Group 1

Assignment 3

Week 5

5.1.1) *First compile and run the thread timing code as is, using Mark6, to get a feeling for the variation and robustness of the results. Do not hand in the results but discuss any strangenesses, such as large variation in the time measurements for each case.*

normal

5.1.2) . *Now change all the measurements to use Mark7, which reports only the final result. Record the results in a text file along with appropriate system identification.*

Include the results in your hand-in, and reflect and comment on them: Are they plausible? Any surprises? Mention any cases where they deviate significantly from those shown in the lecture.

 # OS: Windows 10; 10.0; amd64

 # JVM: Oracle Corporation; 1.8.0_212

 # CPU: Intel64 Family 6 Model 186 Stepping 2, GenuineIntel; 16 "cores"

 # Date: 2024-01-17T05:14:52+0800

hashCode()	1.8 ns	0.04	268435456
Point creation	26.8 ns	0.87	16777216
Thread's work	4043.1 ns	49.41	65536
Thread create	513.3 ns	15.36	524288
Thread create start	43879.4 ns	3013.51	8192
Thread create start join	76352.6 ns	6836.85	4096

ai value = 1556420000

Uncontended lock	2.5 ns	0.04	134217728

No, there are all similar values with the examples from the lecture, where other results are a little faster because of a faster cpu (maybe).

Group 1

5.2.1) *Measure the performance of the prime counting example on your own hardware, as a function of the number of threads used to determine whether a given number is a prime. Record system information as well as the measurement results for 1. . . 32 threads in a text file. If the measurements take excessively long time on your computer, you may measure just for 1. . . 16 threads instead.*

 # OS: Windows 10; 10.0; amd64

 # JVM: Oracle Corporation; 1.8.0_212

 # CPU: Intel64 Family 6 Model 126 Stepping 5, GenuineIntel; 8 "cores"

 # Date: 2023-09-25T17:31:57+0200

countSequential		151300235.0 ns 18546869.03	2
countParallelN	1	155179675.0 ns 9056611.26	2
countParallelN	2	151893610.0 ns 61352437.66	2
countParallelN	3	173473710.0 ns 5697428.94	2
countParallelN	4	141873355.0 ns 7371791.70	2
countParallelN	5	114065145.0 ns 7792364.84	4
countParallelN	6	54969327.5 ns 7111219.10	8
countParallelN	7	47906332.5 ns 6207662.45	4
countParallelN	8	40948645.0 ns 2633282.66	8
countParallelN	9	40764712.5 ns 2887061.25	8
countParallelN	10	55624790.0 ns 19989114.08	8
countParallelN	11	78986372.5 ns 7093442.03	4
countParallelN	12	78048635.0 ns 5155190.16	4
countParallelN	13	43484542.5 ns 2979389.82	8
countParallelN	14	40102070.0 ns 1678135.40	8
countParallelN	15	56882695.0 ns 16190969.71	8
countParallelN	16	78285762.5 ns 4079814.51	4

5.2.2) *Reflect and comment on the results; are they plausible? Is there any reasonable relation between the number of threads that gave best performance, and the number of cores in the computer you ran the benchmarks on? Any surprises?*

Group 1

It gives best performance when the number of threads is equal to the number of cores in the computer (which is 8 in the experiment). When the thread number is more than the number that the CPU can handle at one time, the performance will drop (switch cost).

