PCPP Assignment 3

Group name: Thread Heresy
Real names : Nedas Surkus, Niclas Abelsen,
Github username: nesu, niab

October 2022

Contents

5.1 **3**
1 3
Nedas' Result 3
Niclas' Result 5
Conclusion 6
2 6
Nedas' Result 6
Niclas' Result 7
Conclusion 7

5.2 **7**
1 7
2 7
Nedas Results 7
Niclas' Result 7

5.3 **8**
1 8
Nedas Results 8
Niclas' Results 8
2 9
3 9
Nedas Results 9
Niclas' Results 10
Conclusion 11

5.4 **11**
Nedas Results 11
Niclas' Result 11
Conclusion 11

5.5 **11**
1 11
2 12
3 12
4 12
5 12
Conclusion 12

6.1 **13**
1 13
2 13
3 14
4 14

6.2 **14**
1 14
2 15

6.3 **16**
1 16
2 17
3 17
4 17

5.1

1

Nedas' Result

```
# OS:   Linux; 5.15.0-47-generic; amd64
# JVM:  Ubuntu; 11.0.16
# CPU:  null; 8 "cores"
# Date: 2022-09-30T10:22:14+0200
```

Mark 1

```
 0.005 s     0.2ns
```

Mark 2

```
24.5 ns
```

Mark 3

```
24.4 ns
24.4 ns
24.0 ns
23.9 ns
23.8 ns
24.0 ns
24.0 ns
24.4 ns
25.4 ns
25.4 ns
```

Mark 4

```
24.9 ns +/-  0.569
```

Mark 5

```
520.0 ns +/-  1205.59          2
115.0 ns +/-    41.16          4
100.0 ns +/-    27.00          8
114.4 ns +/-    99.31         16
32.2 ns +/-     5.32         32
30.5 ns +/-     3.85         64
53.3 ns +/-    73.82        128
30.0 ns +/-     2.25        256
28.7 ns +/-     0.46        512
27.2 ns +/-     0.29       1024
44.6 ns +/-     6.81       2048
49.0 ns +/-     6.11       4096
27.5 ns +/-     2.43       8192
26.1 ns +/-     0.10      16384
```

```
26.0 ns +/-     1.27      32768
26.8 ns +/-     1.78      65536
25.5 ns +/-     0.88     131072
25.9 ns +/-     1.10     262144
25.6 ns +/-     0.62     524288
25.6 ns +/-     1.82    1048576
24.0 ns +/-     0.36    2097152
23.9 ns +/-     0.41    4194304
24.7 ns +/-     0.51    8388608
24.3 ns +/-     0.62   16777216
```

Mark 6

```
multiply                                  655.0 ns    1439.42           2
multiply                                  165.0 ns     127.04           4
multiply                                  132.5 ns      61.01           8
multiply                                  576.9 ns     480.44          16
multiply                                  162.8 ns     126.40          32
multiply                                   39.1 ns       6.21          64
multiply                                   64.9 ns      85.40         128
multiply                                   62.8 ns      22.76         256
multiply                                   72.1 ns      40.16         512
multiply                                   33.4 ns       0.45        1024
multiply                                   34.8 ns       3.92        2048
multiply                                   36.7 ns       4.30        4096
multiply                                   26.7 ns       1.41        8192
multiply                                   34.3 ns      12.64       16384
multiply                                   28.5 ns       3.15       32768
multiply                                   28.4 ns       3.73       65536
multiply                                   25.5 ns       1.07      131072
multiply                                   25.4 ns       0.44      262144
multiply                                   25.4 ns       0.45      524288
multiply                                   26.2 ns       0.96     1048576
multiply                                   24.8 ns       0.44     2097152
multiply                                   24.2 ns       0.64     4194304
multiply                                   24.0 ns       0.30     8388608
multiply                                   24.2 ns       0.33    16777216
```

Niclas' Result

The system info:

```
# OS:   Linux; 5.15.0-47-generic; amd64
# JVM:  Ubuntu; 11.0.16
# CPU:  null; 8 "cores"
# Date: 2022-09-30T10:22:14+0200
```

