Javeriana Cali University

Computer and Systems Engineering

Final Project

Parallel Programming

Authors:
Andrés Felipe Delgado and Ana María García

May 27 - 2020

1 Introduction

In the present project, the same algorithm used in exam 2 will be used, which consists of the parallelization of a program that works on the processing of bmp extension images to change their properties.

2 Description of the problem

The operation of this program has already been described in the report made in Part 2.

In this project, what will be carried out will be a toolbox where the user can choose between 6 possible functions to change the properties of an image that is entered as a parameter. The functions it can perform are as follows:

1) Curvecurve: This distorts the image.
2) Low: This divides the height of the image in half.
3) Fat: This doubles the width of the image.
4) High: This doubles the height of the image.
5) Thin: This halves the width of the image.
6) Opposite: This rotates the image 180 degrees.

In Exam 2, only Opposite was performed and this algorithm was parallelized. In this project these 6 functions will be available both sequentially and in parallel.

3 Functions

The result of each of the functions will be shown from function 1 to 6 from left to right.

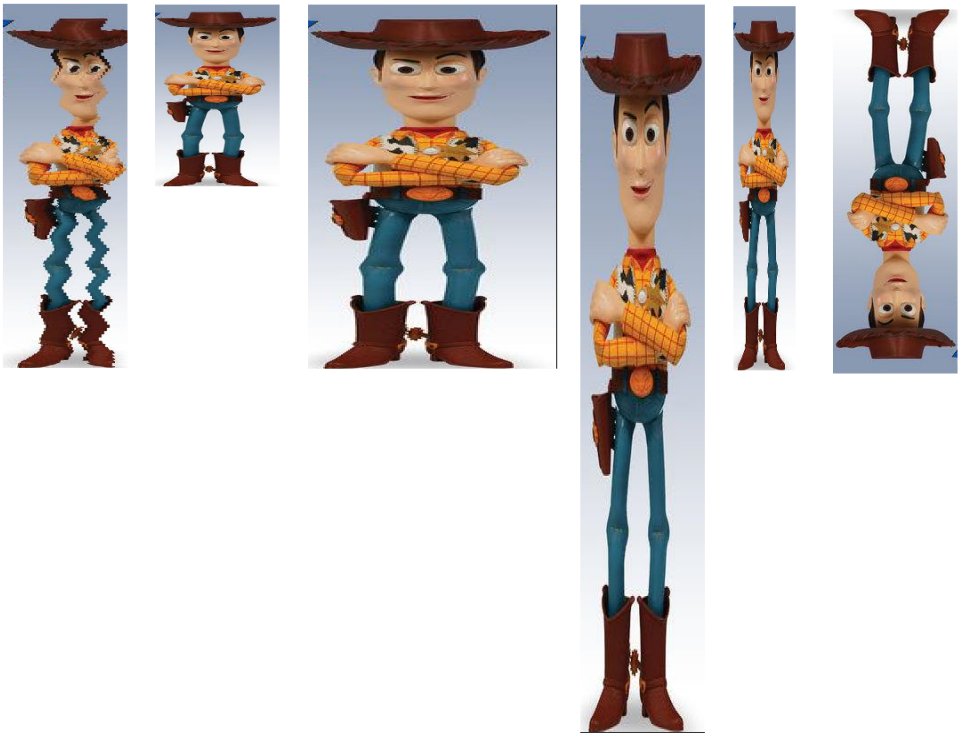

Figure 1: Functions Comparison

In order to run the program both parallel and sequentially, you must enter an input image as the parameter you want to modify, then an existing output image (it can be a copy of the input image) and the function number according to the enumeration shown above.

4 Secuential Execution

This code was modified compared to exam 2, since a new input parameter was added which indicates the function to be performed on the input image, joining the 6 functions in the same main.c file.

5 Parallel Execution

For parallel implementation, as in exam 2, we continued working with Pthreads using c.

In exam 2 we realized that the parallelization could be done by data, since the image that is entered as a parameter is divided by the number of threads

and a fragment is sent to each thread so that it can make the pertinent modifications. Finally, the result of the execution of each thread is joined and joined in the same image which will be the final result of the algorithm.

