Parallel implementations of an image reader using *C POSIX thread* library and *Java Concurrency* framework

Marco Benelli
E-mail address
marco.benelli@stud.unifi.it

Luca Bindini
E-mail address
luca.bindini@stud.unifi.it

Abstract

In this paper we will analyze the performance of a simple image reader implemented sequentially and in parallel, synchronously (using pthread *in* C *and old* Java *framework) and asynchronously (using* Java Concurrency *framework).*

1. Introduction

A typical operation that is performed on a modern computer is to read an image and possibly perform some operation on it. The main purpose of this paper is to describe several implementations of a simple image reader in *JPEG* format that operate both sequentially and in parallel.

The languages that have been used are *C* and *Java*, in particular in *C* the *POSIX thread* library has been used for the realization of the parallel version of the reader in a synchronous way while in *Java* the low level framework has been used for the realization of the parallel version synchronous version of the reader and the *Java Concurrency* framework for the asynchronous parallel version. As for the actual reading of images from disk in *C*, the *OpenCV* open source library was used, while in *Java* the standard library class was used, which already offers various tools for managing images.

To evaluate the performances, high resolution images were used both at *1080p* and in *4k*.

2. C version

In the *C* version, we make use of two different functions to read the images. The first function does the actual reading from disk, while the second one returns the array of images.

In the standard library there aren't ways to read an image, so we used the *OpenCV*[1] library, and more specifically the `imread` function.

The code for this version has actually been written in *C++*, since *OpenCV* is a library for *C++* and not *C*. However, we are still calling this the *C* version because the reader itself doesn't use any *C++* features, but mostly because we used the *POSIX threads* library to handle the threads instead of the tools found in the standard library since *C++11*.

The standard library headers we included are `<chrono>` (introduced in *C++11*) and `<filesystem>` (introduced in *C++17*). The former was used to evaluate the performances of the different implementations, while the latter to iterate on directory elements to get the image names.

One final header we used is `<unistd.h>`, another *POSIX* header that we used to get the number of processors of the machine.

2.1. Sequential implementation

The sequential implementation is obviously the simplest of the two. All it does is, given a vector of strings (the image names), it creates an array containing all of images in matrix form. Therefore, the more interesting discussion is around the interface.

We wanted to have a similar interface across all versions (meaning *C* as well as *Java*), this is why we created two functions: `sequentialRead` and `getImages`. The first function does the heavy lifting and puts the array of images in an external static variable, which is then returned by the second function.

2.2. Pthread implementation

The *pthread* version is more complex. Analogously to before, the `main` function can interact with this implementation via two functions: `parallelRead` and `parallelGetImages`. The first one now takes an extra argument compared to the sequential implementation, which is the number of threads to be created.

Since this is the parallel implementation, `parallelRead` has to create the threads and join them. Here follows its code. Keep in mind that the arguments are `str_` (the names of the image files) and `n_` (the number of threads). On the other hand, `imgs`, `str` and `n` are external variables. They had to be external because the threads need access to them.

```
imgs = new Mat[str_.size()];
str = str_;
n = n_;
auto *t = (pthread_t *) malloc
    (n * sizeof(pthread_t));
for (int i = 0; i < n; i++)
    pthread_create(&t[i],
                   nullptr,
                   imgRead,
                   (void *) i);
for (int i = 0; i < n; i++)
    pthread_join(t[i], nullptr);
free(t);
```

As we can see, when we create each thread, the function that they will execute is `imgRead`, so let's look at its code.

```
int myIdx = (int) idx;
for (int i = myIdx;
     i < str.size();
     i += n)
    imgs[i] = imread(str[i]);
return nullptr;
```

This code shows that each thread reads a non contiguous subset of images. The fact that it's not contiguous isn't important in this case, since we are not dealing with a *SIMT* architecture, we chose to do it this way just because it was the most convenient way.

3. Java version

Like the *C* version, we make use of two different functions to read the images: `read` which does the actual reading from disk and `getImages` which returns the array of images.

To read the images the standard *Java* library was used directly, in particular the `BufferedImage` class (which is present in `java.awt.image`) and the `ImageIO` class which allows you to read an image from a file through the `read` method.

The `System` and `Runtime` classes were also used in the *Java* code.

The first one was useful for the purpose of measuring the times through the `currentTimeMillis` method, while the `Runtime` class was useful for determining the number of processors present on the running machine thanks to the `availableProcessors` method.

3.1. Sequential implementation

The sequential implementation, implemented in the `SequentialImgReader` class receives a list of files to be read (through the `read` method) and read them with a for loop sequentially. The images are finally returned through the `public` method `getImages`.

