Parallelization of an image processing application using MPI, OpenMP and CUDA

Fabrizio Indirli (*fabrizio.indirli@outlook.it*),
Seongbim Lim (*seongbim.lim@polytechnique.edu*)

Abstract—The project for the INF560 Parallel and Distributed Algorithms course consisted in parallelizing an existing application to improve its execution time when ran on multiple execution units. The target application was an image processing program that applies different filters (*Grey Filter*, *Sobel Filter* and *Blur Filter*) on GIF files. 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. After testing various approaches, an hybrid model was built to apply the best parallelization strategy according to the input set and the available resources.

Index Terms—HPC, Parallelization, Offloading, MPI, OpenMP, CUDA.

1 Introduction

NOWADAYS, in the era of Big Data and High Performance Computing, huge quantities of data are used to compute complex outputs in an always growing number of sectors: weather forecast, automotive, data analysis, engineering applications, and so on. This led to the need for more powerful computing systems, capable of executing operations as fast as possible. Up to the late 90s, the main approaches to achieve this speedup in the CPUs were the increase of frequency (thanks to advancements in the lithography technology) and of the IPC (*Instructions per cycle*), by optimizing the microarchitecture. However, in the last years these approaches have not been very effective, mainly because the problems of *dark silicon* and *heat management* put a limit on the CPUs' maximum frequency. This led to new paradigms to achieve performance increase in computation: *parallelization* on multiple cores and *offloading* on specific processing units that are optimized for executing certain type of computations (e.g. GPU). As a result, modern processors have usually multiple cores, capable of fetching and executing multiple instructions at each clock cycle, and various kind of accelerators are available for both the consumer and the professional/scientific market. Thus, modern *supercomputers* are made of thousands of nodes that can work in parallel.

However, gaining a speedup from a multiprocessor and/or accelerated system is not always trivial: developers must explicitly program their applications to exploit the available computing resources, by finding pieces of code that can be run in parallel (for example, splitting input data on different tasks) and then implementing the parallelization primitives in their code. In order to allow the developers to do so, various technologies have been developed. In this project, 3 of them have been used:

- **MPI**: the *Message Passing Interface* is a library for C/C++ that is used to exchange data between different processes (which can also run on different nodes) in a *Distributed Memory Model*.
- **OpenMP**: this API, integrated in the GCC compiler, provides some compiler directives to automatically parallelize certain

portions of code, such as *for-loops*, on vectors whose cells are independent from each other, on different threads. It uses a *Shared-Memory model* and it's widely used in HPC, also because it hides some management complexity and it's compatible with C/C++ and FORTRAN.

- **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.

The goal of the project was to parallelize an existing application using the aforementioned techniques, combining them in a way such that the best parallelization strategy is automatically found and applied taking into account the program's input and the available computing resources.

2 THE PROGRAM

The object of the study was an image processing application that applies some filters on the input images. The **input** of the program is a *GIF* file, and the program automatically applies the filters on all the sub-images of the file. The applied filters are, in order:

1) **Grey filter**: It converts the image to greyscale by averaging rgb value at each pixel. It has one for loop.
2) **Blur filter**: It 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.
3) **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 of it is 3×3. 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.

2.1 Domain decomposition strategies

By analyzing the program, 3 possible levels of parallelism have been found:

- **Different sub-images**: Each GIF input file may contain multiple sub-images that can be processed independently from each other
- **Pixels of each image**: The filters usually iterate on all the pixels of the sub-image, which are then treated independently from each other, provided that they can access a structure containing all the original pixels.
- **Filters pipelining**: Although it's not possible to treat the different filters as they were completely independent from each other (since they should be applied always in the same order), it is possible to build a pipeline in which each node (or group of nodes) specializes in computing a specific filter.

3 SEQUENTIAL PERFORMANCES

The **input set** used to test the performances was composed of 15 GIF images, each with different sizes and number of sub-images. Here are the computation times with the sequential algorithm:

file id	images	width	height	filters time
1	1	1484	913	0.188928
2	10	500	500	0.377636
3	1	200	200	0.004415
4	10	500	281	0.099927
5	1	1200	600	0.111504
6	1	9932	7016	11.922019
7	1	9933	7016	11.794629
8	33	30	60	0.001166
9	1	585	450	0.039339
10	5	1920	1080	1.226660
11	20	1200	1200	2.196476
12	1	320	200	0.003229
13	1	800	950	0.125662
14	19	640	360	0.321406
15	10	480	480	0.220969