In this project, the parallelization is carried out in the same way as in exam 2, with the change that in this you must verify what function you want to perform on the image to know what algorithm to execute and also divide the threads that the user enters.

6 Results

The results of the execution with images of different sizes are shown below: NOTE: The image data is represented in pixels and the times in seconds.

6.1 Function 1: Curvecurve

Curvecurve	Image Sizes	Sequential	2 Threads	3 Threads	4 Threads	Speed Up
1	140x455	0,017	0,015	0,015	0,014	1,21428571
2	1000x1000	0,106	0,085	0,082	0,078	1,35897436
3	3000x3000	0,835	0,634	0,631	0,594	1,40572391
4	6000x6000	3,321	2,552	2,425	2,314	1,43517718
5	10000x10000	9,5	7,255	7,051	6,727	1,41221941
6	15000x15000	21,603	16,568	15,927	15,425	1,40051864

Figure 2: Function 1: Execution Times

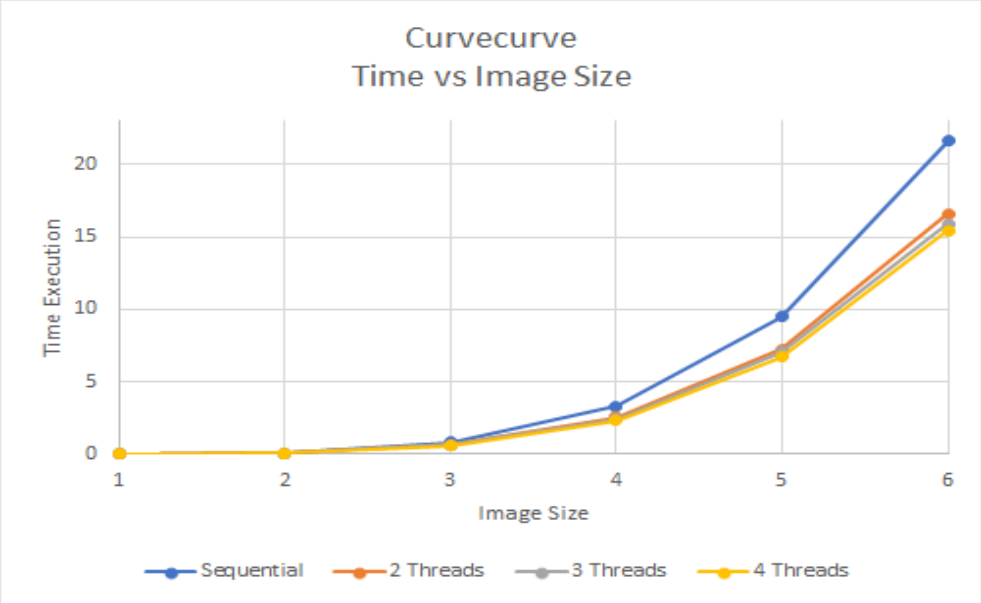

Figure 3: Function 1: Time vs Array Sizes

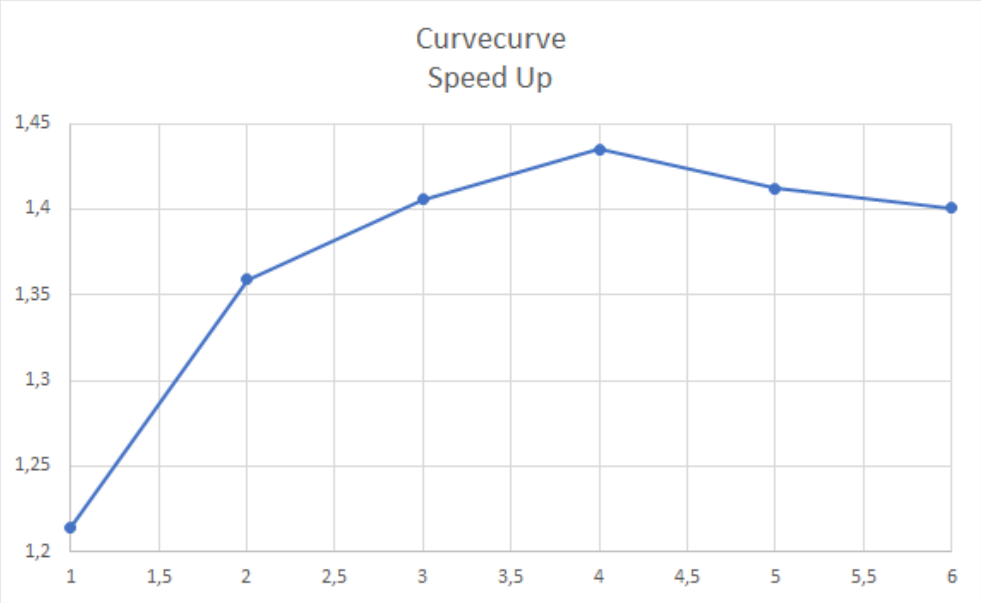

