Parallel Computing project assignment 2019-2020

Project Title: Multithread Java JPEG Image Reader

Student: Francesco Areoluci

Credits: 9 CFUs

Abstract

The purpose of the proposed project is to develop a Java application that can be used to asynchronously load images from a directory in order to accomplish image elaboration during the loading process. This approach can be used to reduce the amount of processing time by avoiding the synchronus wait of the images loading. The proposed developed application explores a variety of multithread approaches of the Java language: from the simple usage of Runnables implementation to Callables execution with the usage of a Tread Pool.

1 Introduction

The developed application can be used to load images from a requested folder. The process of file loading is a costly operation due to the limited read performances of the physical memory where these files are stored. In fact, even using an SSD (Solid State Disk), the read operation performance is much slower than a read operation performed on RAM (Random Access Memory). Moreover, the RAM bursting and caching mechanisms provide a considerable speedup in memory reading/writing. Physical disks don't allow parallel read/write operations, thus the applications that intensively use the disk will be always limited in performance by the its I/O.
An approach to this problem, which can be used in image processing applications, is to perform an asynchronous loading of the images. Using an asynchronous approach one or more thread will be responsible to load images in memory, while other threads will consume these images to process them. This solution can be used to avoid the wait of image loading and perform operations on these images while the loading is still in progress.
The Java language enables the developer to use a variety of multithread solutions in order to accomplish the desired results. In this application the following approaches have been explored:

- **Runnables**: this interface can be used to develop a simple thread usage
- **Callables**: this interface can be used to implement a thread process which result can be retrieved anytime using Futures
- **Thread Pool**: a mechanism that can be used to manage thread creation and task execution
- **ConcurrentHashMap**: a concurrent data structure that handle the concurrent read/write accesses
- **Atomics**: lock-free variables that can be shared between threads and accessed without a synchronization mechanism

2 Java multithread

In the following chapters, the explored multithread approaches offered by the Java language will be described.

2.1 Runnables [1]

A class implementing the Runnable interface can be used to be executed by a thread. To implement this interfaces, the class must implement the run method, which contains the code that should be executed by the thread.
A runnable class can be defined as the following:

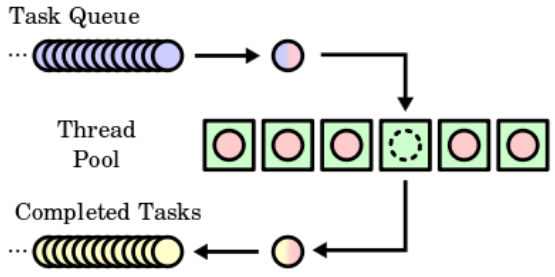
```
public class MyRunnable implements Runnable {
    public MyRunnable() {}
    public void run()
    {
        ...
    }
}
```

The runnable object can then be used to start a thread:

```
MyRunnable r = new MyRunnable();
Thread t = new Thread(r);
t.start();
t.join();
```

2.2 Thread Pool Executor [2]

Thread management can be improved using Java Executors.
A thread pool can be used to improve the thread management and the thread invocation overhead of the application. Threads are instantiated when the pool is created. Tasks can be submitted to the pool, which uses the instantiated threads to accomplish these tasks. Tasks are queued and once a task is completed, the thread return to the pool and can be reused.



Thread pool are particularly useful to execute a large number of asynchronous and small tasks. Java offers thread pool functionalities with the ExecutorService and ThreadPoolExecutor classes.

2.3 Callables [3]

The Callable interface can be used to implement a class that can be used to start a thread. To implement the interface, the class must override the call method. This interface is similar to the Runnable interface, but the call method can return a value, which can be retrieved using Future. Moreover it can throw exceptions.

