Performance oriented software project

Computer Architecture G88 - Team 5

Marina Torelli Postigo 100383479
Antonio Luengo Morillo 100405888
Álvaro Marco Pérez 100383382

Table of contents

Introduction	**3**
Sequential version	**3**
Parallel version	**4**
Performance evaluation	**5**
Sequential version	**6**
Parallel version	**7**
Impact of scheduling.	**9**
Performed tests	**9**
Conclusions	**11**

1. Introduction

In this project, students were asked to implement two versions of a program in C++ language, one that executed sequentially and another that did it parallelly.

The program was supposed to copy and transform BMP format photos in a given directory, by using two different filters, Gaussian blur and Sobel operator, for enhancing the edges of the elements in the frame, into a target directory.

Both versions of the program were needed to be first implemented and executed with compiler optimizations disabled and enabled. The parallelization of the program was performed with the help of OpenMP libraries.

For last, students had to carry out an analysis of the impact in the performance of both implementations and its execution with either the compiler optimizations enabled or not.

2. Sequential version

The problem was implemented using only code files. The algorithm was divided into four functions, following the scheme below:

- **main (int argc, char* argv[])** function receives as arguments the values written on the command line whenever it is executed. Performs tests on the number of arguments and it values. Depending on the argument in the *mode* position, the function will execute different algorithms
    - **copy** algorithm makes use of the standard library 'std' and makes a copy of every file in the given directory, into the target folder.
    - **gauss** algorithm reads the header of each file, extracts its dimensions and creates a byte data structure with every pixel and two extra bytes. Then it calls *'gaussianblur_2D'* and stores the result in the target directory.

- **sobel** algorithm follows the same structure as the gaussian algorithm, but before storing the files, the function '*sobel_filter'* is called.
-

- **gaussianblur_2D**(unsigned char * arr, unsigned char * result, int width, int height) function receives as parameters the dimensions of the image (which will be used in loops as boundary) '*arr*' as an the input image and '*result*' as the output image. For every row and column of the image it will call the function '*accessPixel*' that will perform the algorithm described in the gaussian blur formula.
  - **accessPixel** (unsigned char * arr, int col, int row, int k, int width, int height)function will take the input parameters and compute with them the operations in the Gaussian blur formula.

$$res(i,j) = \frac{1}{w} \sum_{s=-2}^{2} \sum_{t=-2}^{2} m(s+3, t+3) \cdot im(i+s, j+t)$$

- **sobel_filter**(unsigned char * arr, unsigned char * result, int width, int height) function will take every pixel in the '*arr*' structure, using loops with boundaries '*height*' and *width*', and call the function '*accessPixelSobel*' and storing the result of the operation in the '*result*' variable.
  - **accessPixelSobel** (unsigned char * arr, int col, int row, int k, int width, int height) function will take the input parameters and compute with them the operations in the Sobel operator formula.

$$res(i,j) = |res_x(i,j)| + |res_y(i,j)|$$ that come from,

$$res_x(i,j) = \frac{1}{w} \sum_{s=-1}^{1} \sum_{t=-1}^{1} m_x(s+2, t+2) \cdot im(i+s, j+t)$$

and

$$res_y(i,j) = \frac{1}{w} \sum_{s=-1}^{1} \sum_{t=-1}^{1} m_y(s+2, t+2) \cdot im(i+s, j+t)$$

The program will run on the provided mode, create the files, (if the input parameters are correct), and print the time of computation.

3. Parallel version

For the implementation of the parallel program we needed to use the OpenMP library in C++ language.

In the developed sequential version of the program, the global algorithm included four different iterating loops. In the '*main*' function, the first two loops iterated through the files that were on the source directory, one in the '*gauss*' algorithm and the other in the '*sobel*' one. The remaining loops were on functions '*gaussianblur_2D*' and '*sobel_filter*', both iterated through every pixel of the given image, in every information channel.

Our first approach was to parallelize all the loops in the algorithm. We included static loop scheduling option of the OpenMP library, but when executing the program, the operational output time was greater than in the sequential implementation. We realized that the threads created in the main function were also creating threads on the called functions in its algorithm, (for example, when executing the program in '*sobel*' mode, the created threads will create more when calling the '*gaussianblur_2D*' function and the '*soble_filter*' function), this decreased the performance of the execution.
Consequently, we decided to modify the parallelization approach we had made on our first try and implementing the thread creation only in the loops in the '*gaussianblur_2D*' and '*sobel_filter*' functions. With slight modification we finally achieved an optimized implementation that improved the sequential program.

Regarding racing conditions, as every thread is manipulating different parts of the data structure of the pixel in the image, there should not be any problem. The same scenario comes when writing in memory, each iteration .

