Multithread Image Reader with Java Threads

Edoardo Cagnes
edoardo.cagnes@stud.unifi.it

Andrea Giorgi
andrea.giorgi2@stud.unifi.it

Abstract

In this paper we present an Image Reader written in Java with both sequential and parallel implementation. To achieve this, we have used Java Threads and elements from the Concurrent Framework. We've also provided a benchmarking feature to measure performance between the two variants.

1. Introduction

The main purpose of this project is to analyze the difference of performances between sequential and parallel implementations of computationally expensive operations. Load an image from disc to memory is one of those and can be quite time consuming when used repeatedly on multiple images.

1.1. Tools used

The software we present is entirely made in Java with the IDE IntelliJ Idea by JetBrains [1]. We have built a GUI with the Swing Framework that allows the user to load and view images from a selected directory.

1.2. Java Threads and Concurrent Framework

Java provides built-in control over flow of executions with low level primitives and with the Concurrent Framework. In this project we've used multiple instances of the class Thread, in combination with some concurrent elements to achieve parallel implementation. More details about implementation choices will be described lately in this paper.

2. Implementation

Both the sequentially and the parallel version of this software are built under the Model View Controller Paradigm:

- All classes containing and operating on data act as Model;
- The elements of the Swing GUI are the Views;
- Controllers are all the classes that intercept the events fired by the interaction of the users on the Views.

Whenever the Model changes its status, it is notified to the views that update themselves. This is achieved using the Design Pattern Observer in pull configuration with aspect implementation as explained in [2].

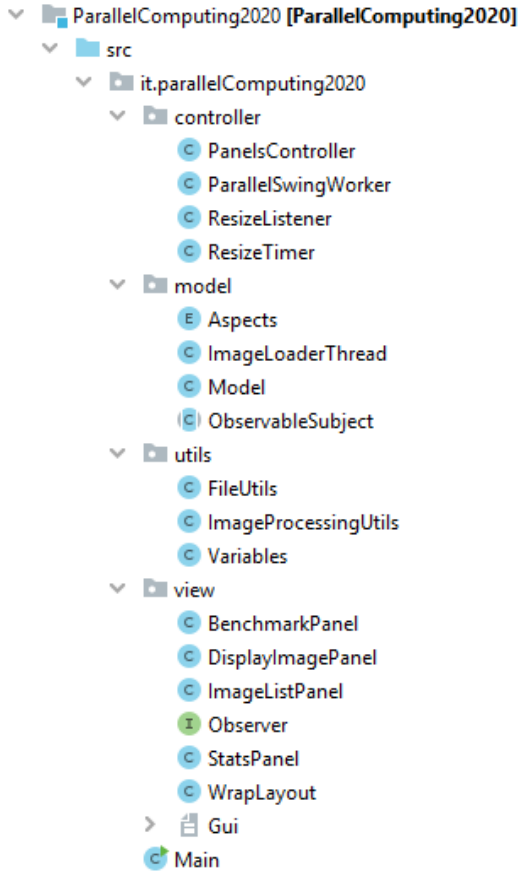

Fig. 1 - Project structure

The instruction we used to load images from the disk is `ImageIo.read(File file)`, which returns an instance of a `BufferedImage` object. This is then wrapped inside a

swing `JLabel` and displayed inside a `JPanel`. We will now describe the implementation process starting from a focus on the BufferedImage class; we will then describe the sequential version, and, eventually, explain how we added the parallel variant.

2.1. BufferedImage Class

An Image stored on disk is encoded in a sequence of bits. To display it, it must be first loaded on the RAM and then decoded into a matrix of pixels. As anticipated, the images in our project are loaded from the disk to a bunch of `BufferedImage` objects. This class was designed to load and decode almost every image format, and it's mainly formed by a `Raster` and a `ColorModel`. The first one is composed of a `DataBuffer` (the image's raw data) and a `SampleModel` that describes how the pixel of the Raster are stored inside the `DataBuffer`: it's the basic information about the rectangular array of pixels. The second one contains the methods to translate a pixel value to color components (for example, red, green and blue).

