ASSIGNMENT 4 SOLUTIONS

Exercise 4.1

Mark 1

The results can be found in the file Mark1.txt.

First of all, to compare against Peter Sestoft's results, we increment the counter in the loop inside the method of mark 1 to 1 million, just as Peter has done.

Running the program several times, we get results spanning from 4.2ns to 4.8ns. This is slightly faster than Peter's results. The reason for this is not surprising. Many factors could explain this performance enhancement such as different versions of windows (we are running in windows 10), faster CPU or different JVM version. All of which could explain the slightly faster running time. However, our variation is the same as Peter (0.5), which is better to compare against.

Mark 2

The results can be found in the file Mark2.txt.

Running the mark 2 method several times, we get results spanning from 25.6ns to 25.8ns. Again, our results are faster than Peter's, just as we saw in mark 1. We see that the running times are much higher now, due to the dummy variable introduced. Here we tricked the compiler to not remove the for loop as dead code, as we now save the result for each iteration in a dummy variable, resulting in a much more stable running time.

Mark 3

The results can be found in the file Mark3.txt.

In mark 3 we simply run the same code as in mark 2, but with an added outer loop to automate the times the method is run.

We get measurements that are consistent with those in mark 2, however with a slightly bigger span. This can be due to the fact that we ran the program 10 times instead of only 4. Again, as mentioned before, the time is slightly faster than Peter's, which could be due to better/newer hardware, OS or compiler.

Mark 4

The results can be found in the file Mark4.txt.

In mark 4 the results are again as expected – consistent with mark 3 and slightly faster than Peter's results, as discussed above.

In mark 4 we calculate the average time of running the measurement 10 times, where each executes the multiplication method 100 million times. Out of the 10 times we run the method we

find an empirical deviation of 0.963, which is low deviation, meaning that our data spread is low. We expect a low distribution of data because we have a high count, meaning that we stabilize the running time. The time is stabilized as if some iterations for some reason have a high execution time, they will not be emphasized as much as if we had a smaller count.

Mark 5

The results can be found in the file Mark5.txt.

In Mark 5 we are automating the number of times we want to execute the method by putting an outer loop that will double the count until we have reached a running time of 0.25ns for the loop iteration the multiply() method.

The results are consistent with mark 4, and as expected slightly faster than Peter's results.

However, we do not get a sudden increase as Peter did, which may indicate that our garbage collector didn't 'kick in' during execution.

In the result we see that the first 4 iterations are unreliable, as expected, as the iteration count is very low. Here, we are probably not really measuring the time it takes for the execution to run, but more likely we are measuring the compilation time of the multiply() method or the call to the timer's check() method.

Exercise 4.2

Green

ST = Total running time of the count loop

SST = The total running time of the count loop squared

Mean = the average of the total running time

Sdev = the standard deviation

The standard deviation is telling us how the distribution of the numbers is. It tells us how close one iteration, of the for loop that iterates the count, is to the mean (the average). A high standard deviation indicates that the numbers are very spread, whereas a low standard deviation tells us that the numbers are close to each other.

In this example we use the formula:

Which is a formula for calculating the standard deviation.

The completed primitive code can be found in file MeanVar.java.

The result of the program running the first sample can be found in MeanVar1.txt

The result of the program running the second sample can be found in MeanVar2.txt

Sample 1:

In file MeanVar1.txt we see that the program running the first sample has a mean of 32.5ns and a standard deviation of 6.228 ns. That means that if we follow the standard deviation, the number can be 3 times away from our mean, either in negative or positive direction, to be regarded as an outlier.

Therefore:

3 * 5.908 = 17.724ns.

32.5 + 17.724 = 50.224ns.

32.5 – 17.724 = 14.776ns

Hence, every number between 14.776ns and 50.224ns would not be an outlier. Therefore, 25 is not an outlier.

Sample 2:

In file MeanVar2.txt we see that the program running the first sample has a mean of 31,1ns and a standard deviation of 26,674ns. That means that if we follow the standard deviation, the number can be 3 times away from our mean, either in negative or positive direction, to be regarded as an outlier.

Therefore:

3 * 26.674 = 80.022ns.

31.1 + 80.022 = 111,122ns.

