PCPP - Assignment 3

Dallas Trent Maxwell (dalm@itu.dk)
Martin Conradsen (conr@itu.dk)
Mohammad Toheed Sharif (mosh@itu.dk)

October 6, 2021

Exercise 5.2

Mandatory

The solutions for all the mandatory exercises (implementations and stream pipelines) can be found in the file *TestWordStream.java*. The stream pipelines and various tests are found in the `main` method.

6. It is possible to observe whether the parallel version of the palindrome-printing stream pipeline is faster, but the observation is not necessarily an accurate one. When timing each of the functions, the parallel version is consistently around 3x faster than the non-parallel one. However, as demonstrated in earlier lectures, these timings aren't exactly perfect.

Exercise 6.1

Mandatory

1. Verified. The results are found in the file *ex1.1_out.txt*.

2. Without the calculation, the program will deadlock in the case where one account is involved as both a source and a target in two different threads.
Example of an interleaving: Thread 1 wants to transfer from account 1 to 2. Thread 2 wants to transfer from account 2 to 1. Thread 1 acquires a lock on the source, which is account 1. Thread 2 acquires a lock on its source, which is 2. Thread 1 attempts to acquire a lock on account 2, but Thread 2 already has it, and Thread 2 attempts to acquire a lock on account 1, which Thread 1 has. The result is a deadlock.
When running with the other version of the code, the program sometimes deadlocks in the middle of a transaction, whereas the original 4 lines ensure (at least in our limited experiments) that the program always runs to completion.

3. The modified file is found in the submission (file: *ThreadsAccountExperimentsMany.java*).

4. The modified file is found in the submission (file: *ThreadsAccountExperimentsMany.java*).

Exercise 6.2

1. As expected, the most efficient run is with 8 threads, which makes sense on an 8-core system. At some points the executing takes more and more time with each new thread being added, which is probably a result of the time cost of starting a new thread.

`countParallelNLocal` is always faster than `countParallelN` with the same amount of threads. This also makes sense each thread has its own personal counter, which means that it won't have to wait on the other threads before incrementing it.

2. The modified file is found in the submission (file: *TestCountPrimesThreads.java*). The results are found in the file *ex2.2_out.txt*.
The results are very similar to the ones we achieved with the original code. This might be because of a wrong implementation.

Exercise 6.3

Mandatory

1. Solution is found in the file *SimpleHistogram.java*. The variable `counts` is final, and the methods `increment`, `getBuckets` and `getTotal` are synchronized. The method `getPercentage` doesn't need to be synchronized since `getTotal` is already synchronized, and `getSpan` doesn't need to be synchronized because `counts` is final.

2. Solution is found in the file *SimpleHistogram.java*.