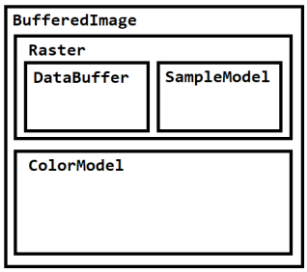

Fig. 2 - `BufferedImage` structure

Once an image is decoded in a `BufferedImage` instance it can be wrapped inside a Swing's component and displayed on screen.

2.2. Sequential Image Reader

In the first version of this software all the images were loaded sequentially. When the software starts, a `JFileChooser` is displayed and the user must select the desired folder. The user then presses on a button on the view, and the controller starts the image loading inside the model. Below we will now describe the most important methods and classes:

- **`ObservableSubject`** is an abstract class that defines the main methods and behaviors of the Subject in the design pattern Observer. It's extended by the model in order to send notifications to the views, as described in the MVC paradigm;
- **`Observer`** provides a notification interface for every object that needs to update itself whenever the `ObservableSubject` changes its status;
- **`Aspects`** is an enumeration containing the possible status changes that can involve the model. In this variant, the `ObservableSubject` has an `HashMap` containing for every enumeration of `Aspects`, a list of `Observer` objects registered to receive its notifications;

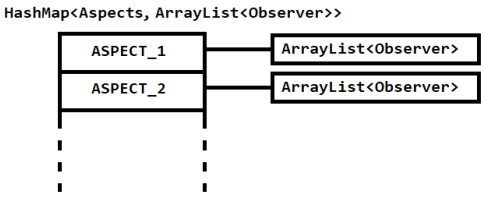

Fig. 3 - The `HashMap` used for notification

- **`Model`** is the class containing data where the images are loaded. The method that deals with this is `loadImages()`, whose purpose is to load every image contained in the user specified directory into the model. For storing data, it uses an `HashMap` where the image name is used as key. This `HashMap` is instantiated every time, to allow the garbage collector to dispose the previously loaded images. Whenever the images are loaded, the model notifies that to the views using the inherited method `notify(Aspects aspect);`
- **`ImageListPanel`** implements the `Observer` interface to be notified when the model has finished the loading operations. It's the view showing the list of loaded images that are loaded inside a `JTable`. It also contains two `JButton` to change the selected directory and to start the image loading;
- **`DisplayImagePanel`** shows to the user the image selected from the `JTable` inside `ImageListPanel`;
- **`PanelsController`** intercepts the user input on the views and converts it in operations on the model. To achieve this, it installs listeners to detect when the user presses on each button, and also provides methods to adapt the image displayed in `DisplayImagePanel` to the size of the Frame;
- **`Gui`** is the class generated by the IntelliJ GUI Designer and acts as a wrapper of the other views;
- **`WrapLayout`** is a class that helps the visualization of the buttons inside `ImageListView` [3];
- **`Main`** contains the entry point method of this software.

During the operations performed by `model.loadImages()` every Swing component freezes itself waiting for this method to respond. This issue will be fixed with the parallel implementation.

2.3. Multithread Image Reader

Several modifications have been made to achieve a parallel implementation. Inside the class `Model` a dedicated method, `loadParallel()`, was added. This method uses the class `ImageLoaderThread` to divide the

work between several parallel threads. The number of threads that will be created is the value assigned to the field `numThread` of the class `Variables`. What basically happens inside `loadParallel()` is:

- With the methods of Java class `File`, it creates an array containing references to every image contained inside the folder selected by the user;
- If the number of the images inside the folder is lower than the number of threads to be created, then as many threads as the images to be loaded will be instantiated;
- The number of images to be loaded by each thread is calculated in term of index and offset[1];
- Threads are created and started;
- `Model` wait for every thread to terminate its execution to measure performances.