TABLE 1
Processing times with the sequential algorithm

In addition to this, we have added 5 additional GIF files with a very high resolution (more than 15 MP) to better study the performance improvements with some of the parallelization paradigms used. The performances of the sequential algorithm on the *additional* input set were the following:

file id	images	width	height	filters time
16	7	4196	2362	10.294351
17	7	3436	4724	15.597356
18	2	7087	7087	17.023842
19	2	9677	4608	14.913220
20	1	9640	7584	12.512858
21	5	4000	4500	7.278869

TABLE 2
Processing times of additional input set with the sequential algorithm

4 MPI-ONLY PERFORMANCES

To understand the impact that MPI parallelization could have on the program, we have implemented MPI to separately exploit the 3 different levels of parallelization and measured the average speedup on the computation of the input set for each method. In addition, each method has also been tried with both *Standard communications* and *Non-blocking communications*.

4.1 MPI parallelization on sub-images

Each sub-image of a GIF file can be processed separately from the others, hence this was supposed to be the most effective *domain decomposition* strategy among the 3 different levels.
The strategy consists in dividing the sub-images evenly among the available *MPI ranks*. Each rank gets information about total number of ranks R and number of sub-images I. Then, if the number of ranks is greater or equal than the number of images, each rank gets 1 image; otherwise, each node gets at least floor(I/R) (integer division) images to compute, and if there is a rest E to that division, it gets evenly distributed between the first E nodes:

$$I_r = \begin{cases} 1 & \text{if } R \geq I \\ (I/R)+1 & \text{if } R < I \wedge r < (R\%I) \\ I/R & \text{otherwise} \end{cases} \quad (1)$$

In the program that we have built, rank 0 is the only process that loads the file, hence it is in charge of computing the number of images that each node has to process. Then:

1) The number of images-per-node is sent to every other node
2) For each other rank j, rank 0 builds and sends an array with the sizes (width and height) of the images that rank j has to compute, so that it can allocate the memory to receive them;
3) For each other rank j, rank 0 sends to it the pixels of the images that it has to compute.
4) Rank 0 applies the filter on its images only after having sent all the other images to the other ranks; the other ranks wait to receive all their images before starting the computation.
5) For each rank and for each sub-image of that rank, Rank 0 receives the image
6) Rank 0 saves the processed image.

We have tested this strategy with 4 ranks, with different combinations of *number of nodes* and *communication type*.

4.1.1 4 ranks on 1 node, blocking communications

In Fig. 2 and Fig. 1 we have plotted the results that were obtained on the initial input set using blocking communications (such as *Send()* and *Recv()*) and the following run command:

salloc -n 4 -N 1 mpirun ./sobelf input output

The average *speedup* was about 1.13 and it was evident only on files with more than 1 sub-image. The results of file 8, which has 33 sub-images but a very low resolution (60x60), suggest that this approach should be used only of images that are big enough, otherwise the communication overheads are greater than the computation speedup.

4.1.2 4 ranks on 2 nodes, blocking communications

In Fig. 3 we have plotted the results that were obtained on the initial input set using blocking communications (such as *Send()* and *Recv()*)

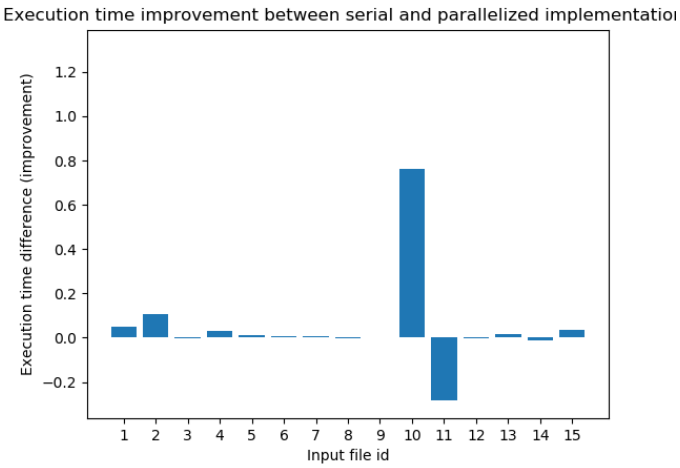

Fig. 1. Difference of execution time between sequential algorithm and MPI parallelization on sub-images, with blocking communications and 4 ranks on 1 node.

