Project 2: Parallel Image Convolution using OpenMP

Diana Laura Osorio Soto
156084
diana.osorioso@udlap.mx

Ingrid Jazmín Guerra Martínez
155964
ingrid.guerramz@udlap.mx

Alejandro Lobato Molina
154422
alejandro.lobatoma@udlap.mx

***Abstract*—Image Processing involves applying functions or operations on images to obtain their transformations. This process requires many computing resources, mainly time and space. For this reason, dividing the process in a parallel machine of distributed memory is intended to reduce execution time and excessive memory access.**

Based on the previous work, this document presents the new development of a parallel implementation of Image Convolution in C language in order to compare its performance against the sequential one used for the Project 1.

I. Introduction

Nowadays multiple applications have emerged for parallel computing, from optics, computer science, physics and computer vision. As parallel processing has become a significant tool for implementing high speed computing, computer vision and image processing applications have been improved significantly.

An image can be regarded as a function *f (x,y)* of two continuous variables *x* and *y* (Fig. 1). To be processed digitally, it has to be sampled and transformed into a matrix of numbers. Since a computer represents the numbers using finite precision, these numbers have to be quantized to be represented digitally. Digital image processing consists of the manipulation of those finite precision numbers [17].

The processing of digital images can be divided into several classes: image enhancement, image restoration, image analysis, and image compression. The scope for this project is focused on image analysis techniques that permit an image to be processed so that information can be automatically extracted from it [3]. To do so, for this project an edge detection operation will be performed.

Image filtering is a technique that allows to apply various effects on photos through convolution operations.

Convolution is described by a rectangular matrix of real coefficients called **kernel** convoluted in a sliding window of image pixels. In other words, it has a 2D filter matrix and the 2D image.

$$[f*g](n) = \sum_{i=-\infty}^{\infty} f(i)g(n-i) = \sum_{i=-\infty}^{\infty} g(i)f(n-i)$$

Fig. 1. Discrete Convolution.

So, as following the Discrete Convolution formula shown in Fig. 1, we can define that for every pixel of the image, take the sum of products. Each product is the color value of the current pixel or a neighbor of it, with the corresponding value of the filter matrix. The center of the filter matrix has to be multiplied with the current pixel, the other elements of the filter matrix with corresponding neighbor pixels [1].

As analyzed in the previous project, it often takes more time than that available to execute several processing operations into a single image, for example point to point processing of a gray scale image of size 1024 X 1024 requires a CPU to make more than one million operations, for color images or even high resolution images this problem increases as it is multiplied by the total number of channels [2]. The 2D convolution operation requires a 4-double loop, so it is not extremely fast, unless small filters are used.

In this project the algorithm proposed for applying image convolution in C language will now be suited to convert it to a parallel version in which the processes will be divided into subroutines per thread to evaluate the performance of its application within three different Operating Systems with different computer architectures.

The document is separated in five sections, in Section II the description of the problem this project will approach is presented. Section III presents the architecture of the equipment used to execute the program for this project, three different computers were used to execute it.

After this comes the methodology in Section IV which explains the steps for the convolution implementation now implementing its parallel version.

In Section V, the processing times are shown and compared with the ones obtained from the previous practice. Finally, the conclusions stated in Section VI present the knowledge obtained in the development of the project and the problems faced through its implementation and design.

II. Description of the problem

The problem to solve during the development of this project is to analyze the algorithm presented in the previous project and detect sections which may become parallel. As mentioned before, an algorithm to apply a convolution kernel was developed during the first project; this, in order to perform Image Analysis for edge detection. Convolution is a simple mathematical operation which is fundamental to many common image processing operators [13]. Convolution is performed by multiplying and accumulating the instantaneous values of the

overlapping samples corresponding to two input signals, one of which is flipped [14]. In order to perform the analysis of the algorithm some concepts need to be reviewed to comprehend how the algorithm could be best modified to provide a successful result. Such components are described in further detail below.

A. Speedup and Amdahl's Law

First, it is important to know the efficiency of the algorithm presented for this problem. The speedup of a parallel algorithm is the ratio of the compute time for the sequential algorithm to the time for the parallel algorithm [12].

In parallel computing, Amdahl's law is mainly used to predict the theoretical maximum speedup for program processing using multiple processors [18].

