Concurrent and High Integrity Software

Digital Image Processing and Analysis in Multicore Systems

Professors: Jorge Coelho and Luís Nogueira

Francisca Barros - 1210099

Porto, 09th of June of 2022

Table of Contents

Table of Figures

1. Abstract

The following report portrays the final project carried out for the course of Concurrent and High Integrity Software (SOCOF, standing for *Software Concorrente e Fiável*) of the Master's in Informatics Engineering, Computer Systems specification.

The main goal of the project was to explore digital image processing techniques, using sequential, multithreaded and thread-pool-based approaches, by implementing two different filters required for this assignment.

The present report covers the full development of this project, describing the steps carried out in order to meet the objectives, and presents an analysis on the results obtained using the three approaches implemented.

2. Digital Image Processing – Overview

Digital Image Processing is the alteration of digital pictures using algorithms (and a computer system). Depending on the method used, it may be feasible to improve the image by sharpening and restoring it.

Two filters that allow for the processing of digital photos were built for the development of this project, one to emphasize forest fires and another to clean an image by eliminating objects and people from it. It is feasible to improve the efficiency of such algorithms by taking advantage of threads and so building multithreaded implementations.

The sub-sections that follow provide a theoretical explanation on how the filters implemented work.

2.1. Highlight Forest Fires

With this filter, the final goal is to transform satellite images with fires, where the fire pixels are replaced by red pixels, and the remaining pixels are replaced with their grayscale equivalent.

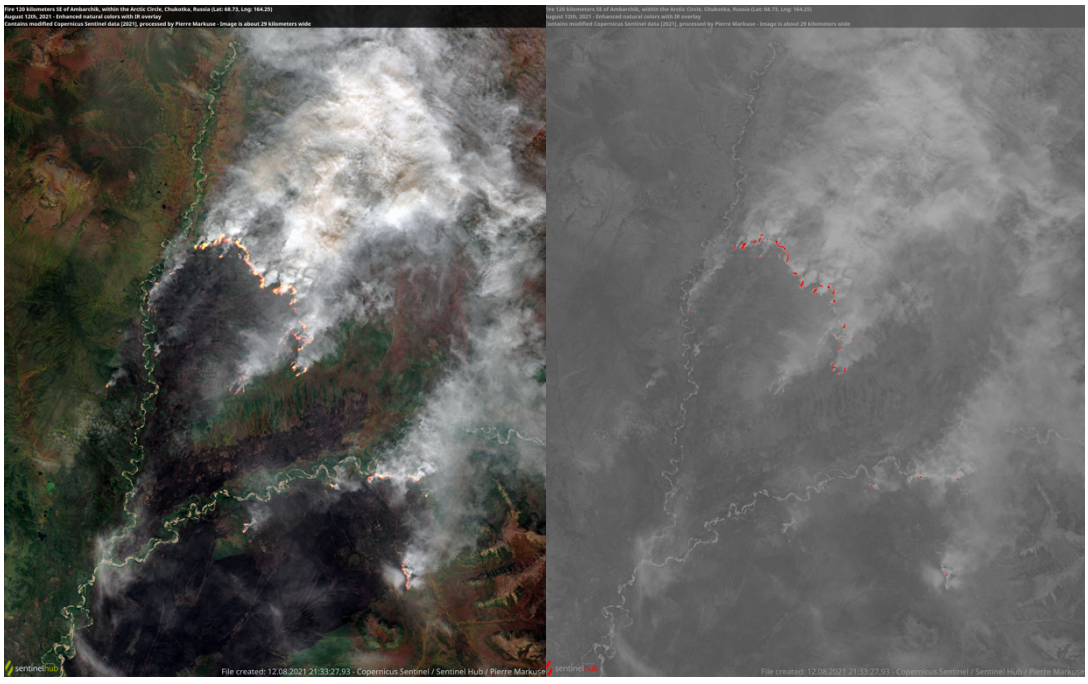

Figure 1 - Original image and transformed image with highlighted red pixels, side by side

In order to achieve this, the following key concepts and assumptions must be considered:

- *Assumption 1*: An arbitrary pixel (x, y), has its red, green and blue values defined by red_{val}, $green_{val}$ and $blue_{val}$ respectively.
    - As such, if this pixel is not red, its greyscale equivalent is related to the average of these three values.