Figure 4: Function 1: Speed Up

6.2 Function 2: Low

Low	Image Sizes	Sequential	2 Threads	3 Threads	4 Threads	Speed Up
1	140x455	0,015	0,015	0,014	0,014	1,07142857
2	1000x1000	0,074	0,074	0,073	0,068	1,08823529
3	3000x3000	0,544	0,523	0,495	0,48	1,13333333
4	6000x6000	2,131	2,101	1,889	1,813	1,17539989
5	10000x10000	5,902	5,877	5,163	5,049	1,16894435
6	15000x15000	13,644	12,401	12,007	11,566	1,17966453

Figure 5: Function 2: Execution Times

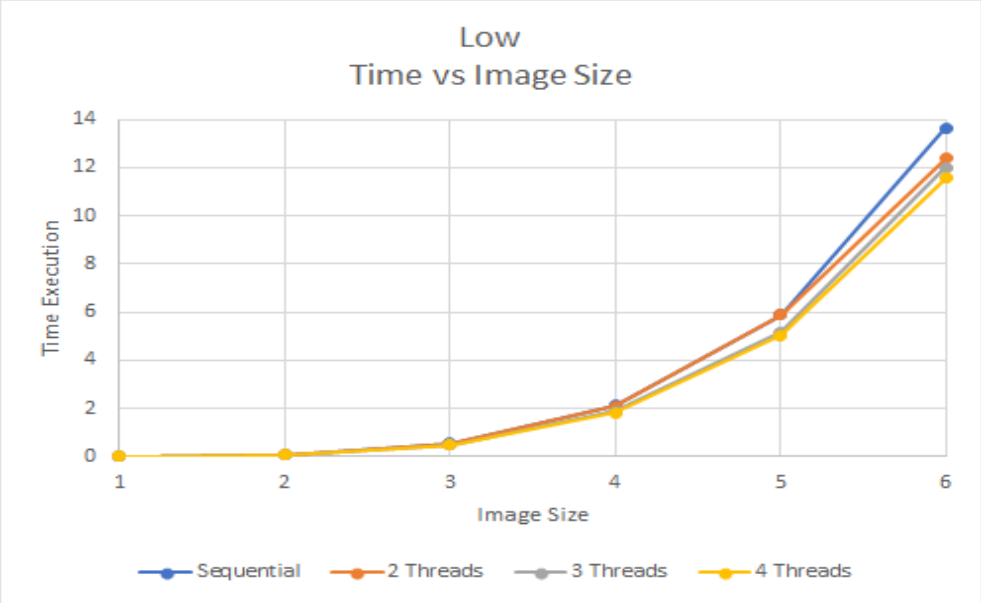

Figure 6: Function 2: Time vs Array Sizes

Figure 7: Function 2: Speed Up

6.3 Function 3: Fat

Function 3	Image Sizes	Sequential	2 Threads	3 Threads	4 Threads	Speed Up
1	140x455	0,022	0,018	0,016	0,015	1,46666667
2	1000x1000	0,171	0,127	0,125	0,117	1,46153846
3	3000x3000	1,415	1,009	0,984	0,941	1,50371945
4	6000x6000	5,743	4,183	4,131	3,844	1,49401665
5	10000x10000	16,427	12,112	11,772	11,135	1,47525819
6	15000x15000	34,777	25,689	25,179	22,989	1,51276698