Below we leave the code of the method run:

Algorithm 1 Method `run()` of `ImageLoaderThread`

```
1: public void run(){
2:   try {
3:     for (int i = start; i < start+offset;
       i++) {
4:       try{
5:         BufferedImage imageRaw =
           ImageIO.read(files[i]);
6:         model.getImages().put(files[i].getNam
           e(), imageRaw);
7:         model.notify(files[i].getName());
8:       }catch(IOException exception) {
9:         System.out.println("File " + files[i]
           + "not accessible");
10:      }
11:    }
12:  }catch(OutOfMemoryError exception){
13:    System.out.println("Memory limit
       reached");
14:  }
15:}
```

As seen from above, in this case the notification method takes a String as parameter rather than an enumeration of `Aspects` as explained in 2.1. This is one of the main modifications that has involved the parallel implementation that we will now explain:

- Now in `ObservableSubject`, an instance of `Observer` can register itself not only for notifications regarding an enumeration of `Aspects`, but also for those regarding when a single image is loaded. This is achieved by sending notifications containing the name of the loaded image (this is passed as a String instead of using an `Aspects` enumeration). Instances of `Observer` registered for single image notifications are kept in a separate Hash Map that has the filename as key and a `ConcurrentLinkedQueue` of Observers as value. Later we'll explain why in this case, instead of an `ArrayList`, we had to use a `ConcurredLinkedQueue`;
- `Model` now loads the images inside a `ConcurrentHashMap` instead of an `HashMap` as in 2.1. This not only because it's thread safe, but also because we still want to access the loaded images while the loading is occurring;
- `ImageListPanel` now will show the list of the images contained inside the directory right after the user has selected it. It doesn't wait to receive the notifications about the loaded images anymore. In addition to the `JButton` for the sequential loading, now there's also a `JButton` for parallel loading;
- `PanelsController` now also installs listeners to allow parallel image loading without freezing Swing GUI thanks to a Swing Worker that runs the method `model.loadParallel()` in background. However, before starting this, it will disable every button to prevent the user from starting undesired tasks. We've also modified the listener for the `JTable` inside `ImageListPanel`: whenever the user selects an image from there, and the image in question it's not loaded yet, the controller will attach `DisplayImagePanel` as `Observer` on the model with the name of the selected image as parameter. So, right after when the image is loaded, `DisplayImagePanel` will display it;
- `DisplayImagePanel` now implements the `Observer` interface to be notified when an image is loaded. Inside its `update()` method it will also detach itself from observing the model. Since this will happen inside a parallel thread (as seen in Algorithm 1, line 7), it's necessary to use a `ConcurrentLinkedQueue` to store the Observers for an Image. Otherwise Java will throw an exception.

Other than this, to achieve parallel implementation and improve the functionalities of our software, new classes and some other views were created:

- `ParallelSwingWorker` is a class that allows the controller to run the method `model.loadParallel()` in background. This way, the GUI will not freeze anymore when the model loads the images;
- `StatsPanel` displays the performances of both sequential and parallel implementation;
- `BenchmarkPanel` allows the user to test performances by selecting different number of threads to use and then showing the statistics of

[1] "index" is the index of the first element to be loaded and "offset" the number of images that the thread has to load.