A callable class can be defined by implementing the Callable interface.

```
public class MyCallable
implements Callable<String> {
    public MyCallable() {}
    public String call()
    {
        ...
    }
}
```

A callable can be executed using an executor. The return value can be get at any time, by having the callable object reference and using Future mechanism.

```
MyCallable c = new MyCallable();
ExecutorService executor =
Executors.newFixedThreadPool(1);
Future<String> future = executor.submit(c);
String result = future.get();
executor.shutdown();
```

2.4 ConcurrentHashMap [4]

This class implements a concurrent data structure that handle the read/write accesses and synchronization.
It is an hash table and support all the operations of the Hashtable structure, but it is thread-safe. Read operations can be done without locking and write operations lock only a subset of the entire structure. The map can be used as the same as an Hashtable.

```
ConcurrentHashMap<int, int> map =
new ConcurrentHashMap<int, int>()
int value = 3;
int key = 5;
map.put(key, value);
```

2.5 Atomics [5]

Atomics can be used to implement a concurrent lock-free variable read/write access.
Using atomics, threads can concurrently access the variable without the need of synchronization.
Atomics are implemented for the primitive types, and can be used as follow:

```
AtomicInteger aInt = new AtomicInteger(0);
int newValue = aInt.incrementAndGet();
```

3 Developed Application

The application is composed of the following Java classes:

- **ImageReader**: main class
- **BenchmarkSuite**: static class used to perform benchmarks
- **ImageLoader**: class responsible to load jpg images from a directory
- **ThreadPool**: implement a shared thread pool
- **Image**: utility class used to perform image processing (grayscale conversion)
- **ImageLoaderThread**: runnable implementation to perform async loading
- **ImageLoaderCallable**: callable implementation to perform async loading
- **ImageProcessingCallable**: callable implementation to perform image processing

The **ImageReader** is the main class and uses the **BenchmarkSuite** static class' methods to perform analysis over the developed classes.

ImageLoader class provides methods to sequentially and parallelly (synchronously and asynchronously) load the images from a requested system path. The parallel loading can be requested in the following ways:

- Simple Runnable usage
- Runnable with a thread pool
- Callable with a thread pool

Images are loaded in memory using the ConcurrentHashMap field of the class. Using this approach, threads can read/write from/to the same data structure without the need of external synchronization, which is implemented by the hash map. Moreover, this class offers methods to request a loaded image using its path and to pop an image from the map. The class uses ImageIO to load BufferedImages in memory.

ThreadPool class offers an interface to use a ThreadPoolExecutor, implemented with a fixed pool size. A ThreadPool object is shared among users by implementing the Singleton pattern. Through the use of this class, users can request to submit Runnables and Callables in order to start parallel work.

ImageLoaderThread and **ImageLoaderCallable** classes implement the image loading using a reference to the ConcurrentHashMap and the list of images that each thread should load. Moreover, the class ImageLoaderThread uses references to atomics of the **ImageLoader** class in order to signal the completion of the loading. This approach can be used to start image processing asyncronously with respect to the image loading process.

Image class offers method to perform an RGB to grayscale conversion of a BufferedImage. The grayscale conversion can be done in two different ways: a faster

implementation which evaluate the mean value of the R,G and B channels for each pixel and a slower (but more accurate) implementation which uses weighted luminance evaluation and gamma expansion

ImageProcessingCallable class is a Callable implementation that can be used to parallelize the grayscale conversion over multiple images.

4 Results

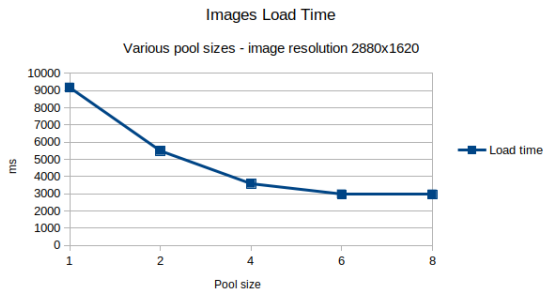
The performances of the application have been analyzed in order to evaluate use case scenarios. The following performance parameters have been evaluated:

1. Images loading time. Performed using a variable number of threads
2. Total execution time of images loading and processing. Performed using a synchronous load/parallel processing and asynchronous loading/parallel processing.
3. Total execution time performed using different thread pool sizes
4. The improvements on total execution time using more than one loader thread.
5. Performance of asynchronous loading/parallel processing vs sequential load/sequential processing.