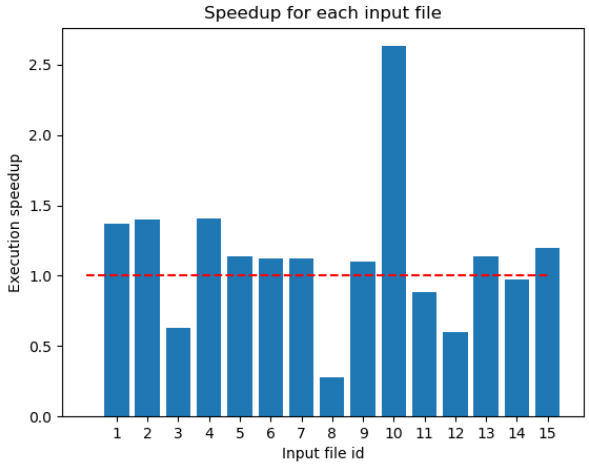

Fig. 2. Speedup of MPI parallelization on sub-images over the sequential algorithm, with blocking communications and 4 ranks on 1 node.

and the following run command:

salloc -n 4 -N 2 mpirun ./sobelf input output

With 2 different nodes, the average speedup was 0.632, with very low performances in the computation of files with multiple images. This was probably due to low network performances.

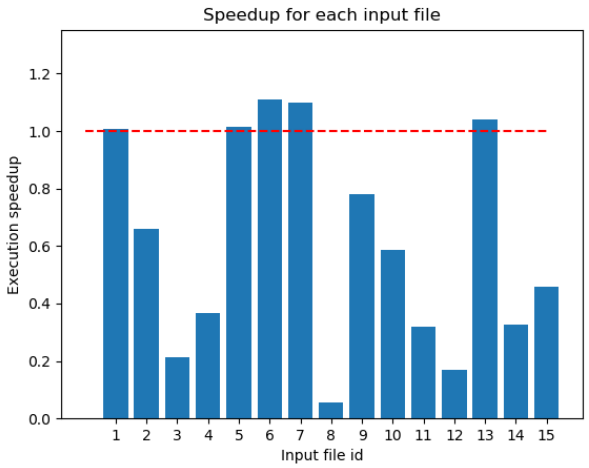

Fig. 3. Speedup of MPI parallelization on sub-images over the sequential algorithm, with blocking communications and 4 ranks on 2 nodes.

4.1.3 4 ranks on 1 node, non blocking communications

With the blocking communications, each node has to wait to receive all its images before it can start the computation, and rank 0 has to send all the images to the other nodes before it can start as well. Hence, we thought that using *non-blocking communications* could improve the performances of the algorithm. Indeed, as shown in Fig. 4, there was an improvement that led to an average speedup of 1.5184, with peaks of 2.5 - 3 on files with high resolution and many sub-images.

4.1.4 4 ranks on 2 nodes, non blocking communications

With the 4 ranks distributed on 2 different nodes (Fig. 5), the results were slightly better than the relative *blocking* case, but the average speedup was still below 1 and, in general, performances were poor.

4.2 MPI parallelization on pixels

The second parallelism method has been tried on MPI, too. In this paradigm, the multiple MPI ranks are used to compute the different parts of each picture. In particular, the pictures are split on their height, and each rank computes one horizontal stripe of the image. The algorithm consists in the following steps:

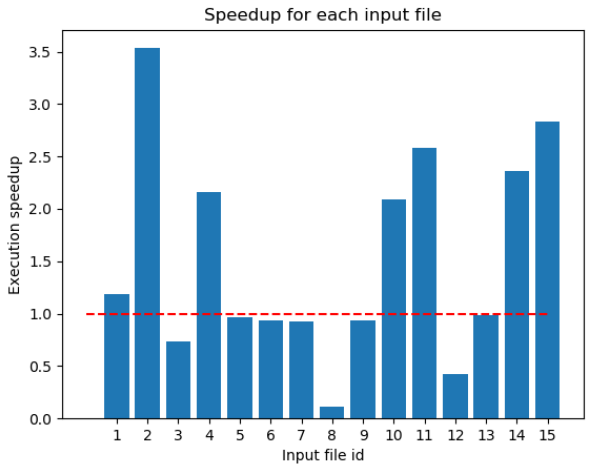

Fig. 4. Speedup of MPI parallelization on sub-images over the sequential algorithm, with non-blocking communications and 4 ranks on 1 node.

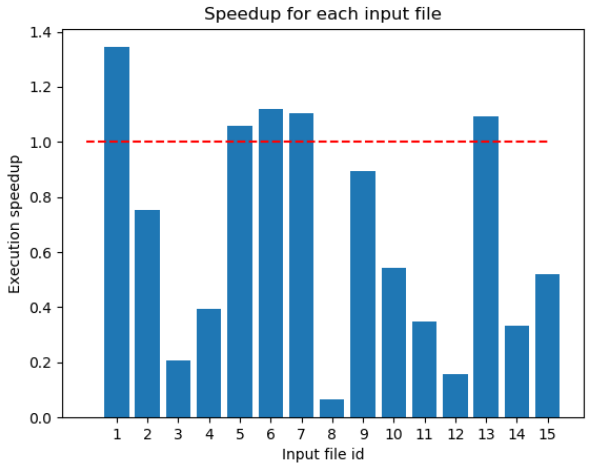

Fig. 5. Speedup of MPI parallelization on sub-images over the sequential algorithm, with non-blocking communications and 4 ranks on 2 nodes.