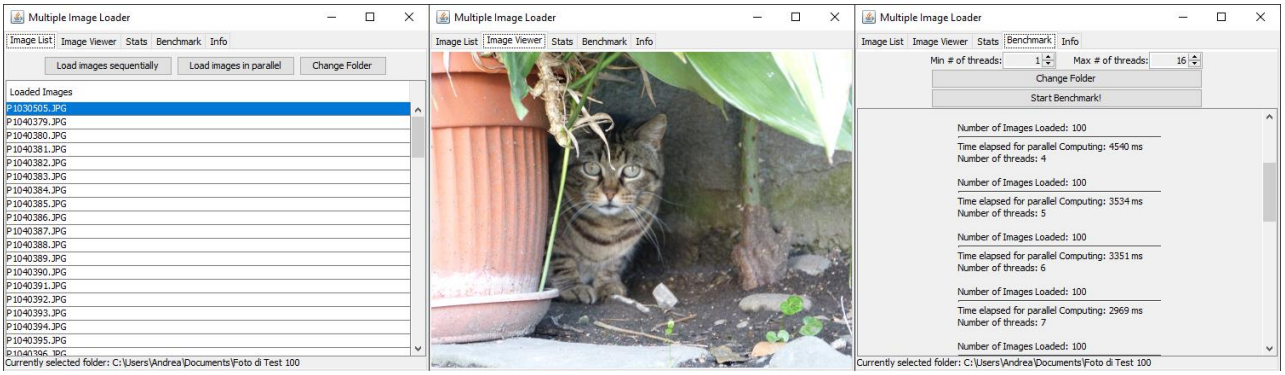

Fig. 4 - The `ImageListPanel`, `DisplayImagePanel` and `BenchmarkPanel` of the final version of our software

each loading. During benchmark, notifications are suppressed.

Thanks to this implementation, the user can select the image to display also during the loading too, and it will be showed when available.

3. Results and performances comparison

We tested the software on a computer with Intel® Core(TM) i7-9700K CPU with a frequency of 3.60GHz. The sample used consists of 100 images approximately 4MB each. To allow our software to load all the images, it was necessary to add the `-Xms1024M -Xmx8192M` flag to VM run configuration, otherwise the memory required to load all the images would not have been enough.

3.1. Sequential

We've done several tests with different number of images: from 10 to 100.

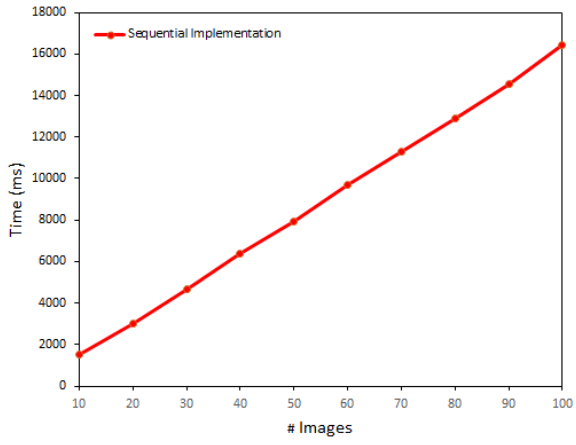

Fig. 5 - Execution time of sequential implementation

As can be seen from Figure 5, the time required to load the images, increases linearly with the number of these.

3.2. Parallel

Firstly, we've done the same tests as the sequential version fixing the number of threads to 8. The time required to perform the same operations drastically decreases.

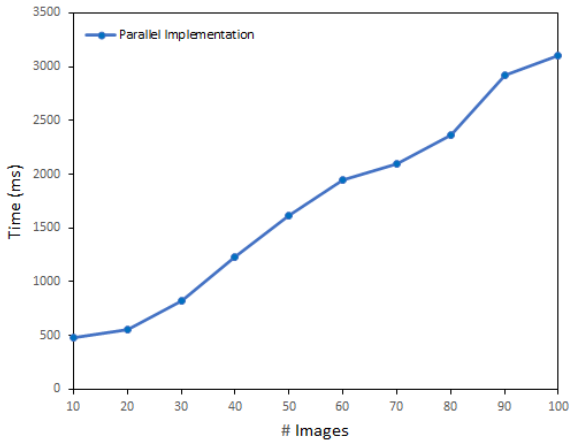

Fig. 6 - Execution time of parallel implementation

We've later fixed the number of images to 100 and, using the Benchmark View, we've tested the parallel version with different number of threads starting from 1 (which took about the same amount of time as the sequential implementation), up to 100.