31.1 – 80.022 = -48.922ns.

Hence, every number between 111,122 and -48.922 would not be an outlier. Therefore, 25 is not an outlier.

Exercise 4.3

4.3.1

Hashcode

As expected the time goes down as the count increases, due to the rise of stability when iterating more times. However, we see some disturbance when the mean rises from 28.0 when the count is 128 and to 178.9 when count is 512. This is probably due to some disturbance caused by either the JIT compiler, the garbage collector or some other factors inside the local system. However, as we see that the standard deviation is also of, we do not trust this result and regard it as an outlier. Except for that we see the running time converges.

Point creation

As expected, we see that the time is much higher than in the hashcode. This is expected because we now create a point object for each iteration. Again, we see the result stabilize as the count increases, and have some running times that stick out, what we expect to be because of external disturbance.

Thread works

Here we really see the running time is high due to the atomic integer. This is also expected as the atomic integer synchronizes every time an increment is made, cause a thread to block and unblock. That is time consuming as can be seen here.

The interesting thing is that the standard deviation actually is very high for all the runs, which indicates that the results are very different. One reason could be that the count never reaches above 65536, because the operations on an AtomicInteger are time consuming, and therefore reaching the limit of 0.25ns very fast.

Another point would be that due to working with an atomic integer, causes a thread to block making it wait for the lock to be released before it can process. This waiting time could may vary depending on the other resources using the thread and would explain the high variation of the standard deviation.

Thread create

Now we see a time that is lower than when incrementing the AtomicInteger. This is due to that a thread is created, but never run. Only the hashcode of the thread is returned. Therefore, we are actually measuring the creation time of a thread object more than the actual thread being run.

Thread create start

Here again we see the same issue as in thread works, but even worse. This is because many threads are created and run concurrently, making them wait for each other whenever incrementing the AtomicIteger – we never wait for them using the join(). This waiting process makes the mean and standard deviation very high as the running times are very different. Also note that the mean is affected by the long execution time of creating a thread.

Thread create start join

Same as in thread create start however, we see that the mean is remarkably lower than in the thread create start. This is due to the fact that this time we actually wait for the thread to finish,

using the join(), before actually moving on. This means that the threads don't interrupt each other. However, the mean is still quite high, which can be explained by the fact that we use an AtomInteger, as described above in thread works.

Uncontended lock

Here it is very clear that the only thing we actually are measuring is the time it takes to get and release a lock. No threads are blocked as no other resources are using the same lock. Hence, the standard deviation is very low when the count increases and the result stabilizes.

4.3.2

The result of the program running mark 7 can be found in TestTimeThreadsMark7.txt

The results are as expected – no surprises. The standard deviation rises when using concurrent programs, because waiting time on locks varies a lot. Using thread safe classes such as AtomicInteger is also quite complicated to measure performance on, as locking, waiting and unlocking is dependent on other threads.

In summary one could say that measuring the performance of parallel programs is quite difficult.

Exercise 4.4

2. The results obtained from the experiment were plausible. The average execution time of the function decreases as the number of threads used to execute the function increases.

The greater the number of cores in the computer, the higher the number of threads that can be created to execute on the function.

3. The performance of LongCounter was better than that of AtomicLong.

In the results obtained from the use of the AtomicLong class, the average execution time of the function decreases as the number of threads increase from 0 to 9, however as the number of threads increase from 10 to 32, the average execution of the function tends to increase.

Taking into consideration the results above, one should **refrain** from using built-in classes and methods when they exist.