Figure 8: Function 3: Execution Times

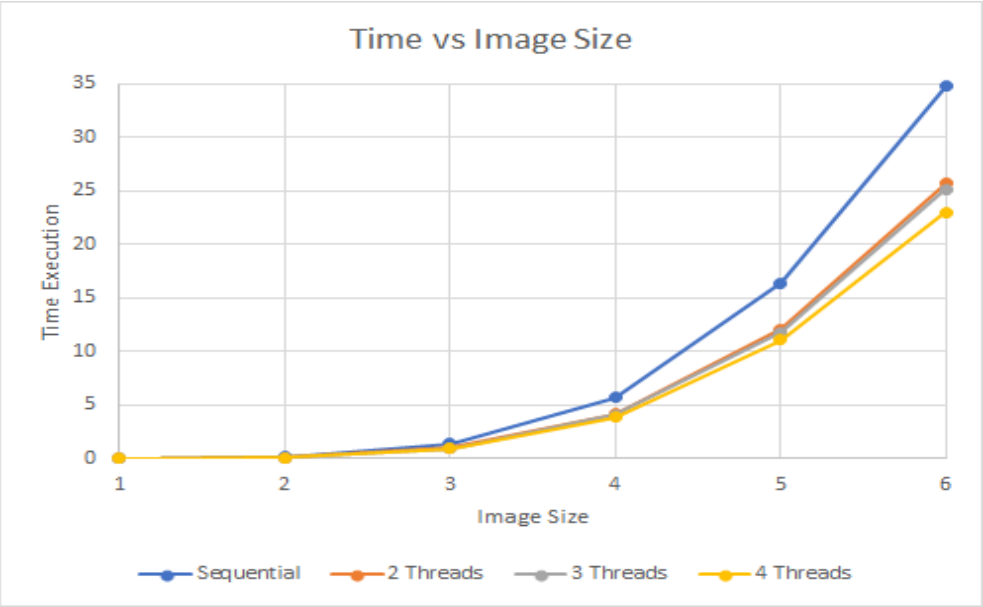

Figure 9: Function 3: Time vs Array Sizes

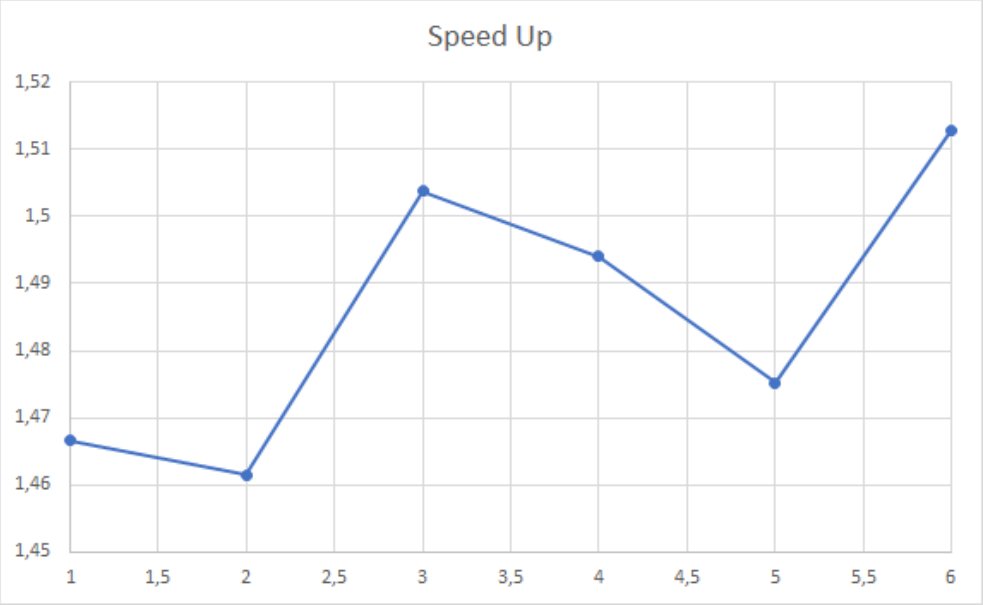

Figure 10: Function 3: Speed Up

6.4 Function 4: High

Function 4	Image Sizes	Sequential	2 Threads	3 Threads	4 Threads	Speed Up
1	140x455	0,022	0,022	0,02	0,019	1,15789474
2	1000x1000	0,17	0,146	0,143	0,141	1,20567376
3	3000x3000	1,404	1,214	1,127	1,16	1,21034483
4	6000x6000	5,762	4,901	4,715	4,69	1,22857143
5	10000x10000	15,574	14,31	13,738	12,997	1,19827653
6	15000x15000	35,268	31,037	28,942	28,719	1,22803719