1) Rank 0 is the only process that loads the file from disk and reads the number of images
2) Rank 0 broadcasts the number of images
3) Rank 0 builds and broadcasts an array with the sizes of the images in the file
4) With these informations, each rank computes the index of the first row and of the final row of the part that it has to compute
5) Rank 0 broadcasts all the images with a non-blocking communication
6) Each rank applies the filters on its part of the images
7) Rank 0 gathers all the parts of the images with an *Igatherv()*

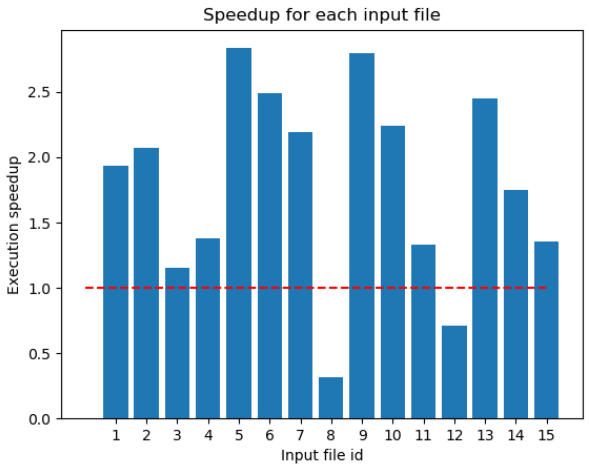

Fig. 6. Speedup of MPI parallelization on pixels over the sequential algorithm, with 4 ranks on 1 node.

Testing this model with 4 ranks on 1 node, an average speedup of 1.80 was measured, and almost all the images in the input set were processed faster than the sequential algorithm. Only the smallest images, such as the 8th, didn't perform better.

4.3 Summary on MPI

Among the 2 different parallelization levels, the *pixels parallelization* seems to be the most appropriate for MPI (Fig. 8). Tests varying the number of ranks have also been executed, and Fig. 7 shows that increasing the number of ranks over 2 was not necessarily improving the performances. This gave precious information on how to approach the *domain decomposition* problem.

4.4 Filters pipelining

We have also tried a simple pipelined architecture, in which the ranks (if at least 3 were available) were divided in 3 groups, each one in charge of applying one filter; after that, rank j would send the picture to rank j + *numRanksPerGroup* or, if j was in the 3rd group, to rank 0. Rank 0 would then receive all the images and save them in the final file.
However, this method resulted in an increase of the computation time for all the tested images,

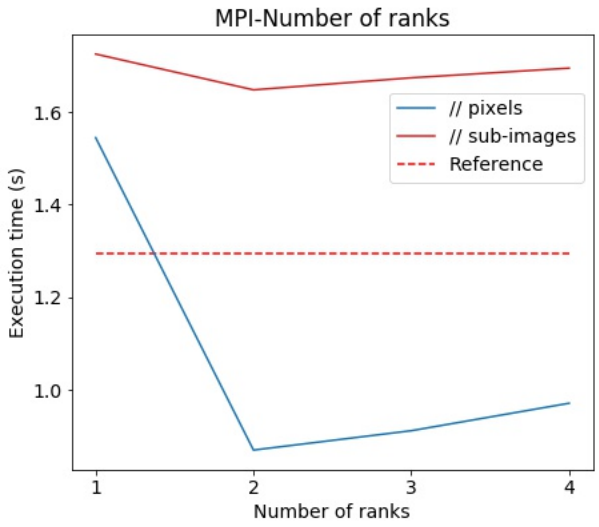

Fig. 7. Average execution time with different MPI strategies and number of ranks (on 1 node).

