

Escuela Politécnica Superior

University of Alacant

Components of the Group:

Cristian Andrés Córdoba Silvestre (05988721G)

Nizar Nortes Pastor (74444345S)

Teacher: Antonio Martínez Álvarez

3/11/2023

IC PRACTICE 2

Searching Cndidates for Parallelization

# 

# ÍNDEX

1. Introduction to the problem pág. 3

2. Introduction to Magick++.………………………………………………………..………………………. pág. 4

3. Management of the program.……………………………………………..……………………………. pág. 5

1. Implementation and explanation of the code.…………………………………………..……. pág. 6
2. Tasks of execution and performance.………………………………………..………………………. pág.9 6. Bibliography.………………………………………………………..……………………………..……………. Pág.

"INTRODUCTION TO THE PROBLEM”

The aim of this task was to execute an issue demanding considerable computational power. We believed that crafting an image filter which alters the image to appear with larger pixels (akin to a drop in resolution) and shifts the hues based on specified intervals to a 16-color set would be suitable. Our initial hurdle was image reading and writing. To tackle this, we utilized a C++ toolkit called Magick++. While this toolkit provides an extensive range of image manipulation tools, we merely employed it for reading, writing, and modifying pixels for our method. For the pixelation effect, we determined the median shade of a defined segment (size can be selected via parameter) and then designated this color to all corresponding pixels, ensuring the entire segment appeared as a uniform color or a "bigger pixel". During the color computation, contingent on the resulting value, we deduced its counterpart in the 16-color set, which then became the final assigned hue. This outlines how the retroGaming.cpp procedure adjusts any image, making it resemble visuals from retro video game.

# INTRODUCTION TO MAGICK++

****

While searching for ways to work with images in C++, we came across the Magick++ library, which processes images and has a plethora of functions related to them. Magick++ is a C++ library used by several projects such as Gimp, Octave, and various GNU Linux operating system families; it's generally quite common. We found an object-oriented image processor that's very easy to manage and has a comprehensive API. The part that interests us is the Image class, one of its attributes being Pixels, which acts like a pixel map accessed by row and column. Each of these pixels contains a PixelPacket which has three numbers in percentage (from 0 to 1) representing the amount of red, green, and blue required to achieve that RGB color. The Image class allows for direct reading from a file and writing to save the image we've worked on. The scope of the library extends much further, especially in the STL Algorithms, offering a myriad of useful functions for image processing such as applying a color filter, resizing, calculating the average color of a pixel block, etc. However, these are not of interest to us for this project.

In our quest to handle images using C++, we stumbled upon the Magick++ library, a platform designed for image manipulation boasting numerous associated functionalities. Many projects, including Gimp, Octave, and various GNU Linux OS families, utilize Magick++, attesting to its widespread adoption. We discovered a user-friendly, object-centric image processing system equipped with an extensive API. Our primary focus was the Image class within the library. One key attribute of this class is Pixels, functioning as a pixel grid accessible by rows and columns. Each pixel houses a PixelPacket, comprising three percentage-based values (ranging from 0 to 1) that depict the requisite red, green, and blue components for the desired RGB shade. The Image class facilitates direct file reading and saving of the manipulated image. While the library's reach spans much broader, especially within the STL Algorithms, showcasing a vast array of image processing tools like color filter applications, resizing, and average color computation for a pixel cluster, these aspects remain outside our current project's purview

# MANAGEMENT OF THE PROGRAM

Here are the commands we used:

$ apt-cache search dev | grep magick // verifies the package's existence

$ sudo apt-get install libmagick++-dev // procures the package

$ sudo apt-get install libgraphicsmagick1-dev // fetches a vital library (dependencies)

Using the above instructions, I deployed Magick++ and discerned the command to execute the code.

$ g++ Magick++-config --cxxflags --cppflags <Program\_Name>.cpp -o <Executable\_name> Magick++-config --ldflags --libs

Diving into the Magick++ API, and gleaning knowledge from the sparse and somewhat muddled samples they offered, we crafted our debut "application" – a simple tool opening an image and resaving it with an altered title. We discerned that Magick++ possesses an "Image" class dedicated to image-related tasks. Our sole focus was the image-loading functionality this class presented, facilitating immediate image editing. Given our unambiguous objective centered on color manipulation, our endeavors were anchored around two primary classes: "Image" and "ColorRGB". The former allowed pixel interaction, while the latter facilitated color management.