The speedup equation according to Amdahl's law is as shown below.

$$speedup = \frac{1}{s + \frac{P}{n}} \tag{1}$$

Where:

- *P* is the parallelized fraction of the code.
- *N* is the number of processors used.
- *s* is the serial fraction of the code $(1 - P)$.

B. Strong scaling vs. weak scaling

Scalability or *scaling* is used to indicate the ability of hardware and software to deliver greater computational power when the amount of resources is increased. It may also be referred as parallelization efficiency, meaning the ratio between the actual speedup and the ideal speedup obtained when using a certain number of processors [7]. It is divided in *weak* and *strong* scaling.

1) Strong scaling: Strong scaling is based on Amdahl's law; this states that, for a fixed problem, the upper limit of speedup is determined by the serial fraction of the code [7].

2) Weak scaling: Weak scaling is based on Gustafson's law; this law is based on the approximations that the parallel part scales linearly with the amount of resources, and that the serial part does not increase with respect to the size of the problem. With this law, the speedup, calculated based on the amount of work done for a scaled problem size, increases linearly with respect to the number of processors [7]. The speedup equation for Gustafon's law is the following:

$$scaledspeedup = s + P * N \tag{2}$$

Where, as well as in Amdahl's law:

- *P* is the parallelized fraction of the code.
- *N* is the number of processors used.
- *s* is the serial fraction of the code $(1 - P)$.

C. Using OpenMP and the C language

As mentioned before, the project's objective is to implement a parallelized version of image convolution for Edge detection using C/C++ language. In order to do this, *OpenMp* is used. It is a scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications on platforms ranging from embedded systems to multi-core systems and shared-memory systems [9]. *OpenMP* consists of a set of compiler *#pragmas* which control how the program works; in order to compile the pragmas, it is necessary to add the *-fopenmp* flag, as well as adding the *omp.h* library. To indicate which part of the code one would like parallized, it is necessary to add *#pragma* lines so that the compiler may interpret them and become a parallel program [19].

D. Sequential Section

Reviewing the previous implementation the sequential section of the program for this version belongs to assignments and declarations of the variables that were used for the program. This is, the inclusion of libraries, loading and writing the image and creating timestamps. It is important to identify the variables that will be used within the parallel section. This is explained in further detail in the section V. Overall, the goal is to identify the key components of the program that will be used later by the team of threads created.

```
declare STBI constants;
include C libraries;
include STBI libraries;
include OpenMP libraries;
imageParameters ⟵ STBILoad();
allocateOutputData();
kernel ⟵
 {-1, -1, -1, -1, 8, -1, -1, -1, -1};
initIndeces();
initTimestamps();
startTime(&start);
Parallel section... stopTime(&end);
timeEllapsed ⟵ end - start;
STBIWritePNG(outputData);
freeMemory(imageBuffer);
freeMemory(outputData);
```

Algorithm 1: `Sequential section`

E. Parallel Section

To identify the section of the code that could be parallel, we applied a certain criteria by answering the following questions:

- Which are the different code blocks within the program?
- Which parts of said blocks are declarations or assignments?
- Which components are operations or function calls to other operations?

Although these may seem quite trivial, they allowed us to properly identify possible parallel regions in the program. By delimiting the scope of each block, we can further focus in its individual components. This leads us to divide the program between sequential and parallel regions. Algorithm 2 describes the portion of the code that became parallel and conforms to the openMP spec.

Input: $(data, kernel)$, image data stream and the operation kernel
Result: $(outputData)$, the result of the operation

```
newPixel=0;
for (pixel in image) do
    center = i;
    newPixel ⟵
     multiply(data[center], kernel);
    centerLeft = i-1;
    newPixel ⟵
     multiply(data[centerLeft], kernel);

    centerRight = i+1;
    newPixel ⟵
     multiply(data[centerRight], kernel);

    top = (i-width);
    newPixel ⟵
     multiply(data[top], kernel);
    topLeft = (i-width) - 1;
    newPixel ⟵
     multiply(data[topLeft], kernel);
    topRight = (i-width) + 1;
    newPixel ⟵
     multiply(data[topRight], kernel);
    bottom = (i+width);
    newPixel ⟵
     multiply(data[bottom], kernel);
    bottomLeft = (i+width) - 1;
    newPixel ⟵
     multiply(data[bottomLeft], kernel);

    bottomRight = (i+width) + 1;
    newPixel ⟵
     multiply(data[bottomRight], kernel);

    outputData[i] ⟵ newPixel;
end
return outputData;
```

Algorithm 2: `Parallel section`

III. Architecture

There were three different computers used to test this project, each computer had different hardware and software characteristics. These characteristics can be seen in Table I.