Figure 11: Function 4: Execution Times

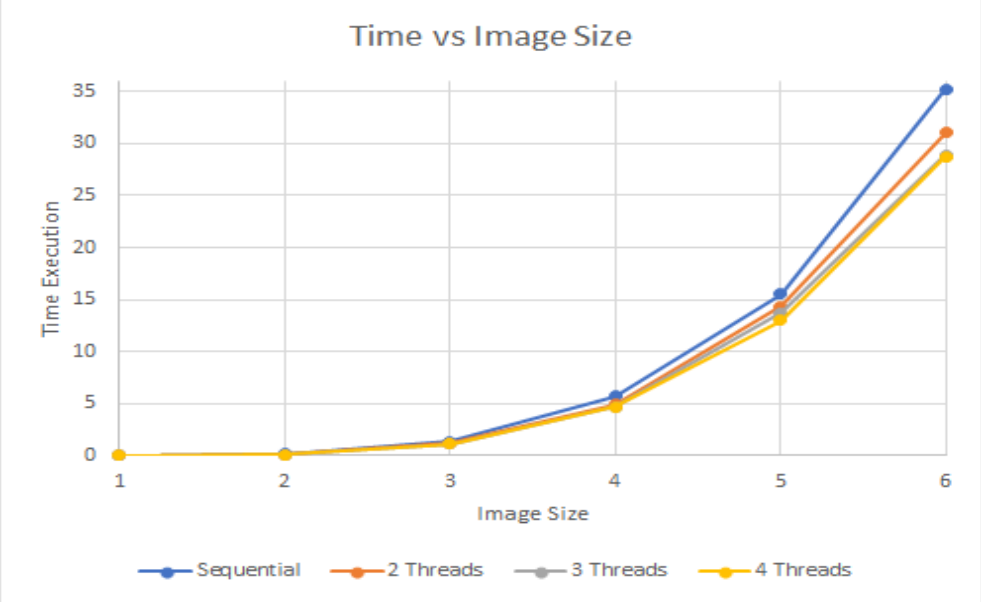

Figure 12: Function 4: Time vs Array Sizes

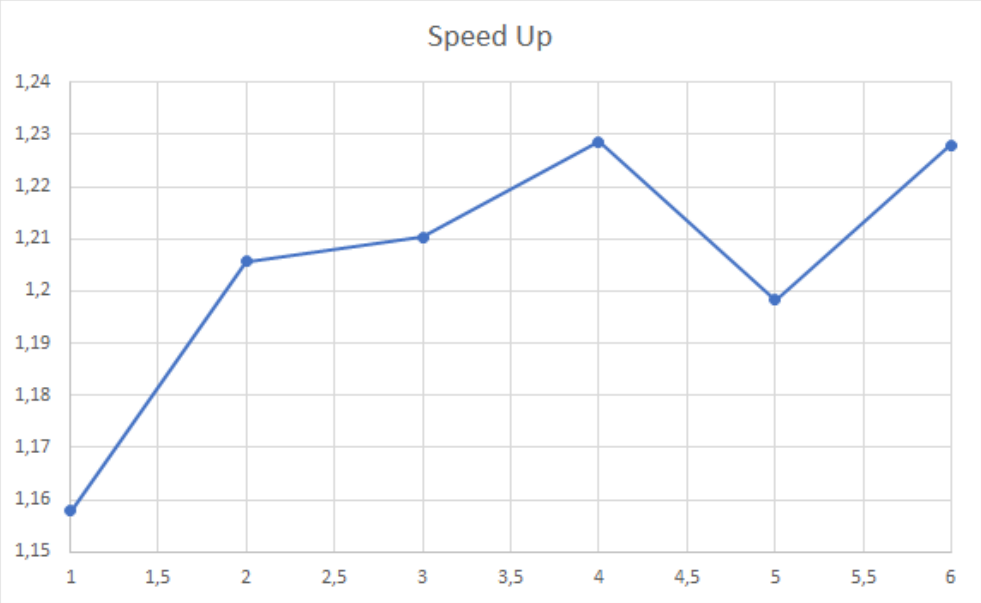

Figure 13: Function 4: Speed Up

6.5 Function 5: Thin

Function 5	Image Sizes	Sequential	2 Threads	3 Threads	4 Threads	Speed Up
1	140x455	0,014	0,014	0,013	0,013	1,07692308
2	1000x1000	0,073	0,064	0,063	0,062	1,17741935
3	3000x3000	0,55	0,451	0,438	0,422	1,30331754
4	6000x6000	2,138	1,73	1,689	1,639	1,30445394
5	10000x10000	5,889	4,809	4,645	4,517	1,30374142
6	15000x15000	13,721	11,293	10,889	10,599	1,29455609