# IMPLEMENTATION AND EXPLANATION OF THE CODE

**Texto

Descripción generada automáticamente**

IMPLEMENTATION AND EXPLANATION OF THE CODE

Main Function: First, the code evaluates the count of arguments, initializes "Magick++" for image operations, and captures the pixel count intended for our enlarged pixel (this input is given as an argument). Following this setup, an introductory "for" loop processes the InputFile/OutputFile combinations, ensuring arguments appear in pairs; solo pairs aren't processed. Utilizing "Image imagen(input\_file\_name)", the image is loaded, employing the Image constructor from the Magick++ toolkit. Subsequently, the image's dimensions are recorded. Next, dual nested "for" loops are employed to scan the image comprehensively, invoking 'subsector' with the image, its boundaries, and the pixel size for each enlarged pixel (henceforth termed 'tamBigPix'). Upon loop completion, the image is saved using the designated name via "imagen.write(output\_file\_name)". An encompassing try/catch structure oversees image loading; if an image isn't located, the software won't halt abruptly

Texto

Descripción generada automáticamente

Subsector Function: This segment is tasked with determining the mean color value of the larger pixels, subsequently painting these pixels uniformly while switching from a 256-color spectrum to a 16-color palette. Initially, it establishes variables for "r" (red), "g" (green), and "b" (blue) alongside an auxiliary ColorRGB variable that streamlines the average computation process. Enclosed are dual nested loops that scan the subsector, their range derived from the main function and provided as an input. The values of "r", "g", and "b" are accumulated and subsequently divided by the sector's total pixel count to deduce the average. Post acquiring the mean value, the "to16Pallete" function is invoked, with its output being stored in a ColorRGB. Following this step, we determine the exact color to employ..

Texto

Descripción generada automáticamente

To16Palette Method: - Given our inability to seamlessly transition from a 256-color spectrum to a 16-color variant, we opted for an estimation. We allocated specific intervals for each hue; thus, if the computed average lands within a certain interval, the matching color from the 16-color palette is returned. These intervals hinge on the red, green, and blue (RGB) values. Within the 16-color palette, each hue utilizes 8 bits, amounting to 24 bits altogether (in contrast to the original 8 bits used to encapsulate the entire color). Consequently, each value spans from 0 to 255, as observed

# 



Task 2.1

Given the CFGs, we can note that our issue is indeed parallelizable. This can be seen in the Main and in the function to16Palette where we make multiple choices, and these can be executed simultaneously by parallelizing the software. In these functions, several parallelizable chunks can be found in the form of concatenaded if, else-if,else

Task 2.2

**Access of Read/Write to Variables**

Most of the variables in the program are used for temporary storage or iteration:

-. **argc** and **argv** are parameters of the main function, providing the number of command-line arguments and their values.

**-. tiempoTotal\_1, tiempoTotal\_2, tiempoTotal, tiempo1, and tiempo1fin** are used for time measurement. They are read from and written to.

**-. tamBigPix** is read once from the command line arguments and is used throughout the **main** function.

**-. Image imagen** represents an image object. The program reads from and writes to it.

-. Variables **ancho** and **alto** are used to store the image's dimensions.

-. **r, g, b,** and **auxColor** in **subSector** are accumulators. They get modified frequently.

**Type of Variables**

**Large Data Matrices:** The main data matrix in the program is the Image imagen. This matrix stores pixel values of the image.

**Rarely Accessed Data:** Command-line parameters like argv[1] for the pixel size are read once and then remain constant throughout the program. Similarly, ancho and alto (width and height) are only read once per image processing.

**Automatic Variables:** There are several automatic variables like loop counters i, j, k, and accumulators r, g, b in the subSector function. These get created and destroyed frequently during the program execution.Here we offer some extra considerations:

**Improvements in the program's flow**

**Image Decomposition:**

Instead of processing the image pixel-by-pixel (or block-by-block),we could divide the image into smaller chunks, or tiles. Each tile can then be processed independently and in parallel.