All the tests are executed on an Intel I7-7700HQ processor and a Crucial MX500 Solid State Disk.

1. Stored image loading time

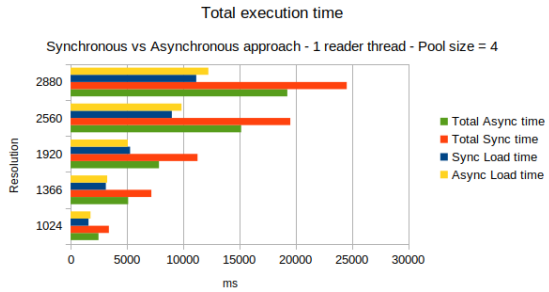
This test scenario has been evaluated to understand how the application performs in the image loading. To perform this test, 200 images with resolution 2880x1620 are loaded in memory. The number of threads used to perform the loading varies from 1 to 8.



The results show a significant gain in the loading time as the number of threads used to perform the loading increases. The I/O bound is reached using 4 threads.

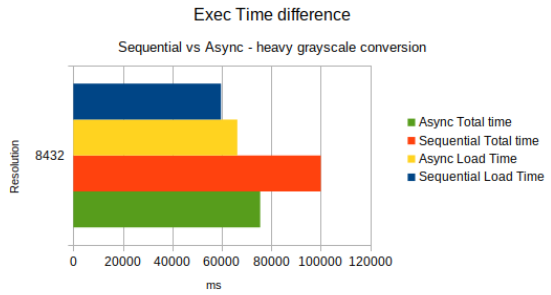
2. Synchronous vs asynchronous approach

This test scenario has been executed to evaluate the advantages of doing image processing during the loading.
The test is executed using a thread pool size equal to 4 and various image resolutions have been used, from 1024x760 to 2880x1620. 200 images per resolution are loaded in memory. The analyzed approaches are synchronous sequential loading/parallel processing and asynchronous loading/parallel processing. The images are loaded using one loader thread.
The processing is done by evaluating the average channels' pixel value. This is done in order to have comparable load and processing execution times.



The results show an improvement on the total execution time thanks to the elaboration performed during the image loading.
As we can see, about an half of the sequential load time is saved in the total time of the asyncronous approach.
The asynchronous processing shows that

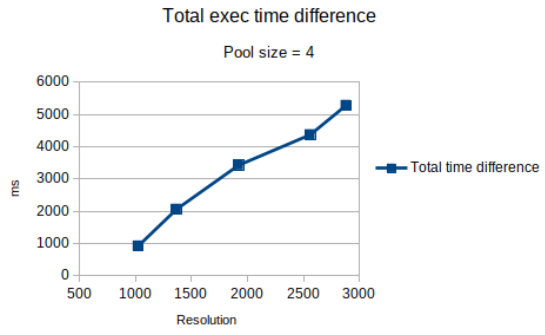
a significant amount of time can be saved as the load time increases. As an example, 16 images of resolution 8432x6168 has been loaded in memory and the weighted channel grayscale conversion has been applied to them.



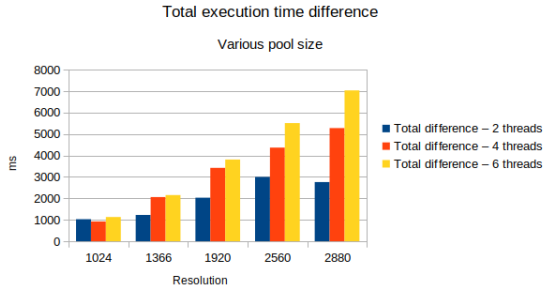
The result shows an improvement of 25 seconds on a total of 100 seconds of synchronous execution.

3. Variable pool size

This test scenario has been executed to understand how much the variation of the number of processing threads affects the execution time.
As previously described, the image resolution affects the execution time.



