Assignments for week 3

Anders Degn Lapiki, Jacob Kjærulff Furberg, Jonas Ishøj Nielsen

03/09/2020

Contents

1 4.1 Microbenchmarks

1.1 4.1 - red

```
System


# OS:   Windows 10; 10.0; amd64\\
# JVM:  Oracle Corporation; 14.0.2\\
# CPU:  Intel64 Family 6 Model 94 Stepping 3, GenuineIntel; 8 "cores"\\
# Date: 2020-09-17T10:27:20+0200\\
```

```
   Mark1

0,009 s     0,5ns
```

Very low, as the multiply is treated as dead code by the JVM.

```
   Mark2

23,6 ns
```

Makes sense, now that the code is used.

```
   Mark3

  23,7 ns
  23,5 ns
  23,5 ns
  25,0 ns
  24,3 ns
  23,5 ns
  23,5 ns
  23,5 ns
  23,5 ns
  23,5 ns
```

Makes sense, now that the code is used and test is run x times to show more reasonable results. The individual result are also close to Mark2 as expected.

```
   Mark4

 23,7 ns +/-  0,626
```

The mean and the standard deviation seems to fit the result when run 10 times.

```
   Mark5
```

```
 425,0 ns +/-   817,60          2
 135,0 ns +/-    67,91          4
 115,0 ns +/-    38,55          8
 103,1 ns +/-    20,04         16
 110,3 ns +/-    43,83         32
  91,9 ns +/-     8,46         64
  48,8 ns +/-    52,13        128
  32,6 ns +/-     2,97        256
  31,3 ns +/-     1,35        512
  30,9 ns +/-     0,52       1024
  30,7 ns +/-     0,26       2048
  30,7 ns +/-     0,18       4096
  33,1 ns +/-     4,33       8192
  30,5 ns +/-     0,36      16384
  33,6 ns +/-    20,52      32768
  29,8 ns +/-     2,65      65536
  26,2 ns +/-     0,51     131072
  23,5 ns +/-     0,55     262144
  23,5 ns +/-     0,15     524288
  23,5 ns +/-     0,15    1048576
  23,5 ns +/-     0,13    2097152
  23,5 ns +/-     0,18    4194304
  23,5 ns +/-     0,06    8388608
  23,5 ns +/-     0,11   16777216
```

Makes sense as the mean converges towards correct answer, standard deviation becomes smaller and garbage collection cause outliers underway.

```
   Mark6

multiply            845,0 ns    1882,00          2
multiply            627,5 ns    1071,89          4
multiply            268,8 ns      75,29          8
multiply            268,8 ns     190,92         16
multiply             88,8 ns      79,74         32
multiply             63,8 ns       9,10         64
multiply            113,1 ns     102,51        128
multiply             97,3 ns      49,50        256
multiply             67,1 ns      30,59        512
multiply             34,6 ns       0,76       1024
multiply             36,3 ns       4,14       2048
multiply             35,3 ns       2,23       4096
multiply             32,0 ns       3,98       8192
multiply             29,9 ns       0,31      16384
multiply             29,2 ns       2,28      32768
multiply             30,8 ns       1,85      65536
multiply             25,3 ns       1,50     131072
```

```
multiply             24,9 ns       0,47     262144
multiply             23,5 ns       0,14     524288
multiply             23,5 ns       0,18    1048576
multiply             23,5 ns       0,12    2097152
multiply             23,5 ns       0,08    4194304
multiply             23,5 ns       0,06    8388608
multiply             23,5 ns       0,08   16777216
```

It fits that the Mark6 is similar to Mark5, as Mark6 is just Mark5 but more generalized.

```
   Mark7

23,5 ns       0,26   16777216\\
```

Not much to say, as the only difference is, that Mark7 only prints the final measurement.

```
   mark7math

pow            19,3 ns       0,03   16777216
exp            21,2 ns       0,05   16777216
log            11,3 ns       0,05   33554432
sin            14,3 ns       0,07   33554432
cos            14,1 ns       0,02   33554432
tan            19,5 ns       0,03   16777216
asin          213,8 ns       0,38    2097152
acos          202,8 ns       0,38    2097152
atan           43,1 ns       0,12    8388608
```

It fits that count differ as they don't all take the same amount of time. The standard deviation should also be larger for the slower functions which fit.

The differences in our results and those found in Microbenchmarks, can be explained by garbage collection, system, java version, and so on.

2 4.2 Standard Deviation

2.1 4.2 - green

The mean is the average of the values, and in Gausian distribution it is represented by μ.
The standard deviation sdev is the σ and is how much an average differs from the mean.
The variance would be calculated as: σ^2.
The code calculates the standard deviation. The code looks different from what is shown in the slide, since the formula has been somewhat rewritten.

The completed code can be seen in `MeanVar.java`.
And the output when running on the array: `[30.7, 30.3, 30.1, 30.7, 50.2, 30.4, 30.9, 30.3, 30.5, 30.8] 32.0 ns +/- 8.576`.
And the output when running on the array: `[30.7, 100.2, 30.1, 30.7, 20.2, 30.4, 2, 30.3, 30.5, 5.4] 30.0 ns +/- 27.974`.

2.2 4.2 - yellow

Without any 25, the 25 is not an outlier as 25 is within |32.0 - 8.576, 30.0 + 8.576|
The output when running on the array: `[30.7, 100.2, 30.1, 30.7, 20.2, 30.4, 2, 30.3, 30.5, 5.4, 25] 30.0 ns +/- 26.015`.
In this example 25 is not an outlier as 25 is within |30.0 - 26.015, 30.0 + 26.015|.