$$grey\ pixel = average\ = \frac{red_{val} + green_{val} + blue_{val}}{3}$$

- *Assumption 2*: A threshold value th, and the reference values for green and blue pixels, ref_{green} and ref_{blue} , are to be used in the identification of a red pixel.
    - A pixel is classified as red if its value is superior to the average of its RGB values multiplied by the threshold value, and its green and blue pixels are below the reference values.

$$red\ pixel > \ average_{pixel} * th$$

$$red\ pixel_{green\ value} < ref_{green}\ \&\&\ red\ pixel_{blue\ value} < ref_{blue}$$

Following the guidelines on the assignment, the reference values for the green and the blue pixel were defined as 100 and 200, correspondingly.

2.2. Clean Image

The goal of this second filter is to create a cleaner version of an arbitrary number of images, by removing objects and people from them.

Figure 2 - Original image and cleaned image, side by side

To compute the value of each pixel of the output cleaner image, the following steps should be followed:

- Choose and indicate the path of an arbitrary number of images;
- Calculate the average for all the correspondent pixels in source images;
- Calculate the distance of the RGB values of each pixel in source images to the average;
- Choose the closest one, and apply to the output image;

3. The Algorithms

The sections that follow provide a brief overview on the implementation of the algorithms, more specifically on the decisions made while implementing their multi-threaded and thread-pool based implementations.

3.1. Highlight Forest Fires

This filter's implementation was previously included in the base source code. In order to accommodate all three implementations, a switch case with the execution option requested (that is, the approach to execute) was created in the primary handler method, *HighLightFireFilter*, in the *Filters.java* file.

3.1.1. Sequential Implementation

The given source code was used for the sequential implementation of this filter, and was included in the switch-case corresponding to the sequential technique.

```
// Sequential Approach
case "sequential":
    for (int i = 0; i < tmp.length; i++) {
        for (int j = 0; j < tmp[i].length; j++) {

            // fetches values of each pixel
            Color pixel = tmp[i][j];
            int r = pixel.getRed();
            int g = pixel.getGreen();
            int b = pixel.getBlue();

            // takes average of color values
            int avg = (r + g + b) / 3;

            if(r > avg*threshold && g < 100 && b < 200)
                // outputs red pixel
                tmp[i][j] = new Color(r: 255, g: 0, b: 0);
            else
                // outputs grey pixel
                tmp[i][j] = new Color(avg, avg, avg);
        }
    }
    break;
```

Figure 3 - Sequential implementation of the Highlight Forest Fires filter

In summary, this solution will: traverse the image's pixels vertically and horizontally; get the corresponding pixel in position (x, y) and compute its average; and determine if it is a red pixel or its greyscale equivalent.

The performance test using the sample file *russia1.jpg* revealed that this approach took roughly *11226ms* to complete. Considering the large size of the image (6.3MB, with dimensions of 4048x5028), it was anticipated that this strategy would be taking the longest in comparison to the approaches provided in the following sections.

3.1.2. Multi-Threaded Implementation

In order to make the computing of such results more effective and efficient, one could exploit parallelism by using threads.

To achieve a multi-threaded implementation, and decreasing the execution speed of the code developed, a new class that extends from *Runnable*, *FilterThread*, was created. Its constructor has, as arguments, the total number of threads (*total_threads*), the ID of the current thread (*thread_id*), the image Color matrix (*image*), and the threshold value (*threshold*).

```
private class FilterThread implements Runnable {

    private Color[][] image;
    private int total_threads;
    private int thread_id;
    private float threshold;

    public FilterThread(int total_threads, int thread_id, Color[][] image, float threshold) {
        this.total_threads = total_threads;
        this.thread_id = thread_id;
        this.image = image;
        this.threshold = threshold;
    }
```

Figure 4 - Constructor of the FilterThread class

Since the image in analysis is a shared resource for all working threads, the ***synchronized*** keyword, which consumes the image as an argument, was utilized. As a result, the system understands that it must first secure a lock on this object before proceeding.