Mark 1

```
 0.006 s     0.3ns
```

Mark 2

```
31.7 ns
```

Mark 3

```
31.4 ns
28.9 ns
29.0 ns
28.5 ns
28.7 ns
28.7 ns
29.3 ns
28.9 ns
28.7 ns
29.0 ns
```

Mark 4

```
29.2 ns +/-  1.119
```

Mark 5

```
529.5 ns +/-   892.27          2
214.1 ns +/-   104.16          4
191.3 ns +/-    86.66          8
186.4 ns +/-   108.03         16
55.4 ns +/-     8.87          32
54.2 ns +/-    14.00          64
84.7 ns +/-   116.34         128
46.4 ns +/-     1.85         256
48.7 ns +/-     9.77         512
59.3 ns +/-    12.88        1024
54.7 ns +/-    11.17        2048
45.4 ns +/-     0.43        4096
42.7 ns +/-     1.00        8192
42.5 ns +/-     0.54       16384
40.0 ns +/-     1.41       32768
40.8 ns +/-     0.80       65536
38.3 ns +/-     0.18      131072
35.5 ns +/-     1.08      262144
33.7 ns +/-     0.85      524288
34.2 ns +/-     0.59     1048576
```

```
34.5 ns +/-     0.93    2097152
31.2 ns +/-     0.81    4194304
29.5 ns +/-     0.55    8388608
29.0 ns +/-     0.30   16777216
```

Mark 6

```
multiply                              1198.4 ns    2208.24          2
multiply                               501.9 ns     211.89          4
multiply                               453.3 ns     121.27          8
multiply                               595.2 ns     303.63         16
multiply                               281.2 ns     237.51         32
multiply                                73.5 ns      24.62         64
multiply                                95.8 ns     106.58        128
multiply                                62.8 ns       9.16        256
multiply                                60.2 ns       4.13        512
multiply                                98.0 ns     127.65       1024
multiply                                57.0 ns       2.42       2048
multiply                                63.0 ns      17.86       4096
multiply                                43.9 ns       3.01       8192
multiply                                43.1 ns       1.07      16384
multiply                                41.7 ns       2.52      32768
multiply                                39.3 ns       1.62      65536
multiply                                37.8 ns       0.83     131072
multiply                                37.7 ns       0.76     262144
multiply                                36.6 ns       1.74     524288
multiply                                34.9 ns       0.21    1048576
multiply                                34.8 ns       0.96    2097152
multiply                                32.6 ns       0.86    4194304
multiply                                31.2 ns       0.83    8388608
```

Conclusion

On the most tests the results were quite similar to the benchmark notes. In Mark 5, our first values are quite higher. However, they drop to a smaller amount than the benchmark notes.

2

Nedas' Result

Mark 7

```
# OS:   Windows 10; 10.0; amd64
# JVM:  Eclipse Foundation; 16.0.2
# CPU:  Intel64 Family 6 Model 158 Stepping 9, GenuineIntel; 8 "cores"
# Date: 2022-09-30T10:37:46+0200
pow                                     19.9 ns       0.50   16777216
exp                                     21.6 ns       0.48   16777216
log                                     11.6 ns       0.23   33554432
sin                                     14.3 ns       0.16   33554432
cos                                     14.2 ns       0.18   33554432
tan                                     19.7 ns       0.38   16777216
asin                                   214.0 ns       3.40    2097152
acos                                   219.6 ns      24.59    1048576
atan                                    46.0 ns       1.53    8388608
```

Niclas' Result

Mark 7

```
multiply                               28.6 ns       0.47   16777216
pow                                    26.4 ns       0.28   16777216
exp                                    12.6 ns       0.11   33554432
log                                    13.8 ns       0.20   33554432
sin                                    17.4 ns       0.12   16777216
cos                                    17.5 ns       0.04   16777216
tan                                    24.0 ns       0.06   16777216
asin                                   89.0 ns       0.50    4194304
acos                                   90.1 ns       0.64    4194304
atan                                   33.3 ns       0.14    8388608
```

Conclusion

Nedas tests were quite similar to the benchmark notes, however Niclas results were significantly different with regards to asin and acos. This can happen due to internal hardware optimisations of Linux.