2.3 4.2 - red

The general purpose java program for calculating mean and standard deviation of a large data file can be found in `MeanVar.java`.

3 4.3 TestTimer

3.1 4.3.1 - green

The large variation in the time measurement is due to other things than the actual code. The JVM setting up and the likes. Therefore at some point, with enough test cases, the hidden work by the JVM will become insignificant.

3.2 4.3.2 - yellow

Using mark7 the result is:

```
hashCode()                          2,5 ns       0,00  134217728
Point creation                     65,0 ns       0,98    4194304
Thread's work                    5066,3 ns      18,75      65536
Thread create                    1094,3 ns      14,96     262144
Thread create start             95605,9 ns     218,66       4096
Thread create start join       222040,8 ns     373,98       2048
ai value = 1433540000
Uncontended lock                    4,8 ns       0,01   67108864
```

All the results makes sense, as creating a thread takes longer time than creating an object. It also makes sense that starting a thread takes longer time, and joining adds extra time.

4 4.4 TestCountPrimes

The result can be seen in `4.4.1.txt` and was done by running with 1, 3, 5, ..., 31 threads.

4.1 4.4.2 - green

As expected the standard deviation decreases with count and the change between the mean results diminish as count increases. When this isn't true it is possibly due to garbage collection.
When running on a 8 core computer, using 13-15 threads seems optimal.

4.2 4.4.3 - yellow

The result can be seen in `4.4.3.txt`.
The performance of AtomicLong is better than LongCounter and the optimal thread count has also been increased from 13-15 to 23.
If a general use adequate built-in classes and methods exist, they should be used.

4.3 4.4.4 - red

The result can be seen in `4.4.5.txt`.
This changed doesn't make the result faster and infact only makes it slower. The result is more similar to that found in 4.4.1.

5 4.5 - Synchronised test

5.1 4.5 - green

We made a minor change to the code. The code given in the exercises was:

```
Mark6("Uncontended lock",i -> {synchronized (obj) {return i;}});
```

This would not work, so we changed it to:

```
Mark6("Uncontended lock",i -> {synchronized (new int[0]) {return i;}});
```

The results from this code where:

```
Uncontended lock                  1585.0 ns    3485.13          2
Uncontended lock                   487.5 ns     330.25          4
Uncontended lock                   452.5 ns     239.34          8
Uncontended lock                   171.3 ns      86.01         16
Uncontended lock                   243.1 ns     224.93         32
Uncontended lock                    63.9 ns      18.55         64
Uncontended lock                    58.9 ns      12.66        128
```

```
Uncontended lock                    66.1 ns      14.36        256
Uncontended lock                    56.7 ns       7.41        512
Uncontended lock                    40.2 ns       3.81       1024
Uncontended lock                    39.5 ns       3.98       2048
Uncontended lock                    39.1 ns       1.35       4096
Uncontended lock                    19.1 ns      14.31       8192
Uncontended lock                    11.3 ns       2.70      16384
Uncontended lock                    10.4 ns       0.57      32768
Uncontended lock                    11.5 ns       0.84      65536
Uncontended lock                     4.7 ns       3.21     131072
Uncontended lock                    19.5 ns      19.28     262144
Uncontended lock                     5.2 ns       2.00     524288
Uncontended lock                     5.7 ns       2.21    1048576
Uncontended lock                     7.2 ns       1.94    2097152
Uncontended lock                     4.3 ns       1.94    4194304
Uncontended lock                     4.8 ns       1.12    8388608
Uncontended lock                     4.3 ns       1.55   16777216
Uncontended lock                     5.0 ns       1.45   33554432
Uncontended lock                     4.9 ns       0.91   67108864
```

These results make sense. There are some outliers due to garbage collection, but otherwise the average execution time goes down with more cases. The number stabilises around 4.5 ns at the end.

6 4.6 Volatile vs. Non-volatile

6.1 4.6.1 - green

The result can be seen in `4.6.1.txt` and was done by adding 2 variables at the top of the class, and calling them in Mark6 in main.

```
static int x1 = 0;
static volatile int x2 = 0;

Mark6("NON-VOLATILE int",i -> {return x1++;});
Mark6("VOLATILE int",i -> {return x2++;});
```

The result indicates that updating a non-volatile int is around 8 times faster than a volatile one.

6.2 4.6.2 - red

```
***VOLATILE EXPERIMENTS***
N=10000000
    Volatile time: 79682100
    Volatile time per iteration: 7.9682097
```

```
N=100000000
    Volatile time: 779600300
    Volatile time per iteration: 7.7960033

N=214748364
    Volatile time: 300
    Volatile time per iteration: 1.3969839E-6


***NON-VOLATILE EXPERIMENTS***
N=10000000
    Non volatile time: 9087900
    Non Volatile time per iteration: 0.90879


N=100000000
    Non volatile time: 12728600
    Non Volatile time per iteration: 0.127286


N=214748364
    Non volatile time: 200
    Non Volatile time per iteration: 9.313226E-7
```

Excluding the last result in both, the times for volatile integers seems similar but smaller than the result of Mark6. Mark6 may take longer time as the result is also added to the dummy variable which mark6 uses. This would also explain why the non-volatile integer test never gets below 1 ns.

The reason why the last result is strange and wrong, we believe is because of the JVM wrongfully trying to optimize the code.