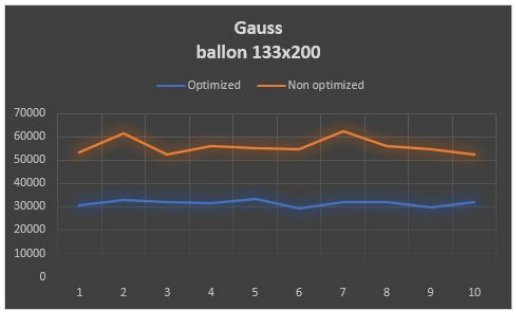
4. Performance evaluation

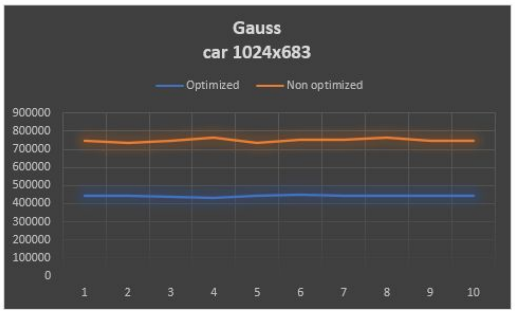
The computer that was used for the performance evaluation had the following software version and hardware specifications::

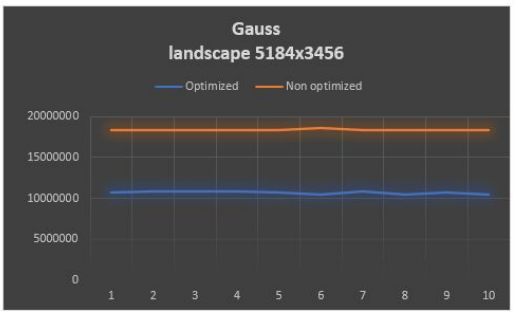
Processor model	Intel Xeon E5-2630
Number of physical cores	4
Number of logical threads	8
Cache memory characteristics	**L1** 4x32KB 8-way set associative instruction cache 4x32KB 8-way set associative data cache **L2** 4x256KB 8-way set associative cache 6x32KB 8-way associative instruction cache **L3** 15MB 20-way set associative shared cache
Operating system	Ubuntu 18.04.3
Compiler version	GNU C++ 4:7.4.0

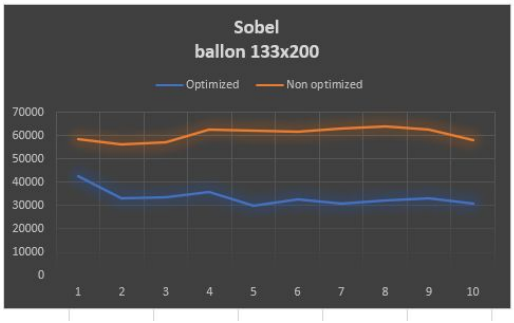
a. Sequential version

For testing the performance of the sequential version, we ran the program several times with the same input images with the purpose of having a clear view of the behaviour of the program, minimizing the circunstancial performance variations.