3.2. Synchronous implementation using low level framework

The synchronous version was created using the *Java* low level framework for parallelization. The public `read` method in this implementation instantiates a certain number of

`ImgReaderThread` objects (inner class that `extends` the `Thread` class), passed as a parameter to the constructor of the class itself, and starts them through the `start` method.

The `start` method will execute the `run` method of the `ImgReaderThread` threads in which each thread will read the images assigned to it, the various threads divide the images to be read. Finally, the `read` method will `join` all threads to wait for them to complete.

Here below the body of the `read` method:

```
imgs = new BufferedImage
    [str.length];
t = new ImgReaderThread[n];
for (int i = 0; i < n; i++) {
    t[i] = new ImgReaderThread
        (imgs, str, n, i);
    t[i].start();
}
for (ImgReaderThread irt : t)
    irt.join();
```

3.3. Asynchronous implementation using high level framework

The asynchronous version was created using the *Java* high level framework for parallelization, namely *Java Concurrency*.

The `public` method `read` in this implementation instantiates as many objects of type `Future`, present in `java.util.concurrent`, as there are images to read.

Then it instantiates an object of type `ExecutorService`, passing it as a parameter the number of threads, which will be responsible for the execution of the `ImgReaderThread` threads (inner class that `implements` the `Callable` interface). All threads are then `submit` to the executor and returned one `Future` per image which will be inserted into the `imgs` array.

It will therefore be up to the `main` to invoke on the `Future` objects, returned through the `getImages` method, the `get` method to obtain the image they read.

Here below the body of the `read` method:

```
imgs = new Future[str.length];
exec = Executors
    .newFixedThreadPool(n);
for (int i = 0;
     i < imgs.length;
     i++)
    imgs[i] = exec.submit
        (new ImgReaderThread
        (imgNames[i]));
exec.shutdown();
```

4. Performance

Now that we have all of these different implementations, we can put them to the test and see how they all perform.

4.1. The setup

All of the implementations were run on the same machine, one with an *Intel Core i7-4770*, a CPU with 4 cores and 8 threads.

The images we used had a resolution ranging from 1080p to 4K and the average size of an image was 856 kB, when compressed with the JPEG format.

For the time measurements, as we said, in the *C* implementation we used `system_clock` from the `<chrono>` header (a *C++11* addition). On the other hand, in *Java*, we used `currentTimeMillis`, a function with the same precision as for the *C* version.

For the times we had to use the number of threads of the CPU, instead of writing it by hand, we got the number at compile time. The way to do that in *C* (when running *Linux*) is including the `<unistd.h>` header and writing:

```
sysconf(_SC_NPROCESSORS_CONF)
```

In *Java* we can achieve the same effect, independently of the operating system, with:

```
Runtime.getRuntime()
    .availableProcessors()
```

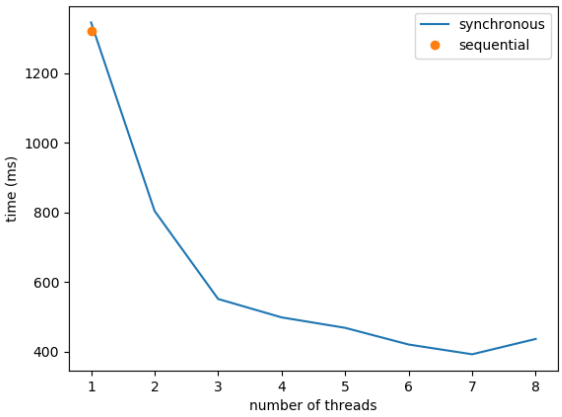

Figure 1. The times taken by the *C* implementations. On the x axis the number of threads, while on the y axis the times.

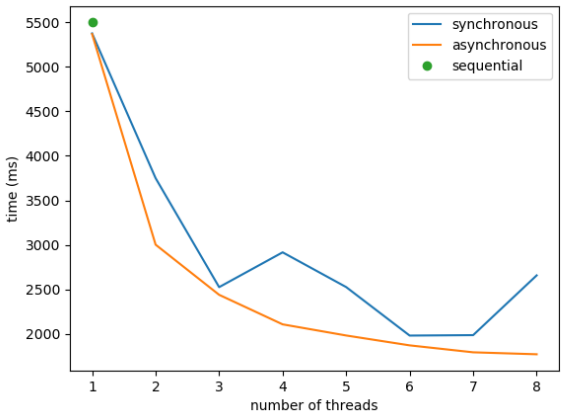

Figure 3. The times taken by the *Java* implementations. On the x axis the number of threads, while on the y axis the times.