Another thing to consider is how threads will interact. The matrix of the picture being processed will be divided into sections, and each thread will only operate on the portion allocated to it. To

accomplish this, the following assertion, in the for loop, was used to determine the height to be considered for the section.

```
for (int i = 0; i < width; i++) {
    for (int j = (height / total_threads) * thread_id;
            j < (thread_id < (total_threads-1) ? (height / total_threads * (thread_id + 1)) : height);
            j++)
    {
```

Figure 5 - Assertion on the section's height

In order to determine how many threads should be used, a performance test with an inclusive range between 2 and 20 was conducted.

```
Insert the name of the file path you would like to use:
../HighlightFires/russia1.jpg
Insert the red value threshold:
1.35
------------ Highlight Fire Filter ------------
-- Sequential:  11226 (ms) --------------------
-- MultiThreaded (2):  5313 (ms) --------------
-- MultiThreaded (3):  5174 (ms) --------------
-- MultiThreaded (4):  5035 (ms) --------------
-- MultiThreaded (5):  5549 (ms) --------------
-- MultiThreaded (6):  5610 (ms) --------------
-- MultiThreaded (7):  5835 (ms) --------------
-- MultiThreaded (8):  5662 (ms) --------------
-- MultiThreaded (9):  5678 (ms) --------------
-- MultiThreaded (10):  5938 (ms) -------------
-- MultiThreaded (11):  5885 (ms) -------------
-- MultiThreaded (12):  5717 (ms) -------------
-- MultiThreaded (13):  6088 (ms) -------------
-- MultiThreaded (14):  5730 (ms) -------------
-- MultiThreaded (15):  6174 (ms) -------------
-- MultiThreaded (16):  6150 (ms) -------------
-- MultiThreaded (17):  6414 (ms) -------------
-- MultiThreaded (18):  5983 (ms) -------------
-- MultiThreaded (19):  6166 (ms) -------------
-- MultiThreaded (20):  6076 (ms) -------------
```

Figure 6 - Multi-threaded performance assessment of Highlight Forest Fires filter

The data shown above indicate that, even with only two threads, the time elapsed is significantly less than the sequential technique. In further detail, four was the ideal number of threads, since its execution time was the shortest of all, 5035ms. Passing this "happy medium," the execution time appears to climb and plateau when 10 threads are used — that is, having ten threads functioning, or twenty, has no significant gain in performance.

3.1.3. Thread-Pool Based Implementation

The only key difference between this implementation and the previous one is the use of a Thread-Pool with fixed size. A fixed-size thread pool creates threads as tasks are submitted, up to the maximum pool size.

```
// ThreadPool approach
case "threadpool":
    ExecutorService executor = Executors.newFixedThreadPool(number_of_threads);
    for (int i = 0; i < number_of_threads; i++) {
        executor.execute(new FilterThread(number_of_threads, i, tmp, threshold));
    }
    executor.shutdown();
    while(!executor.isTerminated()) {}
```

Figure 7 - ThreadPool Based Implementation

The same performance assessment as in the previous sub-section was conducted, with the same inclusive range between 2 and 20, to determine how big the thread-pool should be.

```
------------ Highlight Fire Filter ------------
-- Thread-Pool (2):  11769 (ms) ---------------
-- Thread-Pool (3):  7195 (ms) ----------------
-- Thread-Pool (4):  5461 (ms) ----------------
-- Thread-Pool (5):  5361 (ms) ----------------
-- Thread-Pool (6):  5348 (ms) ----------------
-- Thread-Pool (7):  5617 (ms) ----------------
-- Thread-Pool (8):  5912 (ms) ----------------
-- Thread-Pool (9):  5795 (ms) ----------------
-- Thread-Pool (10):  6075 (ms) ---------------
-- Thread-Pool (11):  6025 (ms) ---------------
-- Thread-Pool (12):  5794 (ms) ---------------
-- Thread-Pool (13):  5813 (ms) ---------------
-- Thread-Pool (14):  6124 (ms) ---------------
-- Thread-Pool (15):  6025 (ms) ---------------
-- Thread-Pool (16):  6155 (ms) ---------------
-- Thread-Pool (17):  5852 (ms) ---------------
-- Thread-Pool (18):  7188 (ms) ---------------
-- Thread-Pool (19):  6214 (ms) ---------------
-- Thread-Pool (20):  6502 (ms) ---------------
```

Figure 8 - Thread-Pool performance assessment of Highlight Forest Fires filter

According on the statistics above, the best thread-pool size should be between four and six. Because the performance evaluation was run numerous times, the results would have modest fluctuations, with the shortest execution time occurring when the thread-pool had four, five, or six threads. As determining how big the thread-pool should be is not a precise science, I chose five as it is the midway number and the second smallest figure in the test above.