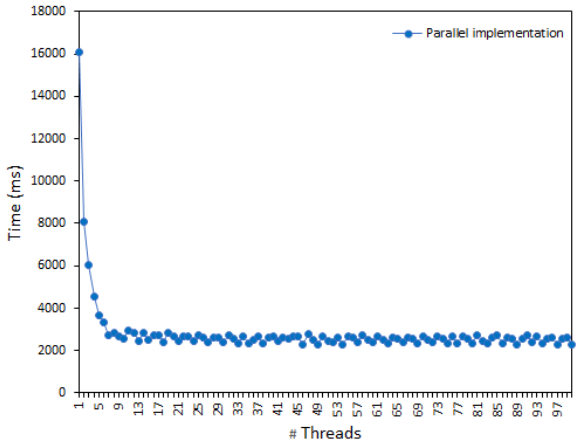

Fig. 7 - Execution time of the parallel implementation with different threads

As seen in the figure above, from eight threads onward the time required is almost the same

3.3. Performance comparison

Combining the two graphs from Fig. 5 and Fig. 6, we can compare the performances between the two implementations of our software.

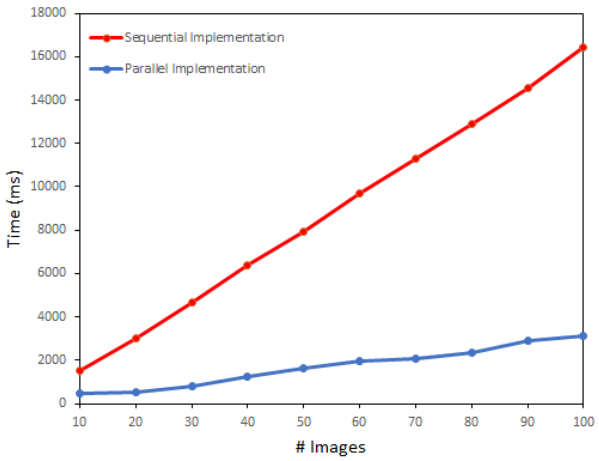

Fig. 8 - Comparison between the two implementations

Using this data, we calculated the speed-up factor defined as $S_p = \frac{t_s}{t_p}$ where t_s is the time required for sequential implementation and t_p the time required for the parallel one. This is shown in the graph below:

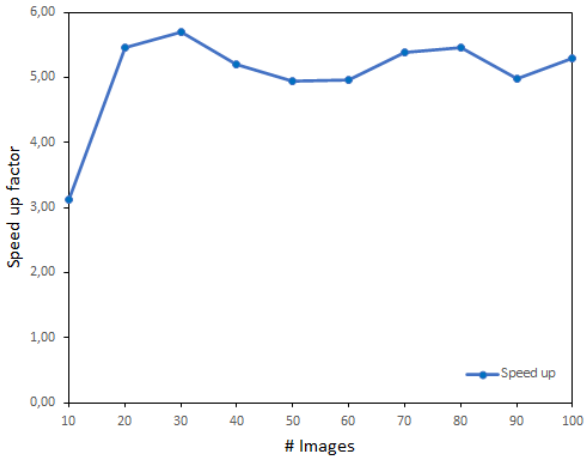

Fig. 9 - Speed up factor between the two implementations

4. Conclusions and observations

From the tests described in the previous paragraph, we can say that, with a great number of images, the parallel version is at least 5 times faster than the sequential implementation.
We've also noticed that, from 8 threads ahead, the performances are almost the same: that's because the computer used for testing has 8 physical (and logical) cores and, using the System Monitor, we can see that those are actually used all together.

An interesting variant of our project could be reverting the logic used to display the images: instead of notifying the view when the image selected by the user has been loaded, notify the threads that the user has selected a specific image and load it immediately. A possible way to do this could be to identify which thread has the task to load the selected image and notify it (even with one interrupt), in order to give that priority to that over the others.

4.1. References

[1] JetBrains. *IntelliJ Idea* *https://www.jetbrains.com/idea/*

[2] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Pearson Education 2008

[3] Java Tips Weblog. WrapLayout https://tips4java.wordpress.com/2008/11/06/wrap-layout/