5.2.3（**2024.1.17**）

countParallelN 1 94155625.0 ns 1864871.75 4

countParallelNLocal 1 92683630.0 ns 749673.38 4

countParallelN 2 58888157.5 ns 794511.90 8

countParallelNLocal 2 58873427.5 ns 1453505.80 8

countParallelN 3 42021787.5 ns 2372702.91 8

countParallelNLocal 3 39982641.3 ns 424667.84 8

countParallelN 4 31917905.0 ns 431371.72 8

countParallelNLocal 4 31451057.5 ns 461349.69 8

countParallelN 5 29282181.3 ns 867602.42 16

countParallelNLocal 5 28098125.0 ns 961262.63 16

countParallelN 6 26357924.4 ns 421126.40 16

countParallelNLocal 6 27153431.3 ns 1541017.63 16

countParallelN 7 90249720.0 ns 33811055.77 8

countParallelNLocal 7 102704340.0 ns 4030956.35 4

countParallelN 8 94454947.5 ns 7401538.02 4

countParallelNLocal 8 92285082.5 ns 4407984.34 4

countParallelN 9 81447867.5 ns 10772597.32 4

countParallelNLocal 9 79624835.0 ns 3873026.85 4

countParallelN 10 85632445.0 ns 5722812.22 4

countParallelNLocal 10 72302960.0 ns 3150230.33 4

countParallelN 11 81467575.0 ns 5634758.80 4

countParallelNLocal 11 70519000.0 ns 7259508.88 4

countParallelN 12 79011900.0 ns 8084430.61 4

Group 1

countParallelNLocal 12 17702998.8 ns 253322.75 16

countParallelN 13 22275008.8 ns 406500.06 16

countParallelNLocal 13 19526541.3 ns 1682645.58 16

countParallelN 14 40702790.0 ns 28523462.89 8

countParallelNLocal 14 64981155.0 ns 3628921.68 4

countParallelN 15 81770537.5 ns 5414158.70 4

countParallelNLocal 15 65488505.0 ns 3247788.80 4

countParallelN 16 80290742.5 ns 4410379.20 4

countParallelNLocal 16 58762825.0 ns 2703474.20 4

It did faster (as more threads, more competition)

5.3.1) *Use Mark7 (from Bendchmark.java) to compare the performance of incrementing a volatile int and a normal int. Include the results in your hand-in and comment on them: Are they plausible? Any surprises?*

OS: Windows 10; 10.0; amd64

JVM: Oracle Corporation; 1.8.0_212

CPU: Intel64 Family 6 Model 126 Stepping 5, GenuineIntel; 8 "cores"

Date: 2023-09-25T16:29:06+0200

Mark 7 measurements

TestVolatile 7.0 ns 0.73 67108864

TestNonVolatile 3.2 ns 0.43 134217728

No surprises, TestNonVolatile is about two times faster than TestVolatile, which is in expect.

additional overhead: When reading a volatile variable, the thread needs to fetch the latest value from the main memory instead of using a local cache. This can result in performance overhead for read operations. When writing to a volatile variable, the new value must be flushed to the main memory to ensure that other threads can see the updated value. This can lead to performance overhead for write operations.

Group 1

5.4.1) *Extend LongCounter with these two methods in such a way that the counter can still be shared safely by several threads.*

See PrimeCounter.java

5.4.2) *How many occurencies of "ipsum" is there in long-text-file.txt. Record the number in your solution.*

Occurences of ipsum :1430

5.4.3) *Use Mark7 to benchmark the search function. Record the result in your solution.*

TestTimeSearch	13209263.4 ns	771208.37	32

5.4.4) *Extend the code in TestTimeSearch with a new method. Fill in the body of countParallelN in such a way that the method uses N threads to search the lineArray. Provide a few test results that make i plausible that your code works correctly.*

Test : See TestTimeSearchCorrectness.java

5.4.5) *Use Mark7 to benchmark countParallelN. Record the result in your solution and provide a small discussion of the timing results.*

TestTimeSearchParallel	3940885.2 ns	330850.37	64

From the result of the benchmark, we could see that we gain a big performance improvement by using a function using multiple threads, which is about three times faster than sequential searching..

Group 1

Assignment 3

Week 6

6.1.1) *Use Mark7 (from Benchmark.java in the benchmarking package) to measure the execution time and verify that the time it takes to run the program is proportional to the transaction time.*

Thread.sleep(50):	315949904,9 ns	2350268,59	2
Thread.sleep(100):	545715475,0 ns	2866201,25	2
Thread.sleep(500):	2549560605,0 ns	5859344,44	2

When we multiply the execution time of the transaction method the increase in running time of the test is proportional with the multiplier.

6.1.2) *Now consider the version in ThreadsAccountExperimentsMany.java (in the directory exercise61). Explain why the calculation of min and max are necessary? Eg. what could happen if the code was written like this:*

min() and max() are necessary so we consistently require the account locks in a particular order, so we don't get an interleaving where a transfer gets the first lock but is unable to get the second lock because is being held by another transfer that is waiting to get the first lock, creating a deadlock. By getting the lock for the min() account first before requiring the lock for the max() account, two transfers will never wait to require the lock held by the other.

6.1.3) *Change the program in ThreadsAccountExperimentsMany.java to use a the executor framework instead of raw threads. Make it use a ForkJoin thread pool. For now do not worry about terminating the main thread, but insert a print statement in the doTransaction method, so you can see that all executors are active.*