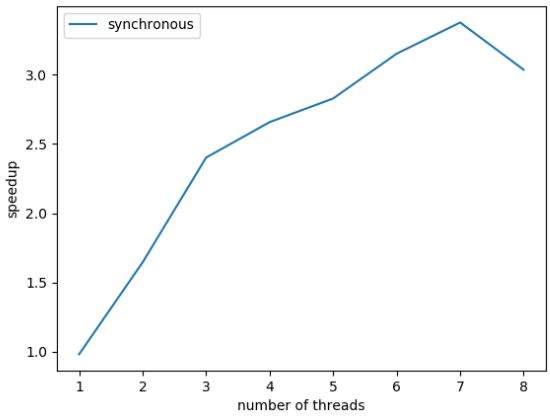

Figure 2. The speedup of the *pthread* implementation. On the x axis the number of threads, while on the y axis the speedup.

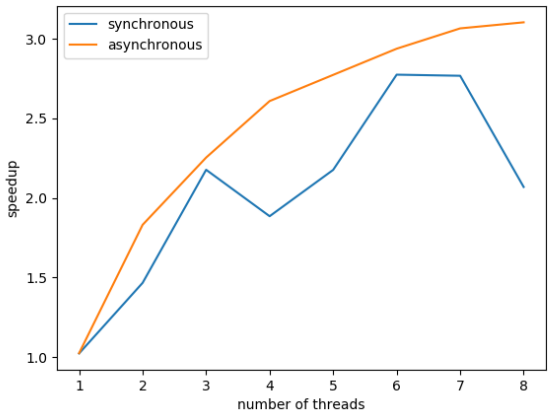

Figure 4. The speedups of the parallel *Java* implementations. On the x axis the number of threads, while on the y axis the speedup.

4.2. Comparison of the different implementations

We'll start the comparison by looking at the *C* version. The graphs in figure 1 shows the time performance while varying the number of threads. The number of threads for this plot and the subsequent goes up to 8 because that is the number of threads of the CPU we used. The number of images used for this test was 64. Figure 2 shows the same information, but looking at the speedup instead of the absolute times.

Similarly to the *C* version, figures 3 and 4 show the absolute times and the speedups of the *Java* version. As we said, in *Java* we have also implemented the asynchronous approach, which seems to be faster than the synchronous one. Like before, the number of images read in these tests is 64.

In the final tests we'll look at how the time changes when we vary the number of images read. Figures 5 and 6 show what happens in *C* and in *Java* respectively. For the parallel implementations, we always used as many threads as the CPU has, meaning 8 in our case. The number of images we tested goes from the number of threads used in the parallel case (8 in our measurement) to 512 (*C*) or 256 (*Java*), always doubling. We chose to stop where we did because if we had continued, we would have filled the machine's RAM.

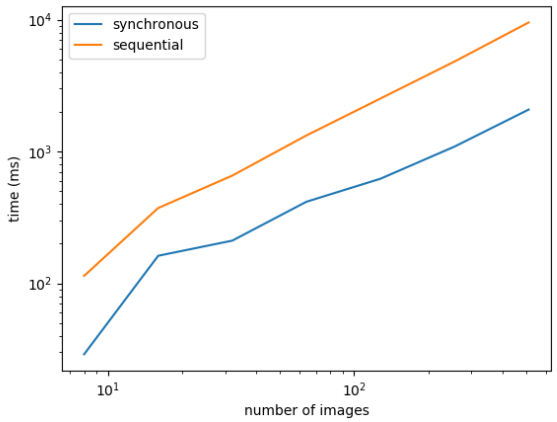

Figure 5. The times of the *C* versions. On the x axis the number of images and on the y axis the time. The number of threads used for the parallel version was the number of CPU threads.

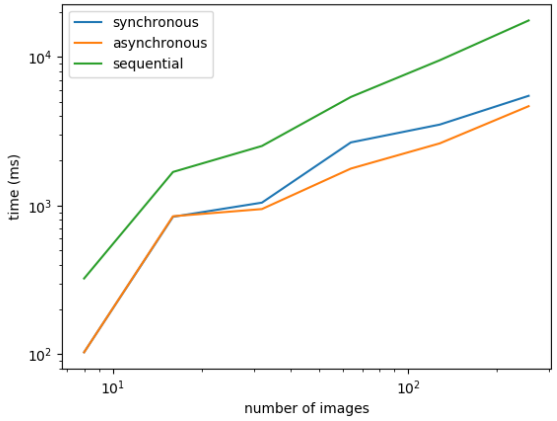

Figure 6. The times of the *Java* versions. On the x axis the number of images and on the y axis the time. The number of threads used for the parallel versions was the number of CPU threads.

5. Conclusions

We have seen the image reader implemented in various ways and using different technologies. Looking at all of the implementations, we clearly see that the *C* implementation is the fastest, with its sequential version beating even the parallel *Java* version. This means that if speed is important for a given application, you should try to use *C*, with parallelization if possible. If you instead decide to go with *Java*, the asynchronous implementation using the high level framework is both the fastest and the more convenient.

References

[1] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.