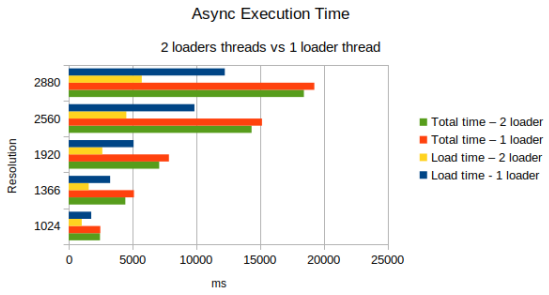
A way to improve to execution time is to increase the pool size, and so the number of tasks that can be done simultaneously.
The executed test is done on the same sets of images using a loader thread.
The differences between the synchronous and asynchronous total time has been evaluated.



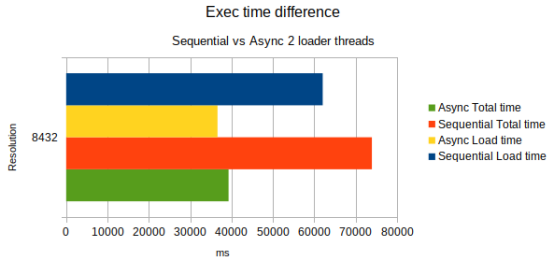
As we can see, the increased number of processing threads affects the total execution times: the improvement is noticeable in higher resolution images.

4. Increase the number of loader threads

As described in test **1**, increasing the number of loader threads can improve the load time of the images.
The same approach has been used to evaluate the performance improvements of using more than 1 loader thread to perform the same test on the same set of images.
The performance of async processing with one and two loader has then been evaluated.



As we can see, the load time is greatly reduced but the execution time show little improvements. This behaviour is caused by the fact that, during the load time, less thread will execute the images processing. So the loader is faster, but the processing is slower.
This approach can be used when the load time is predominant over the processing time. As an example, 16 images of resolution 8432x6168 has been loaded in memory and the average channels grayscale conversion has been applied to them.

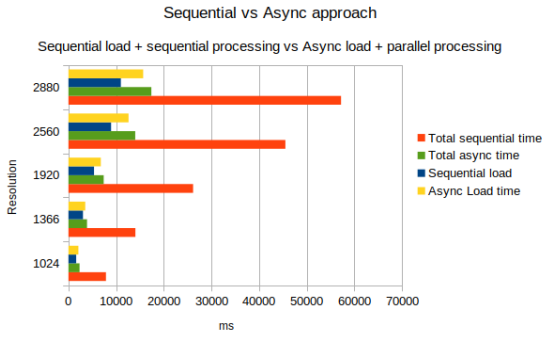


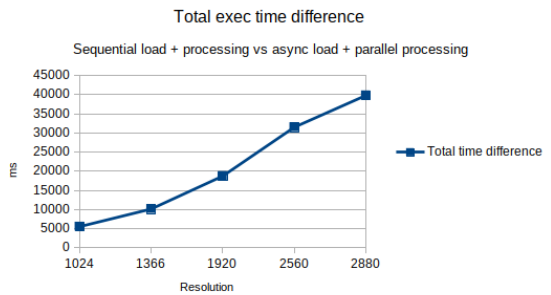
The result shows a great improvement in the execution time: the load time is greatly reduced and the processing time is almost completely absorbed by the load time.

5. Parallel async vs sequential load and processing

The last evaluated scenario has been executed to evaluate the performance of the async approach over the traditional sequential loading and sequential processing. This is done to understand how much faster the parallel version of the application performs.

The test has been done over the same set of images with 1 thread loader in a pool of 6 threads.





5 Conclusions

The developed application allows to syncronously and asyncronously load images from the system storage and to perform processing on these images. The Java language offers a lot of functionalities and abstraction to implement multihread processing. These functionalities have been explored to implement the application. Through the executed tests, the performance of the asyncronous approach has been evaluated. These tests show performance improvements on the execution time, which become particularly relevant when the image resolution increases.

6 References

[1] https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html

[2] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadPoolExecutor.html

[3] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html

[4] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html

[5] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html