TABLE I
Computer Architectures

PC	Operating System	Processor	Memory
A	macOS Catalina 10.15.3	2.9 GHz Intel Core i5, 2 cores	8 GB of RAM
B	Ubuntu 18.04.3	AMD Radeon r6, 4 cores	12 GB of RAM
C	Windows 10 Home	2.20 GHz Intel Core i7, 6 cores	16 GB of RAM

IV. Methodology

The program works with 3 variants of the same picture at 256 by 256 pixels, 800 by 800 pixels and 2048 by 2048 pixels each. On the largest, we are working with an effective 4,194,304 pixels. When reading input data, we allocate 32 MB of memory through the creation of a pointer of unsigned characters through the library's loading function.

Depending on the complexity of the operations performed, this may or may not seem like an appropriate problem size. Said operations consist of simple sums and multiplications. Further, the algorithm developed only performs the operations it requires given certain criteria is met. This means that only the necessary operations are performed for the computation of a new pixel value.

Research on the openMP specification provided hints towards changes that would yield significantly better results. To illustrate, Listing 1 shows the first implementation of the parallel region and Listing 2 shows the changes performed.

```
#pragma omp parallel num_threads(4)
   {
      #pragma omp for
      for(i = 0; i < dimensions; i++){
        ...
      }
   }
```

Listing 1. First parallel implementation

```
#pragma omp parallel num_threads(4) shared(kernel,
    data, output_data)
   {
      #pragma omp for private(i,topLeft, top,
    topRight, centerLeft,centerRight, bottomLeft,
    bottom, bottomRight)
      for(i = 0; i < dimensions; i++){
        ...
      }
   }
```

Listing 2. Second parallel implementation

In order to develop this project, three main steps were performed. These steps were:

- Obtaining an image in three different sizes (small, medium, and large). This images should be in JPEG format.
- After obtaining the images, the program used to perform image convolution is developed. This program had to include a time count for the convolution process.

- Finally, once the convolution times of each image were obtained, these times were graphed in order to compare the time differences between them.

The Convolution process starts with defining the size of the Kernel, for example to take a 3x3 array of numbers and multiply it point by point with a 3x3 section of the image. After defining the size of the kernel it goes sliding onto the image, multiplying the corresponding elements and then add them in the center point of the image. This procedure is repeated until all values of the image have been calculated.

Convolution of an image by a matrix of real numbers can be used to sharpen or smooth an image, depending on the matrix used. If A is an image and K is a convolution matrix, then B, the convolved image is calculated as:

$$B_{y,x} = \sum_i \sum_j A_{y+i,x+j} K_{i,j}$$

Fig. 2. Convolution formula.

If k is a convolution vector, then the corresponding matrix K is such that $K\ i,j$ = **ki kj**

The goal is to create a **conforming program** that compiles according to the openMP spec on restrictions. These restrictions on directives need to be met for a program to conform to the OpenMP model [10].

V. Results and Analysis

It could be distinguished two main ideas behind parallel optimization. First, the need to work with a reasonably sized problem. This means that the amount of data processed by the program has to be large enough for it to make sense that it is being parallelized. Second, the operations performed in the program should be, in some degree, more complex than usual.

Applying these couple of principles to the results obtained, it could be observed why the initial results may seem counter-intuitive. It was initially expected that the parallel region in the *for* statement would reduce the time of execution. However, the results differed greatly from this expectation. It was believed that the increase in time was due to overhead of thread creation and distribution of tasks among thread teams.

Figure 3 shows the output of parallel execution without the usage of the directives.

Fig. 3. Race condition caused pixels to be written randomly.

Comparing this with the expected outcome for the 3 pictures, shown in Figs. 4, 5 and 6. A better definition of the borders in pictures with lower resolutions can be appreciated, due to the density of pixels in the field of view.