5.2

1

For thread create start we see that standard deviation is not consistent making it hard to conclude whether a certain data is plausible, however we see that most is below 3700. The execution time of this is increasing past 128 iterations. This due to the computer running out of cores to execute the threads on. This includes the physical cores and 'pseudo-cores'.

2

Nedas Results

```
# OS:   Windows 10; 10.0; amd64
# JVM:  Eclipse Foundation; 16.0.2
# CPU:  Intel64 Family 6 Model 158 Stepping 9, GenuineIntel; 8 "cores"
# Date: 2022-09-30T11:16:20+0200
Mark 7 measurements
hashCode()                              2.7 ns       0.06  134217728
Point creation                         50.2 ns       0.99    8388608
Thread's work                        5091.9 ns      62.81      65536
Thread create                         790.1 ns      16.21     524288
Thread create start                103687.1 ns   13110.29       4096
Thread create start join           197514.8 ns    4553.02       2048
ai value = 1433540000
Uncontended lock                       18.9 ns       0.23   16777216
```

Niclas' Result

```
# OS:   Linux; 5.15.0-47-generic; amd64
# JVM:  Ubuntu; 11.0.16
# CPU:  null; 8 "cores"
# Date: 2022-09-30T11:31:45+0200
Mark 7 measurements
hashCode()                              3.1 ns       0.02  134217728
Point creation                         72.0 ns       1.58    4194304
```

```
Thread’s work                      6304.5 ns      34.29      65536
Thread create                       955.0 ns       6.62     524288
Thread create start               89439.2 ns     815.44       4096
Thread create start join         132167.1 ns    2357.65       2048
ai value = 1433540000
Uncontended lock                      5.7 ns       0.06   67108864
```

5.3

1

Nedas Results

```
# OS:   Windows 10; 10.0; amd64
# JVM:  Eclipse Foundation; 16.0.2
# CPU:  Intel64 Family 6 Model 158 Stepping 9, GenuineIntel; 8 "cores"
# Date: 2022-09-30T11:23:54+0200
countSequential                 10445217.5 ns  485115.53         32
countParallelN       1           8934301.3 ns  355945.78         32
countParallelN       2           6096533.1 ns  102481.30         64
countParallelN       3           5324375.0 ns  258769.50         64
countParallelN       4           5148182.5 ns  117390.96         64
countParallelN       5           4098550.9 ns  131992.36         64
countParallelN       6           3633265.2 ns   27635.13        128
countParallelN       7           3408648.4 ns   37397.88        128
countParallelN       8           3320344.6 ns   27907.12        128
countParallelN       9           3396380.5 ns  217639.89        128
countParallelN      10           3339469.2 ns   33535.95        128
countParallelN      11           3452120.1 ns   23733.63        128
countParallelN      12           3435350.3 ns   22111.31        128
countParallelN      13           3426508.7 ns   12882.99        128
countParallelN      14           3509044.0 ns   98177.59        128
countParallelN      15           3486849.1 ns   72784.72        128
countParallelN      16           3597066.8 ns   79355.29        128
countParallelN      17           3706318.4 ns  139644.46        128
countParallelN      18           3610687.0 ns   32485.32        128
countParallelN      19           3643112.0 ns   14885.46        128
countParallelN      20           3680590.5 ns   14119.22        128
countParallelN      21           3732906.6 ns   46428.87        128
countParallelN      22           3904524.2 ns  273601.32        128
countParallelN      23           3819443.8 ns   17785.62        128
countParallelN      24           4067699.1 ns  229346.93         64
countParallelN      25           3985852.8 ns   40212.39        128
countParallelN      26           3972341.1 ns   31902.94         64
countParallelN      27           4061984.8 ns   55336.56         64
countParallelN      28           4090054.7 ns   48347.27         64
countParallelN      29           4129176.6 ns   29728.51         64
countParallelN      30           4180190.2 ns   39140.73         64
countParallelN      31           4248093.1 ns   38922.37         64
countParallelN      32           4365744.4 ns   96436.70         64
```