Figure 14: Function 5: Execution Times

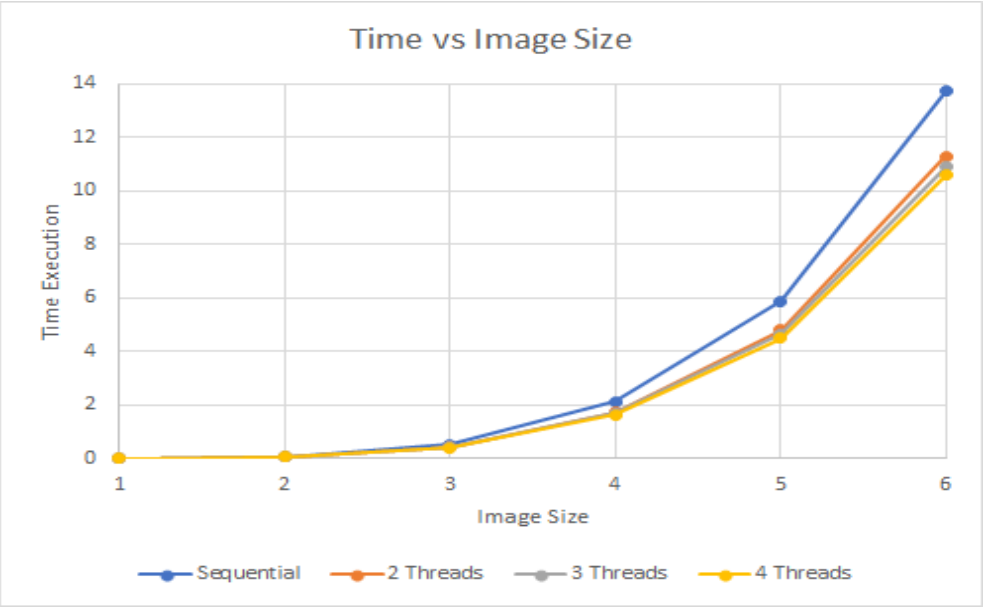

Figure 15: Function 5: Time vs Array Sizes

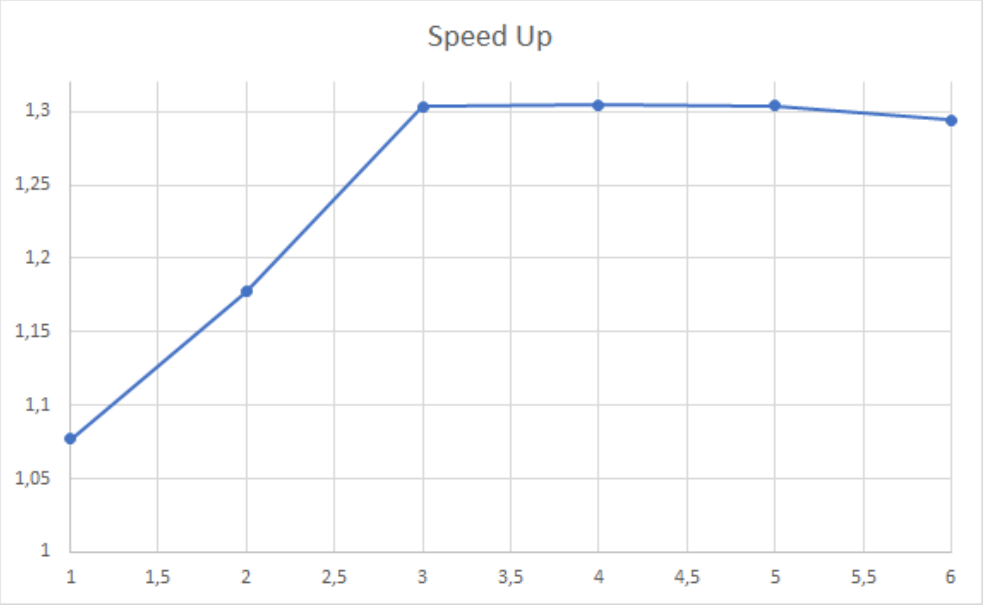

Figure 16: Function 5: Speed Up

6.6 Function 6: Opposite

Opposite	Image Sizes	Sequential	2 Threads	3 Threads	4 Threads	Speed Up
1	140x455	0,019	0,019	0,016	0,015	1,26666667
2	1000x1000	0,106	0,083	0,082	0,082	1,29268293
3	3000x3000	0,852	0,643	0,624	0,6	1,42
4	6000x6000	3,346	2,527	2,481	2,377	1,40765671
5	10000x10000	9,557	7,365	7,036	6,768	1,41208629
6	15000x15000	21,712	16,528	16,036	15,329	1,41640029

