Parallelization of an image viewer and RGB transformation algorithm

Cerullo Salvatore
salvatore.cerullo@stud.unifi.it

Abstract

All operating systems provide image viewers that allow, in addition to viewing digital images, also to make some basic transformations on the images. These changes mainly relate to sharpness, blinding, brightness and contrast. In this article we want to present an algorithm able to read images inside a directory, modify an RGB channel and write the modified image in a new directory. Two approaches will be presented, one entirely sequential and one parallel exploiting the Java programming language and we will evaluate the difference between these two approach through a lot of set of images.

1. Introduction

An image viewer is software that allows viewing, editing and organizing images. The modifications to the images allowed by an image viewer are very different from the modifications allowed by the professional retouching software (eg Adobe Photoshop). In fact, the changes allowed are some basic changes such as changing the brightness and the contrast. In this document we want to present two different approaches of an image viewer capable of modifying a single RGB channel (example in Fig 1). The two algorithms we will present are entirely written in Java. The first will be sequential and will read, edit and write one image at a time. The second algorithm, on the other hand, exploits the competitive java classes to implement a non-blocking parallel algorithm.

Figure 1. RGB modification of the Blue channel

1.1. Image Rappresentation

A digital image is the numerical representation of a two-dimensional image. There are two types of representation: bitmap or vector. We will take care of the bitmap representation.
The bitmap representation consists in representing an image as an array of points (pixels). Each point of the matrix (pixel) assumes a certain value which, for example in color images, represents the level of intensity of the fundamental colors. This value depends on the type of color coding used (eg RGB or CMYK). We will use the RGB color model in which the colors are defined as the sum of the three colors Red, Green and Blue, as we can see in the figure 2.

2. Presentation of the Algorithm

The implemented algorithm is an image viewer. This will read the files in a directory and identify only the .jpg type image files. Once identified the

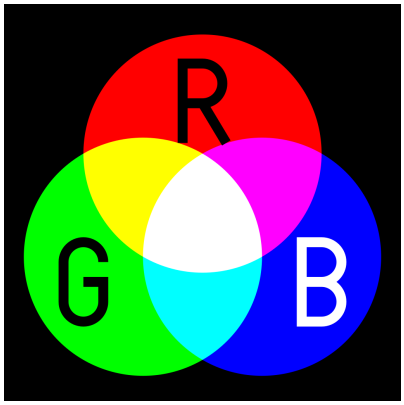

Figure 2. RGB sum

files will have to change the value of each pixel and eventually save the data in a new directory.
The modification that will be made to the image is a change concerning the value of the RGB channels, and in particular the Blue. To do this, before executing any operation on the pixels, a random number will be identified in the range [0.255] and for each pixel will modify the Blue channel by entering this value. The code of the image modification function is shown in Figure 3.
We can then identify the following steps within the algorithm:

- Reading all the jpg images in a given directory;
- Identification of a random value between 0 and 255;
- For each pixel of the image we are going to modify the Blue channel with the value calculated in the previous point;
- Writing modified images into a new directory.

Two implementations of the algorithm will be proposed: a sequential one, and a parallel one.

2.1. Sequential Algorithm

The sequential algorithm was implemented with the Java programming language. The implementation follows the steps described above.
A particular problem that has been identified during the realization of the algorithm is the processing in memory of the images. In fact, for folders containing large amounts of data (about 100 MB), the memory keeps running out. To eliminate this problem, the images are not loaded all in memory, but only the image you want to process will be loaded.
Figure 3 provides the processing code of a single image. It is easy to understand that the output of our system will be given by Loaded $->$ Modified $->$ Written ; Loaded $->$ Modified $->$ Written ; and so on.

```
//Image Loading
Immagine imm= new Immagine(pathDir, files[i]);
System.out.print("Loaded -> ");

//Image Modification
imm.modificaImmagine();
System.out.print("Modified -> ");

//Image writing
ImageIO.write(imm.getBufferedImage(), "jpg",
    new File(pathDirOut.concat(files[i])));
System.out.println("Written ;");
```

Figure 3. Image processing Code

2.1.1 RGB channel editing function

In this section we analyze how the image editing operation is performed. Figure 4 shows the modification code of an RGB channel. We note that the first operation performed is to acquire the new value of the Blue channel. After that each pixel is identified through a double cycle that runs through the rows and columns of the original image. Within the cycle we have the operation of setting the new value in the Blue channel.

```
public void modificaImmagine() throws IOException {
    int param = new Random().nextInt(255);
    BufferedImage modImage = this.getBufferedImage();
    for(int i = 0 ; i < modImage.getWidth() ; i++){
        for(int j = 0; j < modImage.getHeight() ; j++){
            Color c = new Color(modImage.getRGB(i,j));
            modImage.setRGB(i , j , new Color(c.getRed(),
                                c.getGreen(), param).getRGB());
        }
    }
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    ImageIO.write(modImage, "jpg", baos);
    this.byteStream = baos.toByteArray();
}
```

Figure 4. Code of modificaImage() function

2.2. Parallel Algorithm

The sequential algorithm just presented, as it is easy to guess, will have very poor performance for directories containing a large amount of images. To improve performance, a version of the algorithm that exploits java multithreading, and in particular the high-level Frameworks, has been realized.