3.2. Clean Image

Unlike the previous filter, this one was not provided in the original source code and had to be constructed independently by the students. *ComplexFilters.java* was developed as a new file to accommodate the three different implementations necessary.

The constructor of the ComplexFilters class receives an ArrayList will all the filenames to be considered. Then, it loads them into a color matrix and adds them to an ArrayList of all Color arrays.

```
// Constructor with filename for source image
ComplexFilters(ArrayList<String> filenames) {
    this.images = new ArrayList<>();
    this.file_names = filenames;
    // Load each image into the array list
    for (String file_name: this.file_names) {
        Color[][] loaded_image = Utils.loadImage(file_name);
        this.images.add(loaded_image);
    }
}
```

Figure 9 - Constructor of the ComplexFilters class

The principal handler method, *CleanImageFilterImproved*, will handle all approaches with a switch case, much like the previous filter.

3.2.1. Sequential Implementation

The sequential implementation, which is included in the switch-case corresponding to this approach, will select the first picture loaded and generate a temporary duplicate of it. Then it will traverse its pixels vertically and horizontally, obtaining the equivalent pixel in position (x, y) for all pictures evaluated, calculating the average value, and selecting the closest one for the final image.

```
// Sequential Approach
case "sequential":
    // Go through the color-array horizontally and vertically (get the pixel in (x,y))
    for (int i = 0; i < tmp.length; i++) {
        for (int j = 0; j < tmp[i].length; j++) {

            ArrayList<Color> pixels = new ArrayList<>();

            // Fetch values of pixel (x,y) in each image and append to array
            for (int a = 0; a < this.images.size(); a++) {
                Color pixel = this.images.get(a)[i][j];
                pixels.add(pixel);
            }

            // Calculate the average pixel of the array
            Color average_pixel = calculateAverage(pixels);

            // The chosen one
            Color chosen_one = calculateDistanceAndGetResult(average_pixel, pixels);

            tmp[i][j] = chosen_one;
        }
    }
```

Figure 10 - Sequential implementation of the Clean Image filter

Two auxiliar methods were created, in order to aid the computation of the average pixel and the closest one – *calculateAverage* and *calculateDistanceAndGetResult*, respectively.

```
public Color calculateAverage(ArrayList<Color> pixels) {

    int sum_red = 0;
    int sum_green = 0;
    int sum_blue = 0;

    for (Color pixel: pixels) {
        sum_red += pixel.getRed();
        sum_green += pixel.getGreen();
        sum_blue += pixel.getBlue();
    }

    return new Color(Math.round(sum_red/pixels.size()),
                     Math.round(sum_green/pixels.size()),
                     Math.round(sum_blue/pixels.size()));
}
```

Figure 11 - Auxiliar method to calculate the average pixel

Regarding performance, this sequential approach clocked in at 5417ms for it to finish. Since three images are being considered (*clean1.jpg*, *clean2.jpg* and *clean3.jpg*), this filter turned out to be more efficient than the previous one, considering their sequential implementations.

3.2.2. Multi-Threaded Implementation

As the ultimate goal of the assignment is to explore multi-threaded implementations, and its potential increases in performance, a multi-threaded approach was also developed for this filter.

Similar to the previous filter a new class that extends from Runnable was also created, *ComplexFilterThread*. This time, its constructor receives the total number of threads (*total_threads*), the ID of the current thread (*thread_id*), the image Color matrix of the output image (*image*), and the ArrayList of images to be considered (*images*).

```
public ComplexFilterThread(int total_threads, int thread_id, Color[][] image, ArrayList<Color[][]> images) {
    this.total_threads = total_threads;
    this.thread_id = thread_id;
    this.image = image;
    this.images = images;
}
```

Figure 12 - Constructor of the ComplexFilterThread class

Seeing as the output picture is a shared resource between threads, the ***synchronized*** keyword was also used on it. The technique selected to specify how the threads will function is the same as in the previous filter – that is, the matrix of the processed output picture will be separated into parts, and each sector will be assigned to a single thread.