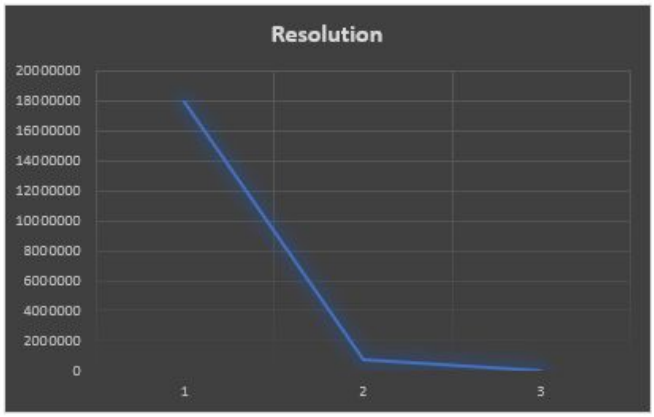


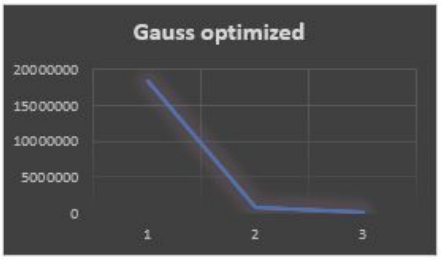


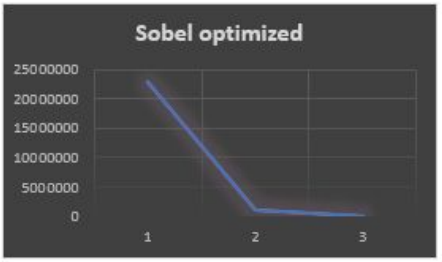


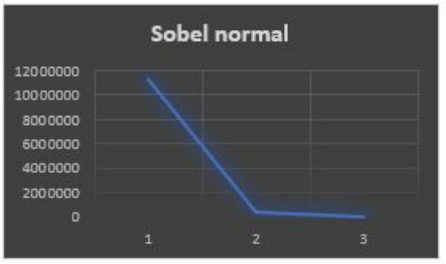
In the figures we can see the time spent in picoseconds (y axis) in each of the executions of the programb (x axis), corresponding to a low, middle and high resolution image of the database, and two of the modes of execution. It can be clearly seen that the execution under the compiler optimizations decreases in an average of 84,03% in the execution time.

Taking into account the increase of the size in the images, and the average execution time of each one, we can deduce that the time complexity of the sequential program is O(n), as it can be seen on the figure below. The increasement in time is directly proportional to the increasement in the resolution, making the implementation linearly complex.