Exercise 4.5

Results:

```
# OS:   Mac OS X; 10.14.6; x86_64
# JVM:  Oracle Corporation; 11.0.8
```

```
# CPU:  null; 16 "cores"
# Date: 2020-09-23T21:56:12+0200
Uncontended lock                379.8 ns     736.97         2
Uncontended lock                144.8 ns      57.02         4
Uncontended lock                127.0 ns      36.73         8
Uncontended lock                290.8 ns     568.44        16
Uncontended lock                 28.5 ns       3.61        32
Uncontended lock                 28.3 ns       4.69        64
Uncontended lock                 26.7 ns       3.09       128
Uncontended lock                 25.3 ns       1.34       256
Uncontended lock                 52.2 ns      32.88       512
Uncontended lock                 44.3 ns      17.02      1024
Uncontended lock                 24.1 ns       0.34      2048
Uncontended lock                 25.2 ns       3.38      4096
Uncontended lock                 10.9 ns       6.26      8192
Uncontended lock                  8.7 ns       0.69     16384
Uncontended lock                  9.1 ns       2.02     32768
Uncontended lock                  7.7 ns       2.67     65536
Uncontended lock                  2.3 ns       0.84    131072
Uncontended lock                  2.0 ns       0.02    262144
Uncontended lock                  2.0 ns       0.07    524288
Uncontended lock                  2.0 ns       0.06   1048576
Uncontended lock                  2.0 ns       0.03   2097152
Uncontended lock                  2.0 ns       0.06   4194304
Uncontended lock                  2.0 ns       0.06   8388608
Uncontended lock                  2.0 ns       0.03  16777216
Uncontended lock                  2.0 ns       0.07  33554432
Uncontended lock                  2.0 ns       0.07  67108864
Uncontended lock                  2.0 ns       0.03 134217728
```

The results look plausible. This is because with an **uncontended** lock on the hardware, the greater the number of times a function is executed, the more stable the average execution time of the function is going to be.

Exercise 4.6

Results:

```
***VOLATILE EXPERIMENTS***
N=10000000
```

Result	27106885.1 ns	2014925.31	2
Result	13445738.1 ns	413321.64	4
Result	6663773.0 ns	54883.71	8
Result	3312516.8 ns	45232.97	16
Result	1667244.4 ns	23147.40	32
Result	829016.6 ns	14265.98	64
Result	415163.2 ns	4824.73	128
Result	226804.5 ns	17569.81	256
Result	117558.6 ns	4622.52	512
Result	57735.5 ns	4568.14	1024
Result	28622.4 ns	1471.96	2048
Result	14058.4 ns	391.38	4096
Result	6898.5 ns	308.28	8192
Result	3547.7 ns	125.06	16384
Result	1804.8 ns	88.61	32768
Result	867.7 ns	14.70	65536
Result	446.9 ns	36.77	131072
Result	216.5 ns	10.56	262144
Result	112.2 ns	10.01	524288
Result	57.4 ns	3.49	1048576
Result	27.4 ns	1.09	2097152
Result	13.5 ns	0.62	4194304
Result	6.9 ns	0.32	8388608
Result	3.3 ns	0.10	16777216
Result	1.6 ns	0.01	33554432
Result	0.8 ns	0.01	67108864
Result	0.4 ns	0.00	134217728
Result	0.2 ns	0.00	268435456
Result	0.1 ns	0.00	536870912
Result	0.1 ns	0.00	1073741824
N=100000000			
Result	282196420.0 ns	13758969.35	2
N=214748364			
Result	13.5 ns	1.94	2
Result	6.9 ns	0.51	4
Result	3.9 ns	1.71	8
Result	2.0 ns	0.91	16
Result	0.9 ns	0.30	32
Result	0.4 ns	0.06	64
Result	0.3 ns	0.13	128
Result	0.1 ns	0.01	256
Result	0.1 ns	0.01	512
Result	0.0 ns	0.01	1024
Result	0.0 ns	0.00	2048
Result	0.0 ns	0.00	4096
Result	0.0 ns	0.00	8192
Result	0.0 ns	0.00	16384
Result	0.0 ns	0.00	32768
Result	0.0 ns	0.00	65536
Result	0.0 ns	0.00	131072
Result	0.0 ns	0.00	262144
Result	0.0 ns	0.00	524288
Result	0.0 ns	0.00	1048576
Result	0.0 ns	0.00	2097152
Result	0.0 ns	0.00	4194304
Result	0.0 ns	0.00	8388608
Result	0.0 ns	0.00	16777216

Result	0.0 ns	0.00	33554432
Result	0.0 ns	0.00	67108864
Result	0.0 ns	0.00	134217728
Result	0.0 ns	0.00	268435456
Result	0.0 ns	0.00	536870912
Result	0.0 ns	0.00	1073741824