Niclas’ Results

```
# OS:   Linux; 5.15.0-47-generic; amd64
# JVM:  Ubuntu; 11.0.16
# CPU:  null; 8 "cores"
# Date: 2022-09-30T11:37:48+0200
countSequential             11720842.2 ns  165694.20          32
countParallelN       1      12317291.9 ns   98958.45          32
countParallelN       2       9366203.1 ns  665463.55          32
countParallelN       3       6798552.9 ns  171655.01          64
countParallelN       4       7149450.7 ns   71656.80          64
countParallelN       5       7440323.1 ns  116307.14          64
countParallelN       6       6922672.4 ns   33737.23          64
countParallelN       7       6252830.3 ns   22664.21          64
countParallelN       8       5812157.7 ns   51457.44          64
countParallelN       9       7369465.1 ns   64036.53          64
countParallelN      10       7101862.8 ns   60501.21          64
countParallelN      11       6919608.0 ns   90131.15          64
countParallelN      12       6745338.7 ns   85471.35          64
countParallelN      13       6923346.1 ns   53071.51          64
countParallelN      14       6713524.8 ns   45841.82          64
countParallelN      15       6552076.1 ns   82788.91          64
countParallelN      16       6621117.1 ns  189373.83          64
countParallelN      17       6951551.2 ns   54939.62          64
countParallelN      18       7045760.6 ns   50708.56          64
countParallelN      19       7056414.9 ns  195615.44          64
countParallelN      20       6923915.9 ns   71080.09          64
countParallelN      21       6922783.7 ns   57518.47          64
countParallelN      22       6908597.1 ns   93571.09          64
countParallelN      23       6916986.5 ns   56295.86          64
countParallelN      24       6934396.4 ns   90237.41          64
countParallelN      25       7044857.7 ns   53227.48          64
countParallelN      26       7185908.0 ns   30953.56          64
countParallelN      27       7206009.2 ns   26654.53          64
countParallelN      28       7225140.4 ns   60889.78          64
countParallelN      29       7323265.9 ns  266897.09          64
countParallelN      30       7267021.1 ns  103142.11          64
countParallelN      31       7243383.2 ns   71325.43          64
countParallelN      32       7242442.7 ns   61527.66          64
```

2

Yes. Its is plausible. After running the tests we can see that in Niclas tests he has 8 cores and his performance decreases until a certain point (15 threads) and afterwards the execution time increases. For Nedas tests, he has 8 cores as well and his maximum performance is achieved with 8 threads. After 16 threads the performance becomes significantly worse.

Thus we can conclude that there is an optimal ratio between number of cores and how many threads are executed.

3

Nedas Results

```
# OS:   Windows 10; 10.0; amd64
# JVM:  Eclipse Foundation; 16.0.2
# CPU:  Intel64 Family 6 Model 158 Stepping 9, GenuineIntel; 8 "cores"
# Date: 2022-09-30T11:32:03+0200
countSequential             10271621.9 ns  180650.86          32
countParallelN       1      10770782.8 ns   96607.50          32
countParallelN       2       7249893.3 ns   57625.74          64
countParallelN       3       5317360.6 ns  133359.36          64
countParallelN       4          5081381.6 ns  197343.90          64
countParallelN       5          4013043.6 ns   68376.04          64
countParallelN       6          3549691.4 ns   29059.67         128
countParallelN       7          3327590.9 ns   27990.49         128
countParallelN       8          3294472.4 ns   46571.90         128
countParallelN       9          3266313.0 ns   32285.10         128
countParallelN      10          3252411.4 ns   28922.20         128
countParallelN      11          3435788.8 ns   48965.19         128
countParallelN      12          3362911.3 ns   12224.58         128
countParallelN      13          3385419.1 ns   34021.54         128
countParallelN      14          3324222.1 ns   24870.22         128
countParallelN      15          3379133.4 ns   21923.52         128
countParallelN      16          3460743.7 ns   29467.61         128
countParallelN      17          3481112.9 ns   16919.50         128
countParallelN      18          3520023.4 ns   17132.59         128
countParallelN      19          3563441.3 ns   22870.49         128
countParallelN      20          3616000.7 ns   43552.61         128
countParallelN      21          3650285.3 ns   26480.86         128
countParallelN      22          3691244.2 ns   15241.96         128
countParallelN      23          3742509.0 ns   23067.23         128
countParallelN      24          3800786.3 ns   33043.39         128
countParallelN      25          3848313.8 ns   15660.81         128
countParallelN      26          3899354.5 ns   38018.48          64
countParallelN      27          3965658.0 ns   63848.34          64
countParallelN      28          4000662.7 ns   36396.40          64
countParallelN      29          4073441.1 ns   28543.13          64
countParallelN      30          4192110.0 ns  118508.21          64
countParallelN      31          4190104.5 ns   46898.58          64
countParallelN      32          4303714.5 ns   50060.18          64
```