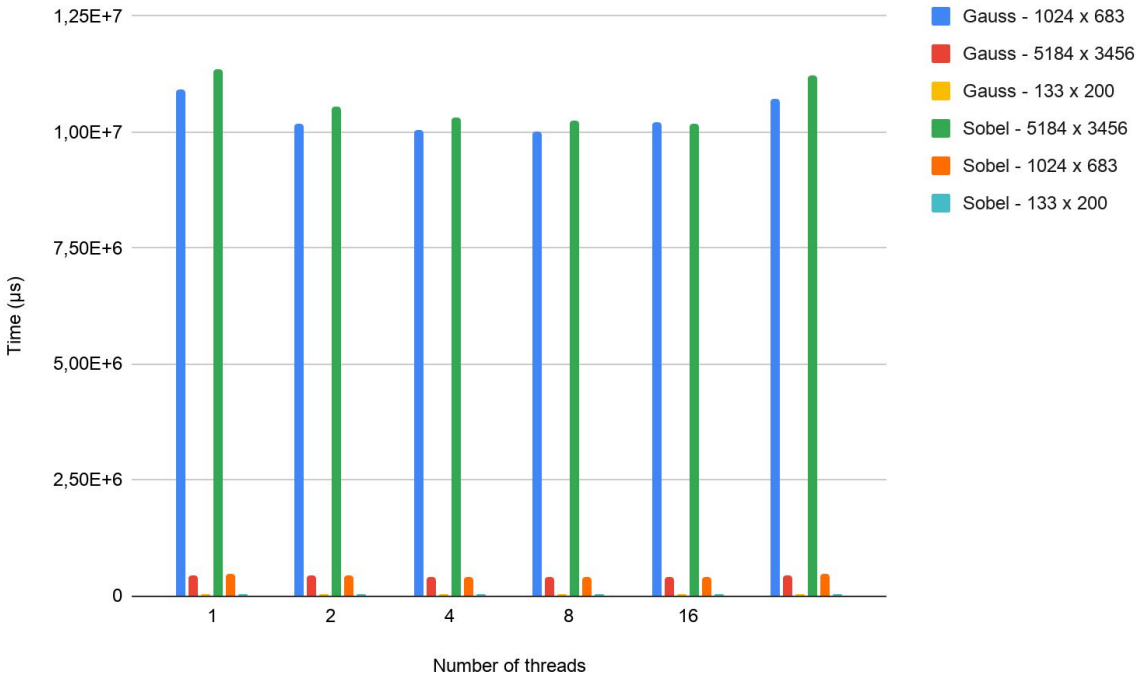




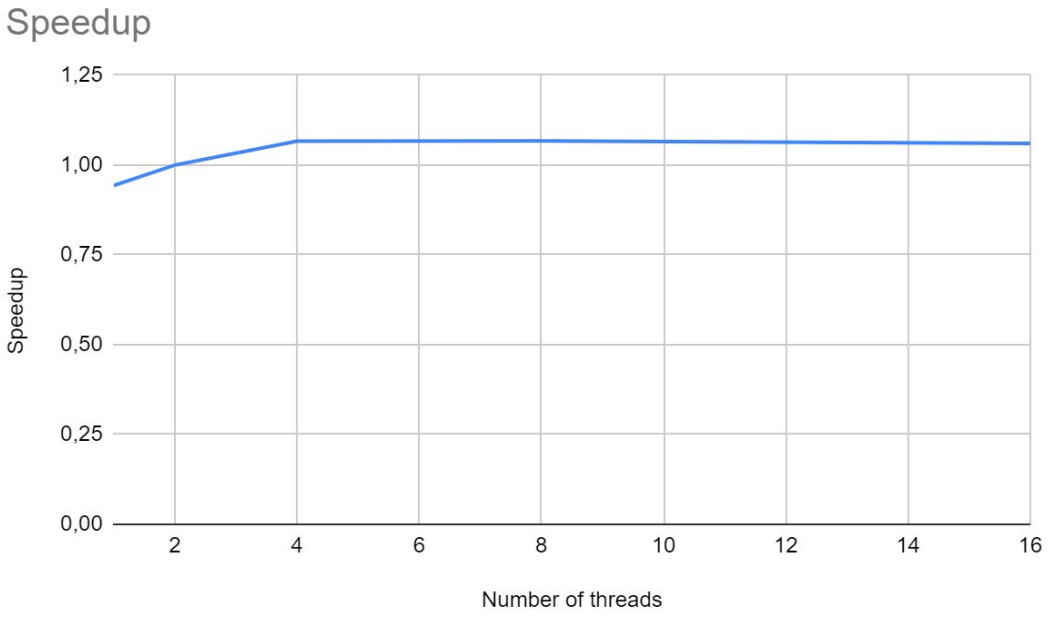


b. Parallel version

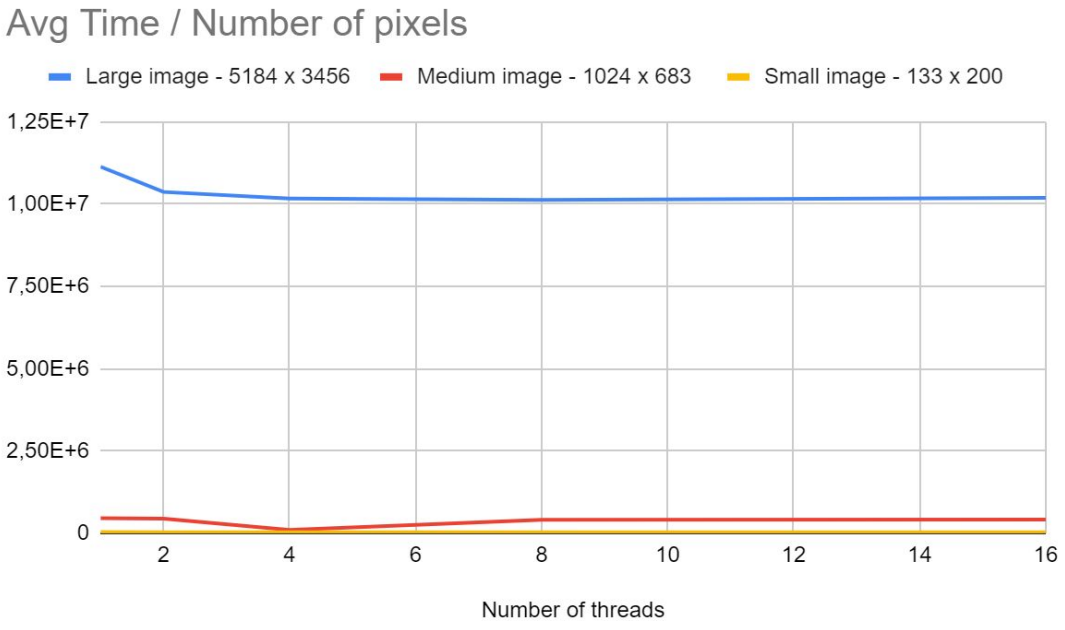
For the performance tests of the parallel implementation of the program we executed and compared three images with different resolutions, in the '*gauss*' and '*sobel*' mode and different number of threads, with the results shown in the figure below.



We can see that the execution with only one thread is equivalent to the sequential execution (the right most), and that the execution time decreases directly with increase of the threads running, until we reach the system physical limit. When the program runs in more threads than the ones logically in the processor, there is no increase in the performance, as the remaining threads need to be dynamically allocated. We can see that there is a slight upturn among the use of the 1, 2 and 4 threads, and then the improvement becomes stagnant. As it can be seen in the following figure.



