EXAM
DM519 Concurrent Programming

Dennis Andersen – deand17

May 6, 2018

Methodology

This software project uses a general and simplistic approach to applying concurrency to a number of search tasks. Although this implies a trade-off on performance, noticeable performance gains over a sequential execution are nevertheless achieved. The implementation aims to leverage Java's concurrency features to do much of the heavy lifting.

The final implementation uses a similar approach to handle the execution of all three methods. Calling one of the methods `m1`, `m2` or `m3` initiates a recursive search of the root directory. The search is concurrent, and finds all sub directories and relevant files (`.txt` for `m1`, `.dat` for `m2` and both for `m3`). For each file found, an appropriate search method is initiated, and submitted as a task to an `Executor`. The tasks submitted are kept in a thread safe collection of `Futures`. Meanwhile, the main thread waits for tasks to finish, and as they do, the returned values are polled from the collection of `Futures` and either collected or, under the right circumstances, returned immediately. Once all `Futures` for a method have been processed, the corresponding `Executor` is shut down.

Concurrency in the software is handled by primarily making use of Java's `ExecutorService`, `Futures` and `CountDownLatches`. In addition, `ConcurrentLinkedDeques` are used to hold `Futures`.

Using `Executors` and `Futures` provides a simple and easy way to manage an unknown, but presumably large number of tasks. `FixedThreadPool Executors` are used to make both the directory search and searching the contents of files concurrent. In this way, once the directory search is complete, resources are automatically redirected and applied to searching the contents of files found.

`Futures` are mainly used as a way of collecting results from tasks, without having to resort to using shared data structures. This is also the reason for the main thread being used as a *busy-wait* consumer to collect results from `Futures`.

`CountDownLatches` are used as a way of ensuring that `Futures` have been produced, before the main thread is allowed to begin consuming said `Futures`.

Advantages

The main highlight of the implementation is its very straightforward nature. The same general approach is used for each of the three `m` methods, making it easy to follow and understand what happens. The various directory search tasks, are organized into a single method. Other more specialized tasks, such as searching for specific contents in files, are instead separated into their own methods. This separation of functionalities, should help with the readability of the code, and also makes it easier to maintain.

Another advantage of the simplistic approach and separation, is that it allows the software to be easily adapted and expanded upon. The ability to handle other file extensions would be a simple addition to the directory search, and one of the methods for searching file contents, could also easily be adapted or used as a template to meet other search requirements.

In short, the performance benefits of concurrency can easily be gained, without much of the added complexity that often follows. In addition, because no shared data structures are used, thread safety is not a worry either.

Performance is also worth highlighting. The software will attempt to adjust to the number of processing cores in an effort utilize as much processing power as possible. Testing shows significant improvements in runtime. Going from single threaded execution to dual threaded, more than halves runtime, and—although less dramatic—gains are still noticeable as all processing cores are put to use (Fig. 1).

Lastly, reliability should be another strong point. The software has been stable and without errors. However, in the unlikely event of an error occurring, error handling should take care of all but the most obscure or unlikely case.

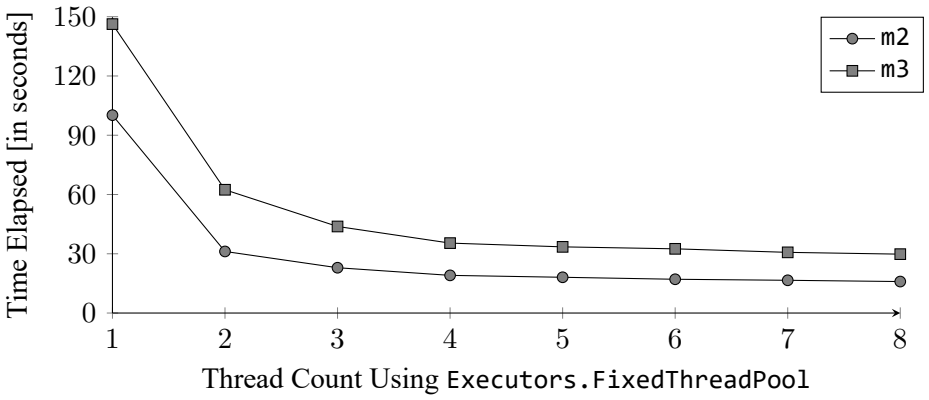

Figure 1 Results of running the software on a 6.9GB directory with 719 folders, 1920 similar sized `.txt` files and 6577 `.dat` files of varying size. The figure shows the runtime of methods `m2` and `m3` when using the `FixedThreadPool ExecutorService`, with input parameters 1–8, i.e. number of threads available. Going from 1 to 2 threads shows the largest improvement, more than halving runtime. Additional gains are noticeable up to a thread count of 4 (which was the core count of the test system), at which point diminishing returns set in. Method `m1` has not been included in this figure, as runtime in all cases was below 1 second.

Test Specifications *System:* Intel Core i7 2600K 3.4GHz (Quad Core), 16GB DDR3 1333MHz ram, 7200Rpm HDD. *OS:* Windows 10 Pro version 1709. *Software:* IntelliJ IDEA 2018.1.2, Java JRE: 1.8.0_152-release-1136-b29.

Limitations

From the simplistic approach to the implementation, the most obvious limitation that follows, is of course the sacrifice of performance. There are a number of places where a more specialized approach would likely yield noticeable performance gains over the current implementation.

One such place is in the main thread, concerning the use of *busy-wait*. Although the main thread is used as a consumer and the `Executor` as the producer, the workload of the consumer is more often than not minimal. This of course means resources are wasted waiting for tasks to finish. These would of course have been better spent on search tasks. A shared data structure could have been used to collect results or results could have been processed concurrently afterwards.

Possible gains are also likely with a different approach to the implementation of method `m2`. If we imagine a case where there is only *one* file which has the minimum line sum, the current implementation uses more time the deeper in the directory structure the file is. A more sophisticated approach could perhaps alleviate this problem.

On the subject of directory traversal, a `FileVisitor` could have been used instead of a `DirectoryStream`. During initial testing, the `FileVisitor` was found to be faster. However, not only was its implementation more complex and cumbersome, but it occasionally performed unreliably, and so ultimately the idea of using it was discarded.

Another obvious limitation of the software, is of course the narrow case where a directory consists of only a single file, in which case the software will run sequentially. A possible improvement would be to read different parts of the file concurrently and then collect the results, which should lead to a performance gain. Attempts at this solution were made, but were unsuccessful.

As a final note on performance, it is worth mentioning that memory consumption appears to be very high. Although specific measurements have not been made, monitoring Windows Task Manager indicated that memory consumption by IntelliJ IDEA increased from 1GB to around 2.5GB, while the software was running the tests depicted by figure 1. As the implementation only attempts to scale the software to the number of processing cores, it would be reasonable to assume that performance would likely suffer on systems with small amounts of memory.