We therefore want to present a parallel non-blocking algorithm, realized thanks to the ConcurrentLinkedDeque of Java. In fact, these objects are not blockers and it is therefore possible to read and insert elements inside it without blocking the other threads that access it.

The algorithm will then use this object to store the names of the .jpg files inside the folder. At this point each thread will read and delete an element from the list.

The threads are created in the main by the RemoveTask object (Figure 5) that implements Runnable and which contains a run () method. Within the run () method are inserted the read, modify and write operations, as we seen in Figure 6. It is thanks to the RemoveTask and the concurrentLinkedDeque that non-blocking parallelism can be achieved.

```
for(int i = 0 ; i < numT ; i++){
    threads[i] = new Thread(
            new RemoveTask(immagini,
                pathDir, pathDirOut));
    threads[i].start();
}
try {
    for(int i = 0 ; i < numT; i++){
        threads[i].join();
    }
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

Figure 5. Creation of Threads

3. Evaluations

In this section we want to report the results of the tests and the assessments that can be made by looking at the results. Before introducing the results, we want to present the data on which the two algorithms have been tested. We considered

```
public void run() {
    while(!immagini.isEmpty()){
        String name = immagini.pollFirst();
        try {
            //Image Loading
            Immagine imm= new Immagine(path, name);
            System.out.println(Thread.currentThread().getName() +
                    " - "  + "\tLoaded \t- " + "Image: " + name);

            //Image transformation
            imm.modificaImmagine();
            System.out.println(Thread.currentThread().getName() +
                     " - " + "\tModified - " + "Image: " + name);

            //Image writing
            ImageIO.write(imm.getBufferedImage(), "jpg",
                        new File(this.pathDirOut.concat(name)));

            System.out.println(Thread.currentThread().getName() +
                    " - " + "\tWritten \t- " + "Image: " + name);
        }
    }
```

Figure 6. Portion of the run method of RemoveTask

7 directories containing 1, 2, 4, 8, 16, 32, 64 images.

Table 1 shows the execution times of both the algorithms applied on the 7 directories. As we can see from the table and the graph shown in figure 7, the sequential algorithm's performance is good for directories containing only one image, even better than the parallel algorithm. However, by increasing the number of images in the directory, the performance worsens and, as we can see, the parallel algorithm optimizes processing times by about half of the sequential.

Directory		Algorithms	
N. Images	Size [MB]	Sequential [sec]	Parallel [sec]
1	2,4	4,93	5,63
2	4,7	7,48	5,75
4	10,1	12,21	7,55
8	23,1	26,72	16,05
16	55,9	56,93	32,94
32	179,6	138,26	71,72
64	219,6	190,18	93,39

Table 1. Algorithms elaboration times

All tests for the parallel algorithm also take into account the realization times of the various threads. In addition, the system on which the tests were carried out includes an Intel Core i5-5200U, 8 GB RAM running Ubuntu 18.10.

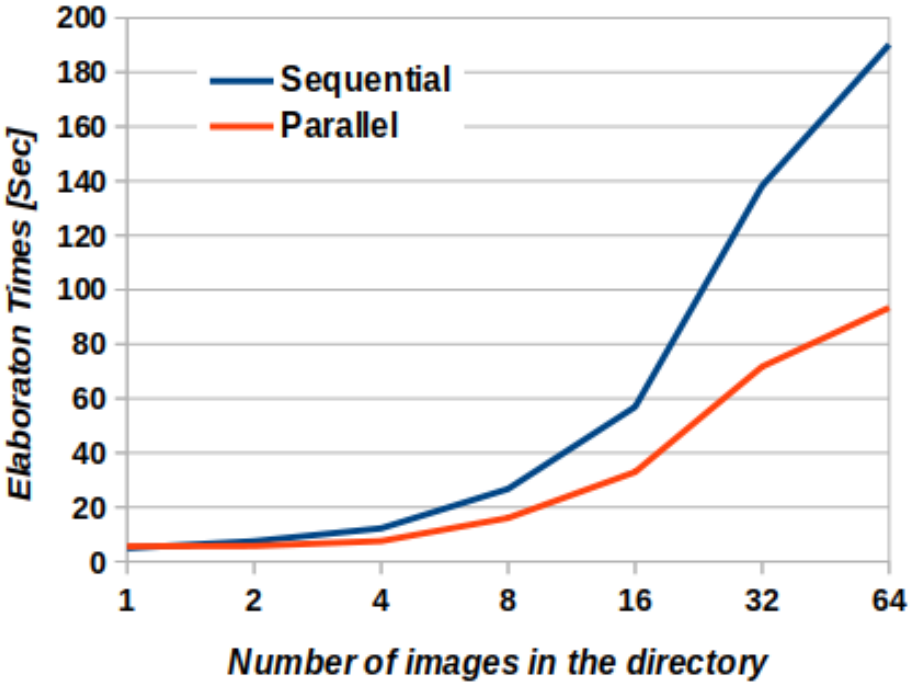

Figure 7. Test results in graphical form.