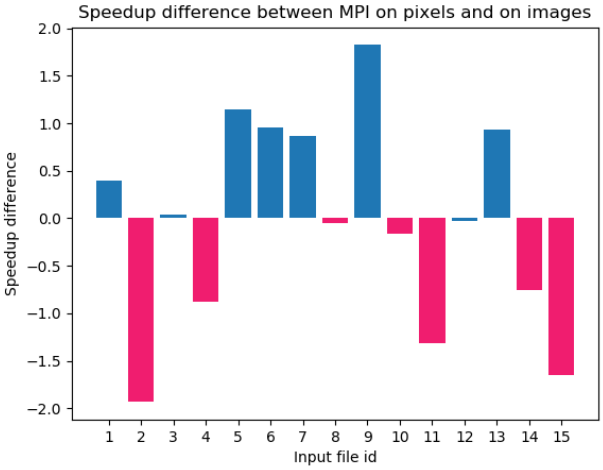

Fig. 8. Difference of speedup between MPI on pixels and on images.

with an average speedup of 0.556. Indeed, this method requires more communications, hence yields more overhead. In the end, we decided not to include this paradigm in the final program.

5 OpenMP-only performances

OpenMP is a shared-memory programming model, thus sharing image array was expected to show a good performance compared to MPI. There were two main approaches. First, OpenMP parallelization at the pixel-level. Second, the program compares whether the number of threads is larger than the number of sub-images. If the condition is true, it parallelizes each filter at pixel-level. Otherwise, each thread filters sub-images once or more times. In addition, there were experiments with different *for* scheduling options.

5.1 Pixel-level parallelization

The grey filter has one *for loop* looping through all pixels within a sub-image. Simply, a *for* directive was used for it. The blur filter consists of four blocks; two blurring top 10% and bottom 10% of image, another one just copying the rest of the image, and last one updating the new image. Two blocks that actually perform blur computation have the same code with different range. If OpenMP was to applied to them separately, it means that there will be a implicit barrier between them. So we merged these two loops and copying block altogether to remove the barrier and removed two implicit barriers, though the improvement was imperceptible.

The *blur filtering* function has four *for* loops; two looping through row and column pixels, other two looping through neighboring pixels for each pixel according to the kernel. Four loops may indicate four *for* directives. However, the kernel size is too small to be parallelized, and fetching pixel information from different rows does not seem to fit to the shared-memory programming model. In conclusion, we decided to use only two *for* directives to loop over all pixels. In addition, the *collapse* directive and different scheduling methods were tried.

The performance was improved a lot compared to the sequential application in general, but it could be improved more depending on the choice of scheduling (figure 10). The static scheduling was fast without specifying the chunk size, while the dynamic scheduling showed a very poor performance without specific chunk size.

This is because the OpenMP static scheduling, by default, tries to assign the equal number of iterations to every threads at the beginning, while dynamic scheduling assigns 1 iteration

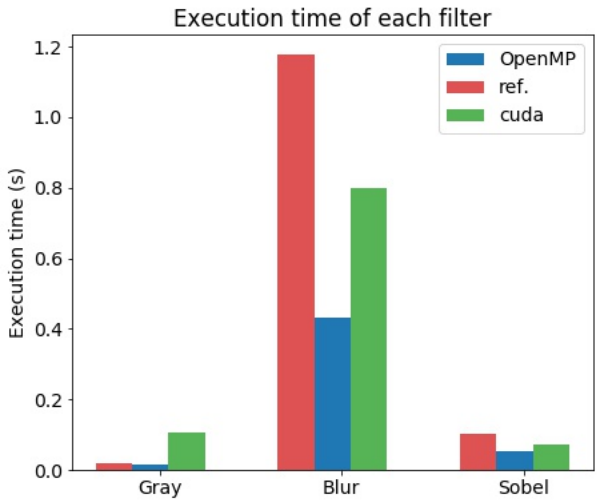

Fig. 9. Analysis of execution time for each filter. Comparison between the sequential algorithm and the one pallelized with OpenMP

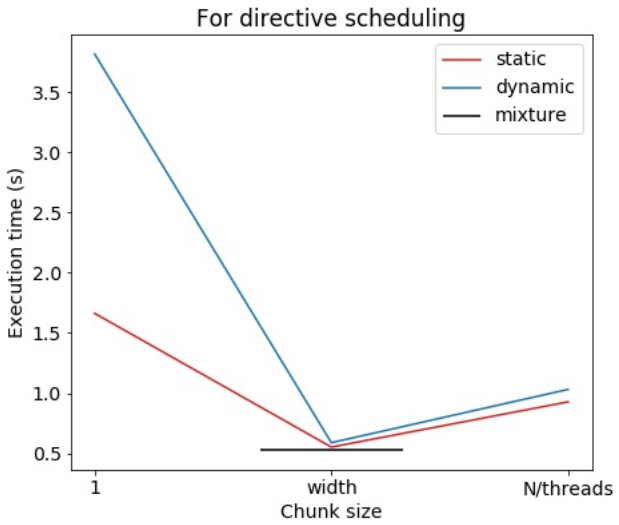

Fig. 10. Performance depending on the choice of scheduling options and the chunk size for *for* directive.

to every thread. As a result, the chuck size 1 was too small to be effective in both scheduling methods, hence dividing iterations evenly gave better result. However, it turned out that the optimal chunk size was neither of them, simply dividing workloads by the number of threads may cause poor performance. After all, every thread usually does not have the same speed and may be executing other programs in the background. Thus, the optimal value should be some value in between. Our choice of the size was the number of width pixels for every *for* directive.