In this strategy, determining how many threads to employ was done in a way similar to the previous filters.

```
Insert the name of the file paths you would like to use
 (1 per line)
 (END to stop):
../CleanImage/clean1.jpg
../CleanImage/clean2.jpg
../CleanImage/clean3.jpg
END
------------ Clean Image Filter ------------
-- Sequential:  5417 (ms) ------------------
-- MultiThreaded (2):  2759 (ms) --------------
-- MultiThreaded (3):  2800 (ms) --------------
-- MultiThreaded (4):  2681 (ms) --------------
-- MultiThreaded (5):  2683 (ms) --------------
-- MultiThreaded (6):  2667 (ms) --------------
-- MultiThreaded (7):  2741 (ms) --------------
-- MultiThreaded (8):  2755 (ms) --------------
-- MultiThreaded (9):  2846 (ms) --------------
-- MultiThreaded (10):  2772 (ms) --------------
-- MultiThreaded (11):  2928 (ms) --------------
-- MultiThreaded (12):  2813 (ms) --------------
-- MultiThreaded (13):  2867 (ms) --------------
-- MultiThreaded (14):  2912 (ms) --------------
-- MultiThreaded (15):  2893 (ms) --------------
-- MultiThreaded (16):  3224 (ms) --------------
-- MultiThreaded (17):  2928 (ms) --------------
-- MultiThreaded (18):  2987 (ms) --------------
-- MultiThreaded (19):  3015 (ms) --------------
-- MultiThreaded (20):  2963 (ms) --------------
```

Figure 13 - Multi-threaded performance assessment of Clean Image Filter

As expected, with only two threads the performance increased, and the time elapsed decreased. However, by using six threads, it seems that the execution time is the lowest of all, at 2667ms. The execution time also seems to hit a plateau at around 10 threads.

3.2.3. Thread-Pool Based Implementation

This implementation also employed a Thread-Pool with a fixed size, like the previous filter, and its performance was evaluated in the same way.

```
// ThreadPools approach
case "threadpool":
    ExecutorService executor = Executors.newFixedThreadPool(number_of_threads);
    for (int i = 0; i < number_of_threads; i++) {
        executor.execute(new ComplexFilterThread(number_of_threads, i, tmp, this.images));
    }
    executor.shutdown();
    while(!executor.isTerminated()) {}
```

Figure 14 - Thread-Pool based implementation of the Clean Image filter

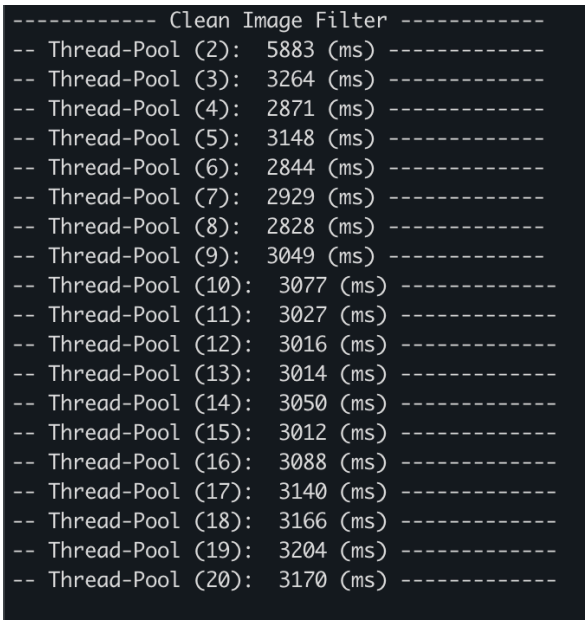

Figure 15 - Thread-Pool performance assessment of Clean Image filter

Using thread-pools of size four, six and eight seems to give the best results, regarding performance, as the image above with the test results illustrates. Since using a thread-pool of size eight produced the lowest execution time – that is, the best performance, at 2828ms – the size of the pool was defined at eight.

4. Conclusions

Based on the foregoing explanations, it is possible to conclude that the objectives set by this assignment were fully met - both filters were developed using all three approaches required, and performance difficulties were handled as best as possible.

By completing this project and report, I was able to expand my understanding of parallel programming, using multi-threaded and thread-pool-based methodologies, as well as explore how to optimize the final performance gain by evaluating the results obtained.

References

Materials provided by the professors, in the Moodle page of the SOCOF course

https://www.javatpoint.com/digital-image-processing-tutorial