Figure 17: Function 6: Execution Times

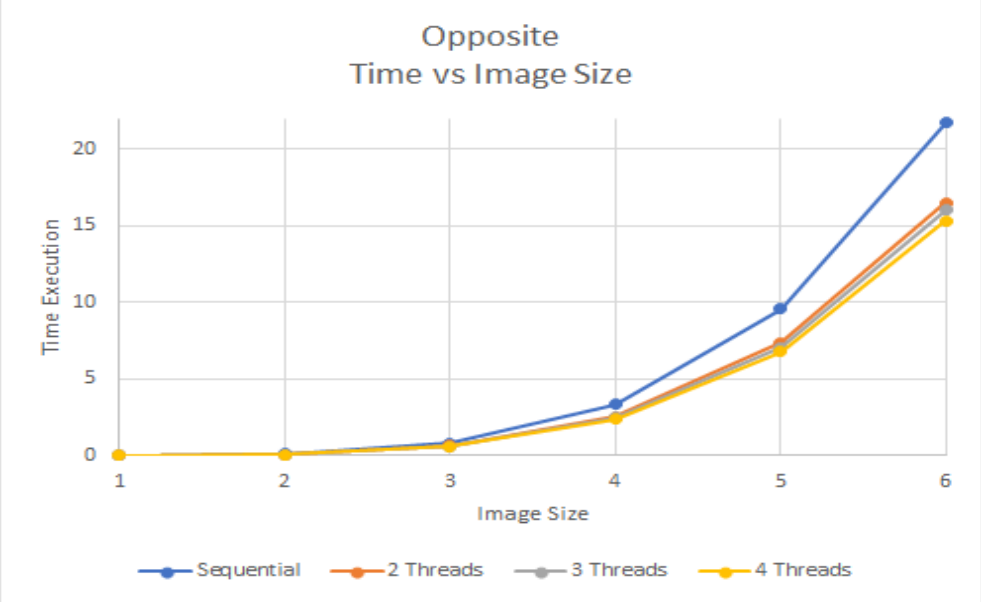

Figure 18: Function 6: Time vs Array Sizes

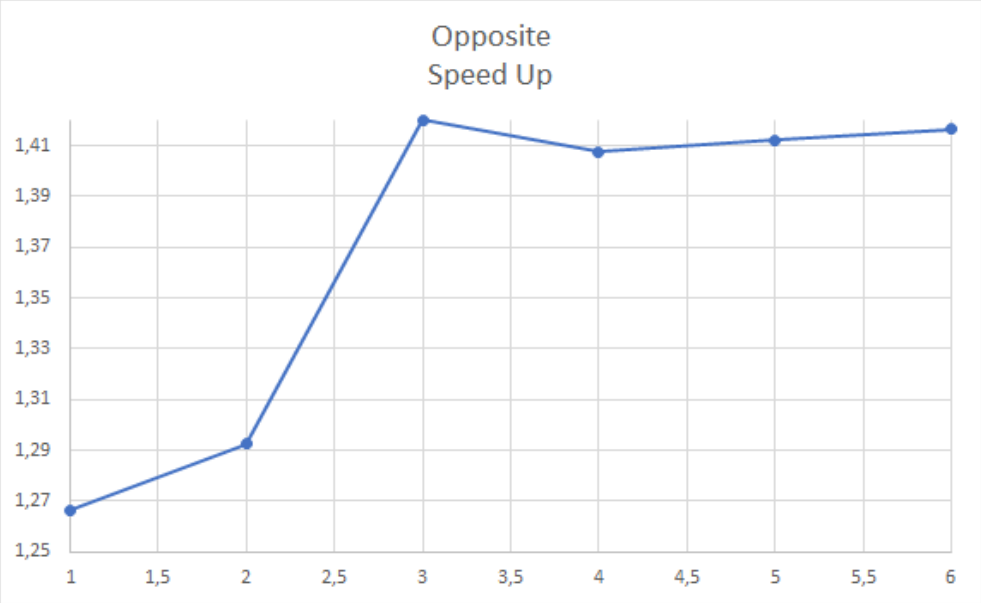

Figure 19: Function 6: Speed Up

7 Maximum Speed Up

The highest Speed Up obtained from the previous functions was 1,51.

8 Conclusions

- In the original program of the Github repository, 9 functions were included, of which 6 could be parallelized. This happened because the other 3 functions were transformations that had to be carried out sequentially in the image, starting from the top and continuing until get to the bottom. To program this with multiple threads you should wait for a thread to run first with the top of the image, then the second part thread, and so on. This would indicate that a thread could not start without the previous one having ended and passing the necessary information to continue processing the image. For this reason these functions could not be parallelized since each part of the image that was processed depended on a previous one.

- To carry out this project, it was helpful to have previously performed the opposite function in exam 2. In order to initially parallelize this function, we had multiple conflicts, but finally, the parallelization of a single function was achieved. Starting from this point, the implementation of the other

5 functions was not very complex since it was already known what were the key points for the parallelization of these algorithms.

- In this project we were able to realize that parallelization does not always work only by dividing data and executing them in parallel, but there are times when you have to look further and check the sequential code to know which functions take the longest and how these can be done to get better.
- We consider that this particular project was a very interesting experience, since it was possible to test the concepts seen in class in an applied algorithm and, in addition, we went beyond just the implementation of pthreads since the key to this parallelization worked was in two specific lines of code belonging to the source code and that we had to go deeper into how these worked in order to make a good program.
- Since in this project we work with a base code that is not ours, we put into practice the real cases where a parallelization engineer must improve and parallelize a code that is not his, because they must understand this code in order to improve it.