Niclas' Results

```
# OS:   Linux; 5.15.0-47-generic; amd64
# JVM:  Ubuntu; 11.0.16
# CPU:  null; 8 "cores"
# Date: 2022-09-30T11:37:48+0200
countSequential                11720842.2 ns  165694.20          32
countParallelN       1         12317291.9 ns   98958.45          32
countParallelN       2          9366203.1 ns  665463.55          32
countParallelN       3          6798552.9 ns  171655.01          64
countParallelN       4          7149450.7 ns   71656.80          64
countParallelN       5          7440323.1 ns  116307.14          64
countParallelN       6          6922672.4 ns   33737.23          64
countParallelN       7          6252830.3 ns   22664.21          64
countParallelN       8          5812157.7 ns   51457.44          64
countParallelN       9          7369465.1 ns   64036.53          64
countParallelN      10          7101862.8 ns   60501.21          64
countParallelN      11          6919608.0 ns   90131.15          64
countParallelN      12          6745338.7 ns   85471.35          64
countParallelN      13          6923346.1 ns   53071.51          64
countParallelN      14          6713524.8 ns   45841.82          64
countParallelN      15          6552076.1 ns   82788.91          64
countParallelN      16          6621117.1 ns  189373.83          64
countParallelN      17          6951551.2 ns   54939.62          64
countParallelN      18          7045760.6 ns   50708.56          64
countParallelN      19          7056414.9 ns  195615.44          64
countParallelN      20          6923915.9 ns   71080.09         64
countParallelN      21          6922783.7 ns   57518.47         64
countParallelN      22          6908597.1 ns   93571.09         64
countParallelN      23          6916986.5 ns   56295.86         64
countParallelN      24          6934396.4 ns   90237.41         64
countParallelN      25          7044857.7 ns   53227.48         64
countParallelN      26          7185908.0 ns   30953.56         64
countParallelN      27          7206009.2 ns   26654.53         64
countParallelN      28          7225140.4 ns   60889.78         64
countParallelN      29          7323265.9 ns  266897.09         64
countParallelN      30          7267021.1 ns  103142.11         64
countParallelN      31          7243383.2 ns   71325.43         64
countParallelN      32          7242442.7 ns   61527.66         64
```

Conclusion

There is not much significant difference between LongCounter and Atomic Long. Thus we can conclude that it is no difference between using inbuilt classes and methods and using user built classes and methods as long as they are implemented similarly.

5.4

Nedas Results

```
# OS:   Windows 10; 10.0; amd64
# JVM:  Eclipse Foundation; 16.0.2
# CPU:  Intel64 Family 6 Model 158 Stepping 9, GenuineIntel; 8 "cores"
# Date: 2022-09-30T12:23:25+0200
violitileTest                                 10.6 ns       1.35   33554432
incrementTest                                4.3 ns       0.88   67108864
```

Niclas' Result

```
vInc                                8.8 ns        0.12   33554432
inc                                1.4 ns       0.01  268435456
```

Conclusion

There is no significant surprises. volatile gives us a weaker form of synchronisation, thus a small time increase from normal int is expected.