Fig. 4. Border detection of the 256 pixel variant.

Fig. 5. Border detection of the 800 pixel variant.

Fig. 6. Border detection of the 2048 pixel variant.

The openMP documentation mentions possible race conditions arise when working with multiple references to different memory spaces take place [5].

To fix this, the openMP directives `shared` and `private`, had to be used.

The private directive tells the team of threads that each variable specified within it holds a unique copy of them. On the other hand, the shared directive tells the team that there's a common, shared space of memory that they will use and write to [4].

Table II is an example of the differences before and after adding the openMP directives. The times shown belong to the macOS system mentioned in table I, applying an edge detection kernel over a 2048x2048 PNG image.

TABLE II
TIME DIFFERENCE USING OPENMP DIRECTIVES

Threads	AVG time without directives	AVG time with directives
1	134,928 μs	118,071 μs
2	353,470 μs	80,782 μs
3	406,450 μs	77,826 μs
4	496,363 μs	72,239 μs

Note: Time is being measured in microseconds due to the algorithm being faster than what previously expected. Using native time constructs or functions often returned unexpected time outputs.

To conclude the analysis, a comparison is made between the speedup of each system mentioned in Table I when running the system 5 times for each image and up to 4 execution threads.

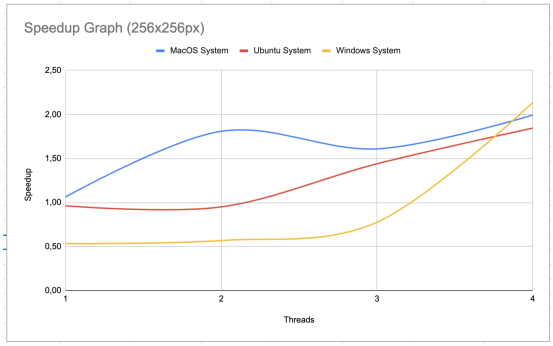

Fig. 7. Graph for an image of Lena of 256 by 256 pixels.

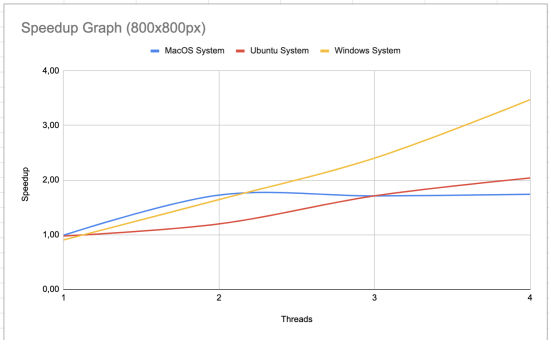

Fig. 8. Graph for an image of Lena of 800 by 800 pixels.

It could be observed that speedup starts to behave as expected when the size of the problem increases. This is an example of what we discussed earlier. A problem that's set to become parallel in some manner, must meet certain criteria for it to be feasible. Commonly, problems tackled using parallel computing meet this requirements.

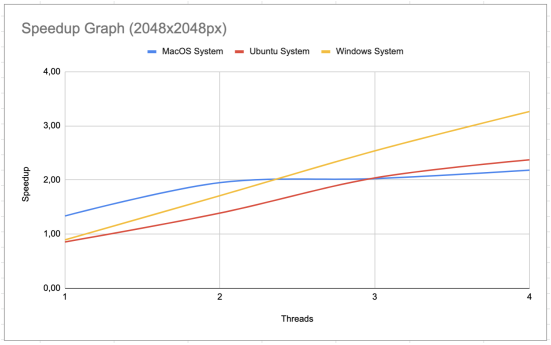

Fig. 9. Graph for an image of Lena of 2048 by 2048 pixels.

It is interesting to note that the Windows and Ubuntu systems seem to perform almost linearly for pictures of 800 by 800 and 2048 by 2048 pixels.

VI. Conclusions

As discussed previously, the area of image processing is of great interest to researchers from different fields given its performance, suitability for several analyses and availability of resources for its implementation. It can be used in different applications of image processing on the basis of its appropriateness, performance, computational cost and applicability.

