily spend more time copying and sending data than filtering it. This heavily limits the potential of this algorithm on non-shared memory systems.