See ThreadsAccountExperimentsMany.java

6.1.4) *Ensure that the executor shuts down after all tasks has been executed.*

See ThreadsAccountExperimentsMany.java

6.2.1) *Report and comment on the results you get from running TestCountPrimesThreads.java*

countSequential	11815044.4 ns

Group 1

countParallelN 1	12088541.6 ns
countParallelNLocal 1	12129303.4 ns
countParallelN 2	7765589.1 ns
countParallelNLocal 2	7754480.2 ns
countParallelN 3	5586763.9 ns
countParallelNLocal 3	5501449.1 ns
countParallelN 4	4264456.9 ns
countParallelNLocal 4	4245277.7 ns
countParallelN 5	3671260.2 ns
countParallelNLocal 5	3644216.2 ns
countParallelN 6	3172800.1 ns
countParallelNLocal 6	3161662.5 ns
countParallelN 7	2864374.5 ns
countParallelNLocal 7	2838376.2 ns
countParallelN 8	2606973.6 ns
countParallelNLocal 8	2593272.3 ns
countParallelN 9	2467011.5 ns
countParallelNLocal 9	2440349.4 ns
countParallelN 10	2322588.4 ns
countParallelNLocal 10	2319134.8 ns
countParallelN 11	2266415.7 ns
countParallelNLocal 11	2179305.2 ns
countParallelN 12	2154408.3 ns
countParallelNLocal 12	2083492.9 ns
countParallelN 13	2072336.2 ns
countParallelNLocal 13	2046452.0 ns
countParallelN 14	2054689.6 ns
countParallelNLocal 14	2022442.4 ns
countParallelN 15	2040972.6 ns
countParallelNLocal 15	2024237.2 ns
countParallelN 16	2020090.5 ns
countParallelNLocal 16	2004681.3 ns
…	
countParallelN 24	2331643.0 ns
countParallelNLocal 24	2322529.5 ns
…	
countParallelN 32	2778359.7 ns
countParallelNLocal 32	2772727.0 ns

The running time initially decreases with the number of Threads used, until around 13 threads where the benefits of multiple threads are maximized. However, the running time afterwards increases as we add more Threads. This might be because of the cost of creating and starting more threads increases the running time without increasing the performance, or in other words to small tasks for too many tasks. It could also be because of the cost of thread switching when the number of threads exceeds the available CPU cores.

Group 1

6.2.2) *Rewrite TestCountPrimesthreads.java using Futures for the tasks of each of the threads in part 1. Run your solutions and report results. How do they compare with the results from the version using threads?*

```
countSequential             3773243.9 ns   50870.25       128
countParallelN  1           4080372.7 ns   63530.48        64
countParallelNLocal  1      4115000.0 ns  100894.13        64
countParallelNFuture  1     4087351.4 ns   96381.79        64
countParallelN  2           2554377.7 ns   34927.21       128
countParallelNLocal  2      2489968.8 ns   33432.66       128
countParallelNFuture  2     2601642.0 ns   36709.21       128
countParallelN  3           1841753.7 ns   27554.13       256
countParallelNLocal  3      1819081.5 ns   42026.99       256
countParallelNFuture  3     2771740.3 ns   39835.21       128
countParallelN  4           1570366.1 ns  145832.47       256
countParallelNLocal  4      2554629.6 ns 2112700.76       256
countParallelNFuture  4     7016475.3 ns 1122737.93        64
..................................
countParallelN 17           6652607.3 ns  629465.63        64
countParallelNLocal 17      5846794.2 ns  499136.60        64
countParallelNFuture 17     4955208.3 ns  310872.02        64
countParallelN 18           6175751.3 ns  909340.72        64
countParallelNLocal 18      6668677.0 ns  338690.53        64
countParallelNFuture 18     5308196.7 ns  234497.82        64
countParallelN 19           6301578.3 ns  863976.25        64
countParallelNLocal 19      6568622.8 ns  786636.28        64
countParallelNFuture 19     5142262.3 ns  266904.52        64
countParallelN 20           5766233.4 ns  945287.15        64
countParallelNLocal 20      5282878.1 ns  404690.01        64
countParallelNFuture 20     5010844.5 ns  296308.02        64
countParallelN 21           6768498.1 ns  826686.94        64
countParallelNLocal 21      1433528.2 ns   24257.02       256
countParallelNFuture 21     1144980.2 ns   21463.30       256
countParallelN 22           1489715.0 ns   26920.79       256
countParallelNLocal 22      1499974.1 ns   17286.77       256
countParallelNFuture 22     1174040.4 ns   23171.86       256
countParallelN 23           1599630.7 ns   36974.82       256
countParallelNLocal 23      2105147.1 ns  876709.59       256
countParallelNFuture 23     4962677.7 ns  180546.44        64
countParallelN 24           7450151.6 ns  474214.56        64
countParallelNLocal 24      5028808.3 ns  686910.56        64
countParallelNFuture 24     5334477.5 ns  220633.18        64
countParallelN 25           8294875.3 ns  596579.67        32
countParallelNLocal 25      7919089.8 ns  435709.89        64
countParallelNFuture 25     5114154.2 ns  442231.74        64
countParallelN 26           8737798.1 ns  675896.53        32
countParallelNLocal 26      8448539.1 ns  367583.52        32
countParallelNFuture 26     5047753.4 ns  675333.52        64
countParallelN 27           8247068.6 ns  837568.96        64
countParallelNLocal 27      8836368.4 ns  370999.15        32
countParallelNFuture 27     4623243.6 ns  818401.67        64
countParallelN 28           1709594.0 ns   96682.96       256
```