We went further and optimized which block would rather be static or dynamic. The basic principle was to make simple code static, such as gray filter or updating new pixel value, and to make complex code dynamic. For example, blur filter, having long different instructions according to the if statement, is not well balanced. Consequently, it had better be dynamic. The improvement seems small in figure 10, but it was certain that it took less time than the others.

To sum up, OpenMP parallelized each filter with optimal choice of scheduling and chunk size. Since OpenMP is a shared-memory programming model, it was expected to be good at sequential array processing, in this case images, and we were able to get a large improvement in performance.

5.2 Sub-image level parallelization

Another possible scheme is to assign sub-images to each thread. If a file had a lot of sub-images, this scheme makes more sense than pixel-level parallelization. Considering the fact that there can be files without any sub-images, sub-images parallelization would not perform well in this specific case. So we put an if statement, that if the number of sub-images are larger than the number of threads, the computer allocates each sub-image to each thread, otherwise it does pixel-level parallelization. As shown in figure 11, there was a confident improvement in performance.

5.3 Summary of OpenMP

To sum up, both pixel-level and sub-image-level parallelization were implemented with optimal *for* directive scheduling. Chunk size was also optimized. Although there were a few images that became slower to be processed, the application achieved about 2 times speedup for most of test images. It turned out that those few images that took more time were the same cases in MPI. They have small resolution in