5.5

1

```
[...]
  public synchronized void add(long c) {
    count += c;
  }
  public synchronized void reset() {
    count = 0;
  }
```

2

```
Array Size: 5697
# Occurences of ipsum :1430
```

3

```
Test time search                16416151.0 ns  194706.64         16
```

4

Implementation

```
  private static long countParallelN(String target, String[] lineArray, int N, LongCounter lc){
    Thread[] threads = new Thread[N+1];
    final int arrayLength = lineArray.length;
    final int dividedWork = Math.round(arrayLength/N);

    for (int t = 0; t <= N; t++) {
      final int start = dividedWork *t;
      final int finish;

      // Creates aditional thread if work is missing
      if(arrayLength < dividedWork*(t+1)){
         finish = arrayLength;
      }else
         finish = dividedWork*(t+1);

      threads[t] = new Thread ( () -> search(target, lineArray, start, finish, lc));
    }

    for (int t=0; t<= N; t++)
      threads[t].start();
    try {
      for (int t=0; t<= N; t++)
        threads[t].join();
    } catch (InterruptedException exn) { }

    return lc.get();
  }
```

```
# Occurences of ipsum :1430
# Occurences of ipsum :1430
# Occurences of ipsum :1430
```

5

```
Test time search                 9082334.3 ns  112568.39         32
```

Conclusion

By dividing work between threads, we can easily achieve quicker times than running the code sequentially. This result is not very surprising even if we do take some time to initialise and join the threads.

6.1

1

The code:

```
Benchmark.SystemInfo();
for (int i = 1; i <= 10; i++) {
  final int noOfTrans = i;
  Benchmark.Mark7(String.format("AccountExperiment"), a -> doNTransactions(noOfTrans));
}
```

Doing 10 account experiments where the number of accounts is from 1 until 10. The first column of the result denotes the number of transactions.

```
> Task :app:run
# OS:   Linux; 5.15.0-48-generic; amd64
# JVM:  Ubuntu; 11.0.16
# CPU:  null; 8 "cores"
# Date: 2022-10-07T11:32:17+0200

1,  AccountExperiment             50177848.1 ns   31696.65          8
2,  AccountExperiment            100379420.4 ns   90301.29          4
3,  AccountExperiment            150650349.2 ns  103807.82          2
4,  AccountExperiment            200792439.2 ns  111328.03          2
5,  AccountExperiment            251121671.3 ns  550444.31          2
6,  AccountExperiment            301087890.2 ns  134894.52          2
7,  AccountExperiment            351475909.6 ns  312421.28          2
8,  AccountExperiment            401951523.2 ns  283822.65          2
9,  AccountExperiment            452170148.6 ns  337461.49          2
10, AccountExperiment            502803181.1 ns  193612.69          2
```

We see that each execution time is approximately proportional to the transaction time. The reason why it is not exact is due to overheads. E.g. the time it takes to create the threads.

2

With min and max

```
Transfer 4593 from 4 to 8   Transfer 385 from 1 to 4    Transfer 1590 from 2 to 4
Transfer 4458 from 8 to 9   Transfer 1347 from 7 to 0   Transfer 1984 from 4 to 5
Transfer 1962 from 3 to 5   Transfer 2596 from 8 to 5   Transfer 4157 from 5 to 9
Transfer 4810 from 5 to 1   Transfer 2522 from 3 to 9   Transfer 1590 from 2 to 4
Transfer 2748 from 2 to 9   Transfer 3028 from 8 to 1   Transfer 215 from 5 to 0
Transfer 1095 from 7 to 0   Transfer 3025 from 0 to 7   Transfer 4479 from 2 to 3
Transfer 3810 from 9 to 4   Transfer 3279 from 8 to 6   Transfer 906 from 3 to 8
Transfer 1538 from 2 to 0   Transfer 2983 from 2 to 0   Transfer 1754 from 0 to 3
Transfer 4658 from 9 to 0   Transfer 218 from 9 to 5    Transfer 323 from 3 to 0
Transfer 2516 from 6 to 7   Transfer 3668 from 9 to 1   Transfer 4852 from 6 to 7
Transfer 4075 from 8 to 3   Transfer 1161 from 6 to 3   Transfer 2800 from 5 to 1
Transfer 886 from 7 to 9    Transfer 2687 from 3 to 6   Transfer 4064 from 8 to 0
Transfer 3076 from 0 to 4   Transfer 950 from 1 to 5    Transfer 278 from 0 to 3
Transfer 1960 from 6 to 0   Transfer 3687 from 4 to 5   Transfer 4514 from 1 to 6
Transfer 644 from 6 to 0    Transfer 2647 from 1 to 3   Transfer 2874 from 9 to 0
Transfer 953 from 3 to 6    Transfer 273 from 2 to 0    Transfer 380 from 1 to 3
Transfer 2105 from 7 to 8   Transfer 3541 from 3 to 7
```

Deadlock occurs when no using min and max. The reason for this is that source and target are gotten from the same pool of accounts. If we do not very out the result with min and max, we can find ourselves in a situation where source is equal to the previous target but target has not been released yet. Thus causing a deadlock.