Group 1

countParallelNLocal 28	1699194.4 ns	42793.00	256
countParallelNFuture 28	1172929.4 ns	11153.62	256
countParallelN 29	1726824.1 ns	34907.05	256
countParallelNLocal 29	1779224.3 ns	46050.02	256
countParallelNFuture 29	1187661.7 ns	22107.85	256
countParallelN 30	1864863.4 ns	104878.18	256
countParallelNLocal 30	1864610.5 ns	50252.83	256
countParallelNFuture 30	1277469.3 ns	39936.13	256
countParallelN 31	1817270.2 ns	20475.48	256
countParallelNLocal 31	1835273.2 ns	72284.39	128
countParallelNFuture 31	1254487.6 ns	29581.63	256
countParallelN 32	2015411.8 ns	102232.44	128
countParallelNLocal 32	1982057.3 ns	74918.17	128
countParallelNFuture 32	1339935.2 ns	57477.64	256

As the number of threads increases, we can observe that using a thread pool provides better performance. This is due to the significant overhead associated with the creation and destruction of threads, which thread pools can reuse threads.

6.3.1). *Make a thread-safe implementation, class Histogram2 implementing the interface Histogram. Use suitable modifiers (final and synchronized) in a copy of the Histogram1 class. This class must use at most one lock to ensure mutual exclusion.*
Explain what fields and methods need modifiers and why. Does the getSpan method need to be synchronized?

We define counts as a **final** array and use **synchronized** keywords on increment() and getCount(). The former ensures safe publication of the counts[] array during initialization, while the latter ensures mutual exclusion when accessing the value of a bin.

There is no need to use **synchronized** keywords on getSpan() as the length of the array is an immutable Integer after initialization. and " question says: you may assume that the dump method given there is called only when no other thread manipulates the histogram and therefore does not require locking, and that the span is fixed (immutable) for any given Histogram object. "

6.3.2) *Now create a new class, Histogram3 (implementing the Histogram interface) that uses lock striping. You can start with a copy of Histogram2. Then, the constructor of Histogram3 must take an additional parameter nrLocks which indicates the number of locks that the histogram uses. You will have to associate a lock to each bin.*

See Histogram3.java.

6.3.3) *Now consider again counting the number of prime factors in a number p. Use the Histogram2 class to write a program with multiple threads that counts how many numbers in the range 0. . . 4 999 999 have 0 prime factors, how many have 1 prime factor, how many have 2 prime factors, and so on. You may draw inspiration from the TestCountPrimesThreads.java.*

Group 1

See HistogramPrimesThreads.java. The result is as expected.

6.3.4) *Finally, evaluate the effect of lock striping on the performance of question 6.3. Create a new class where you use Mark7 to measure the performance of Histogram3 with increasing number of locks to compute the number of prime factors in 0. . . 4 999 999. Report your results and comment on them. Is there a increase or not? Why?*

```
test_histogram3 10 with  1 locks    1909223070.0 ns 1473822054.97         2
test_histogram3 10 with  2 locks    1068259645.0 ns 837775365.74          2
test_histogram3 10 with  3 locks     593596850.0 ns 25299286.16           2
test_histogram3 10 with  4 locks     529367480.0 ns 26400001.43           2
test_histogram3 10 with  5 locks     526213450.0 ns 30539477.78           2
test_histogram3 10 with  6 locks     522030245.0 ns 18644255.88           2
test_histogram3 10 with  7 locks     505420595.0 ns 8048692.75           2
test_histogram3 10 with  8 locks     498936060.0 ns 13198599.53           2
test_histogram3 10 with  9 locks     519260795.0 ns 17450286.18           2
test_histogram3 10 with 10 locks     511221235.0 ns 11457286.43           2
```

Using multiple locks improves performance compared to using a single lock. However, as the number of locks increases, performance gains do not exhibit the same increasing trend. By using lock striping, different locks are used when accessing different bins, reducing lock contention and thus improving performance. When adding locks up to a certain number (8), the program's bottleneck may no longer be lock contention. Factors such as the number of CPU cores, the number of threads, and separation loss could all be reasons limiting further performance improvement.