Jeff Mariconda

CS 511

**Assignment 4 Report**

* Test File Name:
  + input.xml
* Number of pages parsed:
  + 1000
* Number of Consumer threads:
  + 7
* Number of Processors on System:
  + 8
* Actual command run:
  + java -classpath bin/ TokenCount 100 ../input.xml
* Test Results:
  + Part 1 (1 Producer, 1 Consumer):
    - Ran 5 times with the following times (in order):
      * 1575ms, 1525ms, 1583ms, 1701ms, 1549ms
    - Average runtime:
      * **1586.6ms**
  + Part 2 (1 Producer, 7 Consumers with Lock):
    - Ran 5 times with the following times (in order):
      * 1580ms, 1590ms, 1605ms, 1669ms, 1619ms
    - Average runtime:
      * **1612.6ms**
  + Part 3 (1 Producer, 7 Consumers with ConcurrentHashMap):
    - Ran 5 times with the following times (in order):
      * 1300ms, 1301ms, 1366ms, 1247ms, 1357ms
    - Average runtime:
      * **1314.2ms**
  + Part 4 (1 Producer, 7 Consumers with private HashMaps and merge):
    - Ran 5 times with the following times (in order):
      * 1263ms, 1157ms, 1272ms, 990ms, 1015ms
    - Average runtime:
      * **1139.4ms**
* Analysis:

These average runtimes fit the expected results for the following reasons. If we use the first part with only one producer and one consumer thread as the base model, it then makes sense that the second part would be slightly slower on average; even though there are 7 consumer threads running in the second part, which seems like it should be faster, only one thread can modify the frequency hashmap at a time, so it might as well just be one producer and one consumer thread running concurrently. On top of this, having to handle the lock and sleeping threads adds some slight extra processing time, which is probably why it's slightly slower. We can also see that in part 3, the runtime is (relatively) significantly less than part 1. This fits expectations because, rather than locking out the entire hashmap like we did in part 2, ConcurrentHashMap only locks individual buckets, which allows multiple consumer threads to modify the hashmap at the same time (as long as they are not modifying the same key-value pair). Finally, we get our most efficient runtime in part 4, in which we had each consumer thread hold its own private frequency hashmap and then merge it with the shared one once it had finished counting tokens. It makes sense that this yields a more efficient runtime than even that of part 3 because, rather than consumer threads trying to modify the shared hashmap at completely random times, they all merge all of their counts at the end of their runs. This lowers the chance that a thread will get locked out of a bucket because it forces threads to access the shared hashmap in a more orderly way, rather than being completely random.