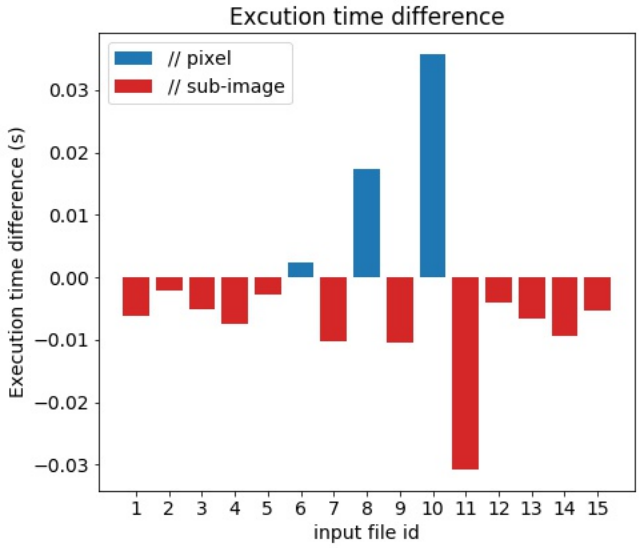

Fig. 11. Comparison between pixel-level and sub-image-level parallelization

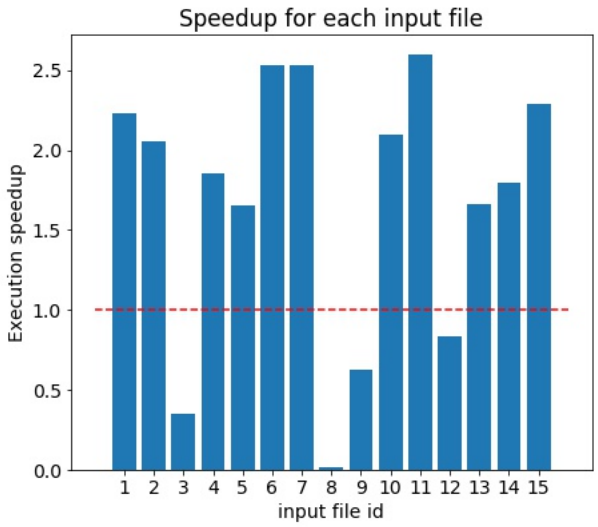

Fig. 13. Speedup ratio of OpenMP parallelization

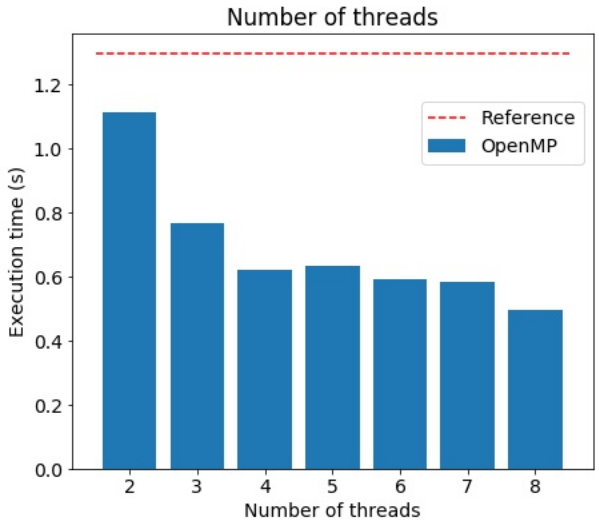

Fig. 12. The impact of the number of threads on performance

common, for example less than around 600 by 500. It seems that small amount of works are not worth parallelizing, probably because of the cost of thread creation overhead. The performance was also measured varying the number of threads. The result was not linear and the performance only improved a lot visually up to 4 threads.

6 CUDA-ONLY PERFORMANCES

Implementing GPU was expected to improve the performance a lot, but it was not the case. A brief result is shown in figure 9. CUDA performance was not as good as the sequential computing when it comes to the gray filter. The possible explanation is that a simple work, such as gray filter, costs more to communicate and transfer data than sequential computing. As soon as the computation becomes complex, for example sobel filter, CUDA gets faster at last. Since blur filter is the most complex and time consuming process, CUDA showed rather moderate improvement compared to the sequential model, but it was still slower than the OpenMP implementation. When the input image became much larger (figure 14), the computation of *sobel filter* on CUDA seemed more efficient, and the gap between OpenMP got smaller.

7 HYBRID MODEL

After analyzing the performances of the single models, an hybrid model has been implemented. This model is capable of using all the 3 parallelization technologies and decides the best strategy according to the input set and to the available resources, following a decision tree.

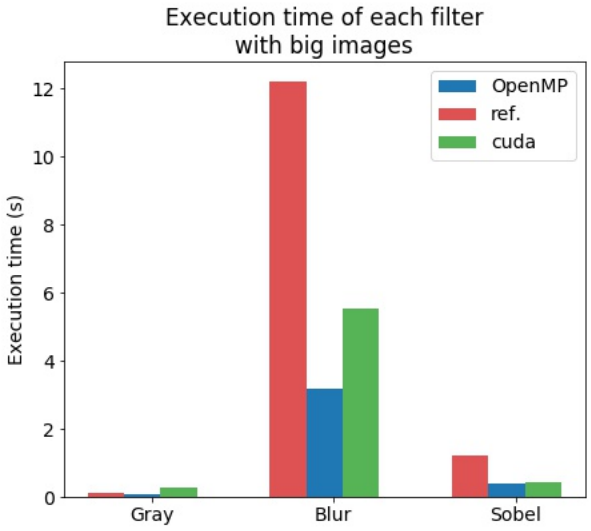

Fig. 14. Execution time with the *additional* big images. Note that time scale is about 10 times larger compared to figure 9.

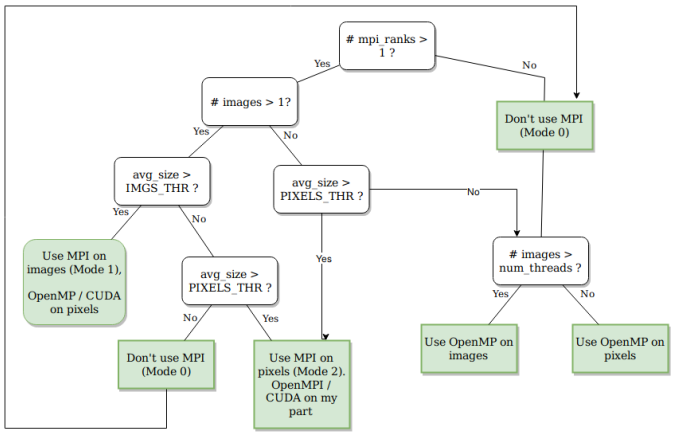

Fig. 15. Decision tree used to select MPI mode

First of all, Rank 0 collects all the needed infos on the input set (number of images, average size of images) and available resources (number of MPI ranks) and selects one of the 3 MPI modes:

- **Mode 0**: MPI is not used, parallelization happens with OpenMP/CUDA. This mode is selected when the resolution is low and/or the is a limited number of images in the file.
- **Mode 1**: MPI is used to parallelize on the different images of the file.
- **Mode 2**: MPI is used to parallelize on the pixels of each image.

