DM519 Spring 2018

Thomas Stenhaug <tsten16@student.sdu.dk>

2018-03-22

1 Methodology

The central units of work in the project, are files containing a plaintext line of comma separated integer-values. I have chosen to make a wrapper-class `CSIFile` around Java's `Path`s to facilitate working with these files.

A trait shared between my solutions to `m1`, `m2` and `m3`, is that they instantiate a `BlockingDeque<CSIFile>` as the main shared datastructure for production and consumption. Initially, an executor for consumers, with one consumer for each processor, is created. A producer then runs from the main thread, producing to the `BlockingDeque<CSIFile>`. The producer runs in the main thread, because the operation is I/O-bound, so it didn't seem worth it to invest complexity in distributing the work over threads. (Although it uses `parallel()` in a stream-chain, just in case the JRE knows how to optimize it.)

While the main-thread is producing work-units, the consumers, one for each processor, accepts and performs their work.

1.1 Synchronizing when consumers are done

For `m1` and `m3`, the mechanism for synchronizing the consumers is a poison pill, that is fed to the queue of `CSIFile`s. The consumers gracefully exit their loop after encountering the poison pill, so after finishing producing work, the main thread shuts down the executor, and `await`s it, which will block until consumers are all done. In the case of `m1`, that concludes its job. `m3` instead, needs to do some additional computations on the collected data. These computations are run in parallel, governed by a separate executor. Here, the synchronization is implemented by waiting for said computations as futures, while constructing the final result.

`m2` is different, because instead of the result being an aggregate of all files, it is an arbitrary element from the `BlockingDeque<CSIFile>`, matching some criteria. This element is represented by a `CompletableFuture`, which is waited for in the main thread after production of `CSIFile`s is completed, and is completed by the first consumers that find an element matching the criteria. Consumers

exit either when they discover that the `CompletableFuture` is completed, or they are cancelled as a result of their executor being shut down.

1.2 Post processing

While `m1` and `m2` are done with their work when their consumers terminate, `m3` requires post-processing. Its consumers are also producers, adding work to a handful of concurrent datastructures.

2 Advantages

2.1 Choice of datastructures, performance

I chose to use "off-the-shelf" datastructures for this, instead of writing my own. For example, two return-values are the least and most frequently occuring number. I chose a `ConcurrentMap<Integer, LongAdder>` for this purpose, which I update like so:

```
frequencies.computeIfAbsent(i, k -> new LongAdder()).increment();
```

This results in lock-free, thread-safe operation on this high-contention data.

One of the return values from `m3` is an ordered `List<Path>`. Initially I thought to populate a `ConcurrentLinkedQueue` in the consumers, and obtain the ordering during the post-processing. This would be an $O(n \lg n)$ operation, in a single thread. Instead I chose a `ConcurrentSkipListSet`, which maintains $O(\lg n)$ for insertion, but with "for free" parallelization, since the operation is already spread across the parallelized consumers.

3 Limitations

Aliquam erat volutpat. Nunc eleifend leo vitae magna. In id erat non orci commodo lobortis. Proin neque massa, cursus ut, gravida ut, Mlobortis eget, lacus. Sed diam. Praesent fermentum tempor tellus. Nullam tempus. Mauris ac felis vel velit tristique imperdiet. Donec at pede. Etiam vel neque nec dui dignissim bibendum. Vivamus id enim. Phasellus neque orci, porta a, aliquet quis, semper a, massa. Phasellus purus. Pellentesque tristique imperdiet tortor. Nam euismod tellus id erat.