One of the challenges faced during the project's life cycle was processing images with C without using a predefined library. After some discussion with our peers, a suitable library was found to handle this task. Another challenge was already mentioned in section V, where the program appeared to perform poorly when teams of threads are created. We would usually stop at this point, but we were interested in seeing why this was happening. This led to understanding better how OpenMP works and allowed us to provide a stark contrast between the initial stages of the project and its conclusion.

References

[1] Chaudhuri, S., 2010. [online] Graphics.stanford.edu. Available at: ¡http://graphics.stanford.edu/courses/cs148-10-summer/docs/04_imgproc.pdf¿ [Accessed 9 March 2020].

[2] E. Olmedo, J. Calleja, A. Benitez and M.A. Medina, "Point to point processing of digital images using parallel computing", Int. J. Comput. Sci. Issues, 9(3): 1-10, 2012.

[3] G. Anbarjafari, "Digital Image Processing". University of Tartu; 2014 [accessed 2020 February 05]. https://sisu.ut.ee/imageprocessing/documents.

[4] Gerber, Richard "Getting started with openMP". (2012, June 7) Retrieved March 7, 2020, from https://software.intel.com/en-us/articles/getting-started-with-openmp.

[5] "Guide into OpenMP: Easy multithreading programming for C++". (2016, June). Retrieved March 7, 2020, from https://bisqwit.iki.fi/story/howto/openmp/

[6] H. L., S., "2D Convolution in Image Processing - Technical Articles". Allaboutcircuits.com; 2018 [Accessed 5 Feb. 2020] https://www.allaboutcircuits.com/technical-articles/two-dimensional-convolution-in-image-processing/ .

[7] Li, X. (2019, December 4). "Scalability: strong and weak scaling". Retrieved March 7, 2020, from https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/

[8] N. Kumar, "Digital Image Processing Basics". GeeksforGeeks; [accessed 2020 February 06]. https://www.geeksforgeeks.org/digital-image-processing-basics/.

[9] OpenMP. (2018, July 1). "About Us". Retrieved March 8, 2020, from https://www.openmp.org/about/about-us/

[10] OpenMP.org, "OpenMP 5.0 API Syntax Reference Guide". (2019, May) Retrieved March 7th, 2020, from https://www.openmp.org/wp-content/uploads/OpenMPRef-5.0-111802-web.pdf

[11] P. Ganesh, "Types of Convolution Kernels : Simplified". Medium; 2019 [Accessed 5 Feb. 2020]. https://towardsdatascience.com/types-of-convolution-kernels-simplified-f040cb307c37 .

[12] "Parallel Speedup". (n.d.). Retrieved March 7, 2020, from http://selkie.macalester.edu/csinparallel/modules/IntermediateIntroduction/build/html/ParallelSpeedup/ParallelSpeedup.html

[13] R. Fisher, S. Perkins, A. Walker, and E. Wolfart, "Image Processing Learning Resources". HIPR2; 2003 [accessed 2020 February 06]. https://homepages.inf.ed.ac.uk/rbf/HIPR2/convolve.htm.

[14] Saxena, Sanjay & Sharma, Shiru & Sharma, Neeraj, "Parallel Image Processing Techniques, Benefits and Limitations"; Research Journal of Applied Sciences, Engineering and Technology. 12. 223-238. 10.19026/rjaset.12.2324. 2016.

[15] Tousi, Ashkan & Vanderbauwhede, Wim & Cockshott, Paul, "2D Image Convolution using Three Parallel Programming Models on the Xeon Phi"; 2016.

[16] tutorialsPoints, "Concept of Convolution". tutorialsPoint; 2020 [accessed 2020 February 05]. https://www.tutorialspoint.com/dip/concept_of_convolution.htm.

[17] Vandevenne, L., 2018. "Image Filtering". [online] Lodev.org. Available at: ¡https://lodev.org/cgtutor/filtering.html¿ [Accessed 9 March 2020].

[18] "What is Amdahl's Law? - Definition from Techopedia". (2018, April 13). Retrieved March 7, 2020, from https://www.techopedia.com/definition/17035/amdahls-law

[19] Yliluoma, J. (2016, June). "Guide into OpenMP: Easy multithreading programming for C". Retrieved March 8, 2020, from https://bisqwit.iki.fi/story/howto/openmp/