For instance, if we have 4 processing units, we could divide the image into 4 roughly equal sections, and each section could be processed by a separate unit.

**Eliminate Nested Conditionals in the Color Conversion:**

The to16Palette function consists of deeply nested if-else conditions. This can be restructured using lookup tables or formulas to streamline the logic and reduce branching, making it more efficient.

A lookup table, for instance, might store precomputed values or boundaries to identify which color in the 16-color palette a given RGB value maps to.

**Pixel Summation in subSector:**

Instead of two nested loops in the subSector function where we sum the red, green, and blue values, and then another two nested loops to set the color, a good option would be to consider using parallel reduction for the summation part.

**Preallocate Memory:**

Instead of creating new ColorRGB objects in every call to to16Palette, consider preallocating a buffer of ColorRGB objects and simply updating their values. This can reduce the overhead of frequent memory allocations.

**Offload Computation:**

If available, consider offloading computation to a GPU, especially if working with large images. GPUs are well-suited for the kind of parallel processing required for image processing.

**Avoid Global Synchronization Points:**

The code measures the time taken for each image processing. While this is great for profiling, in a parallel environment, global synchronization points (like waiting for all threads to finish) can be a bottleneck. Ensure that synchronization is kept to a minimum.

**Problems with the cache**

The cache can be a concern in two primary ways: temporal locality (re-accessing a memory location within a short period) and spatial locality (accessing memory locations that are close to one another within a short period). In a program, tight loops that process arrays can often benefit from or be hindered by the cache depending on how they're structured:

**Pixel Processing in subSector Function**

The image pixels are processed in a nested loop, where inner and outer loops iterate through the y and x coordinates, respectively. This could cause cache misses, especially if the width (ancho) of the image is large. The reason being that the data might be row-major (contiguous in memory by rows) and by iterating over the y coordinate in the inner loop, we jump to different rows, potentially causing cache misses.

To improve cache locality, the loops can be swapped, so the inner loop processes the x coordinate.

**Access Pattern in to16Palette Function**

The function uses a series of nested if-else conditions to determine the color palette. While this isn't directly related to cache behavior, the branching can cause pipeline stalls, which can affect performance. Branch prediction can help, but if the input values for r, g, and b are unpredictable, it may lead to mispredictions.

**Temporal Locality in Main Loop**

In the main loop where images are processed, the entire image is loaded, processed, and then written out before the next image is processed. This means the program doesn't take advantage of temporal locality for cases where multiple images are processed. But this might not be a big concern unless the processing of each image is quite fast compared to the I/O times.

Task 2.3

To adjust the problem's workload, there are several approaches. We might replace the image, affecting the program, with a bigger or smaller one, increase or decrease the pixelation factor taken by argument, or handle several images at once by making minor code changes.

Task 2.4

**Optimization Parameters -Ox**

GCC's standard optimization setting is -O0. As per the GCC handbook: "Without any optimization preference, the compiler focuses on minimizing compilation overhead and ensuring that debugging delivers consistent outcomes." So, the code remains unoptimized.

The elementary optimization level is -O1, where the compiler seeks to generate efficient, concise code in a short compilation duration. It's rudimentary, but it accomplishes its task.

-O2 follows this, serving as the advised optimization tier unless there are specific system requirements. -O2 introduces a few more options than what -O1 offers. Here, the compiler strives to boost code efficiency without sacrificing size or significantly extending the compilation period.

In contrast, -O3 represents the apex of optimization. It initiates certain optimizations that demand more compilation time and memory. Yet, using -O3 doesn't assure enhanced speed; in instances, it might decelerate a system because of extensive binaries and significant memory consumption.

**The –march Setting**

The -march setting dictates the kind of code suitable for a particular processor layout. CPUs vary in attributes, endorsing distinct command sets and code execution methodologies.

When uncertain about the CPU kind or the right configurations, one can employ the -march=native. In doing so, GCC tries to recognize the processor and automatically selects fitting options.

**The -Ofast Setting**

The -Ofast setting encompasses all the -O3 enhancements and some that might not fit all compliant applications.

**The -floop-parallelize-all Setting**

The -floop-parallelize-all configuration attempts to run every loop in parallel, which can be assessed to ensure no loop-dependent interactions, without verifying its efficiency.