COMP3003 Assignment 1 Report

Kristian Rados (19764285), Semester 2 2021

Multithreading Design Explanation

There are three classes which are responsible for starting their own threads: these are the `FileFinder`, `FileComparer`, and `ResultFileWriter` classes. Each of these have their own respective threads encapsulated within the class, with a `start()` method used by the calling class to initiate the thread creation process and consequent code execution. The classes responsible for externally calling these `start()` methods are:

- `FileComparerUI`, starting the `FileFinder` thread, and
- `FileFinder`, starting the `FileComparer` and `ResultFileWriter` threads.

Furthermore, `FileComparer` initialises the `comparisonService` thread pool which creates a number of threads equivalent to the number of available processors.

The `FileFinder` thread is used to recursively find each valid text file in the user-selected directory. The `FileComparer` thread takes those filenames, calculates the max number of comparisons for the progress bar, and iterates through each file to select a "primary" file. For each primary file, a `Callable` task is submitted to the executor service to compare it with all other "target" files that wouldn't result in redundant comparisons. Finally, the `ResultFileWriter` thread takes the results of any comparisons over the minimum similarity threshold and concurrently writes them to the `results.csv` file.

`FileComparer` and `ResultFileWriter`, as a producer and consumer respectively, communicate and share resources through a blocking queue of `ComparisonResult` objects. Each generated comparison is placed into the queue by one of the producer executor service threads, with each placement blocking any other producer or consumer threads from accessing the queue until this one has finished. The next waiting thread is then notified to wake up and place/take its object while placing its own block on the queue, preventing race conditions and deadlocks. Once all comparisons have finished, the producer places a poison object at the end of the queue. This notifies the consumer, `ResultFileWriter`, that the producer has finished, so that it knows to close the `results.csv` file and end itself once all comparisons have been written.

`FileComparer` only submits the poison object once all comparisons, spread among the many executor service threads, have finished. This is done by block-waiting for each of the `Future` objects returned by the `Callable` `comparisonService` tasks to return its own value, with `.get()`, which occurs when that comparison task finishes. As the poison object is added to the blocking queue, `comparisonService` is also shut down, ending all of the comparison threads. The `FileFinder` thread ends as soon as it has finished finding valid text files and has started the producer and consumer threads; it does not wait for those threads to finish.

Scalability Architectural Issues

To ensure that the program scales well, we must ensure that the multithreading strategy used allows for the most efficient use of scaled resources, such as processing power and memory, to accommodate for the increase in input data or users.

Non-functional requirements

1. The time taken to process any arbitrary sequence of comparisons should scale linearly with the number of available processors (e.g. x comparisons should take roughly twice as long to process with 16 processors than with 8).
2. Comparisons with similarity values above the threshold should be displayed to the user within x milliseconds of the comparison finishing (e.g. 1000ms).
3. An error should occur less than $x\%$ of times a series of comparisons is started or stopped by the user (e.g. 1%).
4. The GUI should should respond to user input within x milliseconds (e.g. 500ms).

Problems with very large input data

A very large amount of input data means that the number of comparisons required grows exponentially according to the formula $c = 0.5 \cdot (f^2 - f)$, with the time taken to do these comparisons also growing as the size of the compared files increase. Because of this, the strategy of multithreading used, if any, significantly impacts the processing time needed. As an example, my current implementation assigns an executor service task by iterating through each file, labelling it as a "primary" file, and comparing it with all other as-of-yet uncompared "target" files within that same task. But what if there are 16 processors available for running comparisons, and only 8 very large files to compare with each other? 9 of the available processors $(16 - 8 + 1)$ will remain unutilized, not having been allocated a task, but each allocated task will take a significant amount of time, bottlenecking the application.

Another issue with my multithreading implementation is that it misses an opportunity for multithreading that may cause issues at a large scale: file reading and file comparison are done sequentially in the same thread.

Furthermore, the JVM may run out of memory when dealing with very large files, as each file is read into a `String` object on the heap.

Finally, if the directory chosen by a user has a very large number of small files that are above the similarity threshold, then in my current implementation the `TableView` in `FileComparerUI` would be rewritten many times in a short period of time due to the rapid comparisons, ruining the responsiveness of the GUI and potentially making the user think that the program has crashed when it has not.

Re-engineering to address these problems

The aforementioned utilisation issue caused by large files could be alleviated by having the parent comparison task adjust how it allocates tasks to the executor service depending on the average file size and the difference between the number of files and the number of available processors. In the example with 8 files and 16 processors for instance, it may split each child comparison task into two, doubling the number of threads used. Instead of a single task comparing a single primary file with all other target files, there will be one task comparing the primary file with the first half of target files and another task doing the second half. Instead of halving, the division may be done according to some ratio calculated on the fly.

Another seemingly obvious area for improvement would be to separate the reading of target files, and the comparison between those files and the primary file, to not be done within the same thread. This could be achieved with a file reader class that acts as a producer, reading files into strings that are added to a shared collection of strings locked with a mutex or monitor. In this case, the pre-read files could be accessed by the various comparison threads at any time. This should improve the speed of the program. However, with a collection of strings acting as a cache and each comparison thread needing to make a

local copy of the strings to prevent a blocking bottleneck, we encounter horrible memory issues far earlier than we would have before.

On the other hand, it appears to me that having the file reader repeatedly produce file strings to be consumed through a blocking queue would be redundant, as a comparison executor service task can't continue with any other processing until the target file has been read anyway, as it would have been sequentially in the original design, defeating the point of using a separate thread. Furthermore, there would be the same amount of redundant reading of the same file many times over as there was before.

To address the GUI responsiveness issue caused by many rapid comparisons, the `TableView` could be rewritten to the display in blocks rather than for every single comparison worthy of being seen by the user. For instance, it could be rewritten every 100 milliseconds or every 10 valid comparisons, whichever comes first. However, this may require testing if you wanted to get it just right, as a balance would have to be found between how smoothly the table is updated and how much it affects the responsiveness of the GUI.