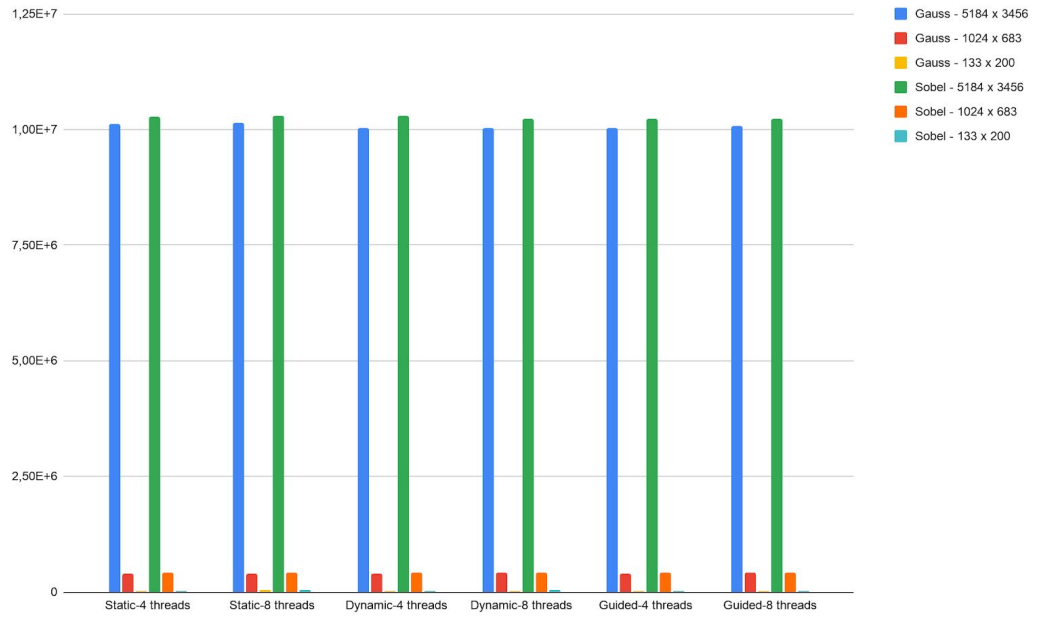
We can also calculate the average time per pixel, so we can observe how images of different sizes take different times to process while using the parallel version of the program.



We can see that the average time decreases as the number of threads increases up to 4, which is the number of physical cores of the computer. After that the time per pixel doesn't get any lower.

c. Impact of scheduling.

As our implementation of the parallel program resides in threading on iterating loops that implement both Gaussian blur formula and Sobel filter formula computations. All the times the loop is executed, the formula is computed, making the variations in the schedule modes barely change, as it can be seen in the figure below.



5. Performed tests

To check whether the implemented program followed the requirements set in the statement of the project, we made use of the binary provided. First we checked the program using boundary testing technique, consisting in providing invalid input parameters to the program when executing it:

Description	Expected behaviour	Binary execution	Developed execution
Less parameters than the expected ones	Error message	Error message	Error message
More parameters than the expected	Error message	Error message	Error message
Parameter #1 non existing directory	Error message	Error message	Error message

Parameter #1 no read permissions on directory	Error message	Error message	Error message
Parameter #1 no write permission on directory	Normal execution as program does not write on this directory	Normal execution	Normal execution
Parameter #1 no execute permission on directory	Normal execution as program does not execute on this directory	Normal execution	Normal execution
Parameter #1 correct	Normal execution	Normal execution	Normal execution
Parameter #2 non existing directory	Normal execution	Error message	Error message
Parameter #2 no read permissions on directory	Error message	Normal execution	Normal execution
Parameter #2 no write permission on directory	Error message	Normal execution, but does not write on the output directory	Normal execution, but does not write on the output directory
Parameter #2 no execute permission on directory	Normal execution as program does not execute on this directory	Normal execution	Normal execution
Parameter #2 correct	Normal execution	Normal execution	Normal execution
Parameter #3 mode not valid	Error message	Error message	Error message
Parameter #3 mode correct	Normal execution	Normal execution	Normal execution
Images on directory not in BMP format	Normal execution, but error message with the different type file	Normal execution, but error message with the different type file	Normal execution, but error message with the different type file
Different types of files in the source directory	Normal execution, but error message with the different type file	Normal execution, but error message with the different type file	Normal execution, but error message with the different type file
Already existing files on the target directory	Normal execution, does not affect those files	Normal execution	Normal execution

As it can be seen in the table above, the tests performed were successfully completed as the provided binary and the expectations set at the beginning, excepting the test that checked the behaviour of the program whenever the output directory does not have write

permissions. The program runs without throwing any exception, but there is no file in the desired directory.

6. Conclusions

During this assignment we have realized that the parallelization of programs has a huge impact in performance. In previous courses, like Operating Systems, the use of threads was messy and complex, but with OpenMP library it was straightforward, making it more accessible and less time consuming.

We also realized the compiler's optimizations advantages. Having a consistent compiler will make programs run faster and waste less resources, making the technology more accessible and democratic.