NON-VOLATILE EXPERIMENTS

N=10000000

Result	14.9 ns	4.45	2
Result	7.6 ns	2.97	4
Result	3.9 ns	1.67	8
Result	2.0 ns	0.86	16
Result	1.0 ns	0.43	32
Result	0.5 ns	0.16	64
Result	0.3 ns	0.14	128
Result	0.1 ns	0.05	256
Result	0.1 ns	0.01	512
Result	0.0 ns	0.01	1024
Result	0.0 ns	0.01	2048
Result	0.0 ns	0.00	4096
Result	0.0 ns	0.00	8192
Result	0.0 ns	0.00	16384
Result	0.0 ns	0.00	32768
Result	0.0 ns	0.00	65536
Result	0.0 ns	0.00	131072
Result	0.0 ns	0.00	262144
Result	0.0 ns	0.00	524288
Result	0.0 ns	0.00	1048576
Result	0.0 ns	0.00	2097152
Result	0.0 ns	0.00	4194304
Result	0.0 ns	0.00	8388608
Result	0.0 ns	0.00	16777216
Result	0.0 ns	0.00	33554432
Result	0.0 ns	0.00	67108864
Result	0.0 ns	0.00	134217728
Result	0.0 ns	0.00	268435456
Result	0.0 ns	0.00	536870912
Result	0.0 ns	0.00	1073741824

N=100000000

Result	14.1 ns	1.94	2
Result	8.0 ns	3.96	4
Result	3.9 ns	1.75	8
Result	1.7 ns	0.21	16
Result	0.9 ns	0.11	32
Result	0.4 ns	0.01	64
Result	0.2 ns	0.08	128
Result	0.1 ns	0.03	256
Result	0.1 ns	0.02	512
Result	0.0 ns	0.01	1024
Result	0.0 ns	0.01	2048
Result	0.0 ns	0.00	4096
Result	0.0 ns	0.00	8192
Result	0.0 ns	0.00	16384
Result	0.0 ns	0.00	32768
Result	0.0 ns	0.00	65536

Result	0.0 ns	0.00	131072
Result	0.0 ns	0.00	262144
Result	0.0 ns	0.00	524288
Result	0.0 ns	0.00	1048576
Result	0.0 ns	0.00	2097152
Result	0.0 ns	0.00	4194304
Result	0.0 ns	0.00	8388608
Result	0.0 ns	0.00	16777216
Result	0.0 ns	0.00	33554432
Result	0.0 ns	0.00	67108864
Result	0.0 ns	0.00	134217728
Result	0.0 ns	0.00	268435456
Result	0.0 ns	0.00	536870912
Result	0.0 ns	0.00	1073741824
N=214748364			
Result	21.1 ns	15.91	2
Result	8.2 ns	3.62	4
Result	4.1 ns	1.81	8
Result	2.5 ns	1.57	16
Result	0.9 ns	0.10	32
Result	0.5 ns	0.26	64
Result	0.2 ns	0.02	128
Result	0.1 ns	0.01	256
Result	0.1 ns	0.03	512
Result	0.0 ns	0.00	1024
Result	0.0 ns	0.01	2048
Result	0.0 ns	0.00	4096
Result	0.0 ns	0.00	8192
Result	0.0 ns	0.00	16384
Result	0.0 ns	0.00	32768
Result	0.0 ns	0.00	65536
Result	0.0 ns	0.00	131072
Result	0.0 ns	0.00	262144
Result	0.0 ns	0.00	524288
Result	0.0 ns	0.00	1048576
Result	0.0 ns	0.00	2097152
Result	0.0 ns	0.00	4194304
Result	0.0 ns	0.00	8388608
Result	0.0 ns	0.00	16777216
Result	0.0 ns	0.00	33554432
Result	0.0 ns	0.00	67108864
Result	0.0 ns	0.00	134217728
Result	0.0 ns	0.00	268435456
Result	0.0 ns	0.00	536870912
Result	0.0 ns	0.00	1073741824

When using volatile, you see that we are much more prone to unstable results with a low count. This is because when the count is low, you are actually measuring the volatile operations rather than the running time of an array. And when the count increases, you converge towards a much lower runtime.

On the other hand, when using a non volatile, the running time is very fast - The operations are not using any time - making the results less unstable even with small count.