The selection of the mode follows the decision tree illustrated in the Fig. 15

In addition, thresholds for OpenMP and CUDA have also been inserted, since small images don't benefit from parallelization with these models. Indeed, after choosing the MPI Mode, ranks choose whether to use OpenMP or CUDA according to those thresholds.

7.1 Hybrid model performances

The hybrid model was tested on both the initial input set and the additional input set.
On the initial one, the model achieved an average speedup of 1.51 when run on 4 ranks (Fig. 16) on the same node. This result was

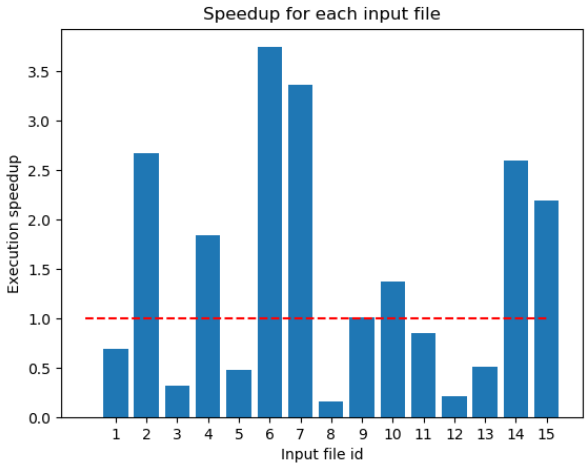

Fig. 16. Speedups of the hybrid model over the serial one, on initial input set

below expectations, and small files (such as 8) are the ones that have a bad impact on the performances index; however it is worth noting that big images such as 6 and 7 are processed 3 - 3.5 times faster than the original model.
We have then tried the model on the additional dataset, made by huge GIF files, on which the model yielded an average speedup of 3.10 when runned on 4 ranks (Fig. 18) and of 2.60 when ran on 2 ranks (Fig. 17) on the same node. This led to the conclusion that the model needs some additional tuning, since it performs well on big files but not on small ones.

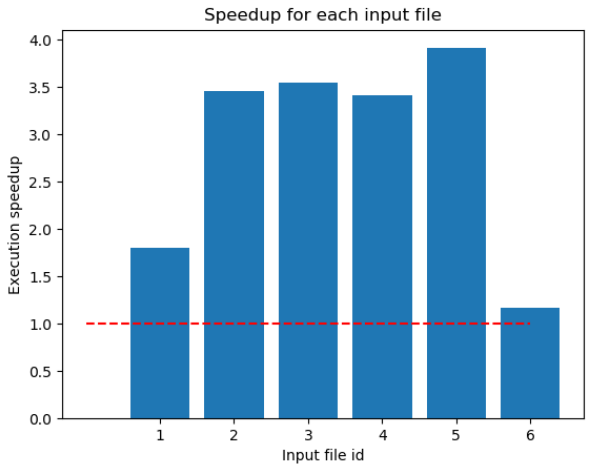

Fig. 17. Speedups of the hybrid model over the serial one, on the additional input set and with 2 MPI ranks

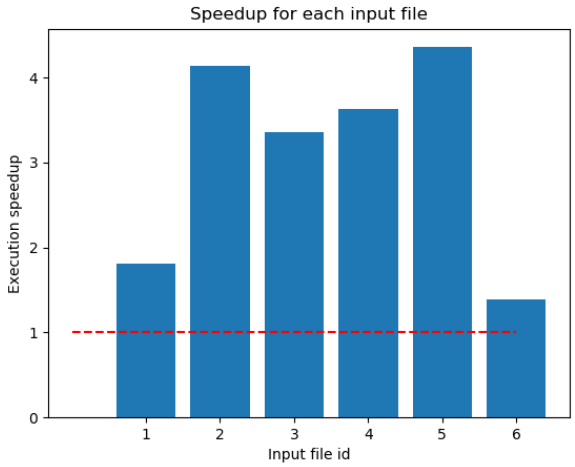

Fig. 18. Speedups of the hybrid model over the serial one, on the additional input set and with 4 MPI ranks

8 Conclusion

The objective of this project is to exploit parallel computing resources and improve the performance of the given program through three major APIs, namely MPI, OpenMP, and CUDA. The given program was the image filtering program. We implemented all the three APIs successfully and it led a lot of improvement in performance. In addition to that, each API usage was experimented with different models and optimized. Since every models had their advantages, and not every users have an access to these APIs and resources, we made an hybrid model, with the aim of selecting the best scenario with given environment and giving the best performances.

References

[1] OpenMPI documentation. *https://www.open-mpi.org/doc/*