3

See *ThreadsAccountExperimentsMany.java.*

4

See *ThreadsAccountExperimentsMany.java.*

6.2

1

```
countSequential               11681169.7 ns   16770.25          32
countParallelN  1             12166437.8 ns   54743.18          32
countParallelNLocal  1        12121803.8 ns   10328.95          32
countParallelN  2              7790367.2 ns   61969.29          32
countParallelNLocal  2         7750045.9 ns   35847.20          64
countParallelN  3              5503221.7 ns   36152.79          64
countParallelNLocal  3         5471959.1 ns   19177.78          64
countParallelN  4              4411101.4 ns   56647.09          64
countParallelNLocal  4         4405430.8 ns   47695.06          64
countParallelN  5              3722571.0 ns   38131.11         128
countParallelNLocal  5         3711116.6 ns   44411.56         128
countParallelN  6              3614989.1 ns   55830.18         128
countParallelNLocal  6         3622389.8 ns   55612.99         128
countParallelN  7              3854786.0 ns   37892.37         128
countParallelNLocal  7         3826365.1 ns   57804.53         128
countParallelN  8              3635597.3 ns   27010.67         128
countParallelNLocal  8         3630729.1 ns   39380.11         128
countParallelN  9              3432523.4 ns   16330.91         128
countParallelNLocal  9         3429267.5 ns   30769.53         128
countParallelN 10              3286280.0 ns   20346.43         128
countParallelNLocal 10         3259313.8 ns   21112.53         128
countParallelN 11              3141682.8 ns   46584.23         128
countParallelNLocal 11         3105103.0 ns   21039.27         128
countParallelN 12              2998523.3 ns   18381.57         128
countParallelNLocal 12         3005737.0 ns   10981.22         128
countParallelN 13              2982450.1 ns   28281.28         128
countParallelNLocal 13         2926079.0 ns   28500.33         128
countParallelN 14              2973336.8 ns   23048.85         128
countParallelNLocal 14         2971098.1 ns   34155.05         128
countParallelN 15              2982479.8 ns   39957.84         128
countParallelNLocal 15         3121819.8 ns  143729.76         128
countParallelN 16              3003946.6 ns   15828.26         128
countParallelNLocal 16         2978713.0 ns   29362.97         128
countParallelN 17              3024240.3 ns   14437.57         128
countParallelNLocal 17         3000780.8 ns   13670.26         128
countParallelN 18              3022010.5 ns   16405.99         128
countParallelNLocal 18         3016594.1 ns   21094.20         128
countParallelN 19              3007139.8 ns   15286.69         128
countParallelNLocal 19         3028311.6 ns   56759.50         128
countParallelN 20              2993044.0 ns   17148.56         128
countParallelNLocal 20         2980746.7 ns   11629.59         128
```

```
countParallelN 21              2992654.4 ns   11502.34         128
countParallelNLocal 21         2978242.2 ns   15980.00         128
countParallelN 22              2994358.8 ns   15940.86         128
countParallelNLocal 22         3006561.1 ns   40009.54         128
countParallelN 23              3035536.6 ns   43123.52         128
countParallelNLocal 23         2986116.3 ns   15168.21         128
countParallelN 24              3005190.8 ns   10789.82         128
countParallelNLocal 24         2996930.9 ns   17425.24         128
countParallelN 25              3027909.6 ns   14938.77         128
countParallelNLocal 25         3008194.6 ns   10949.75         128
countParallelN 26              3041418.7 ns    8665.96         128
countParallelNLocal 26         3036958.9 ns   18802.13         128
countParallelN 27              3066564.0 ns   12244.18         128
countParallelNLocal 27         3064485.8 ns   20624.29         128
countParallelN 28              3106867.7 ns   15038.45         128
countParallelNLocal 28         3100620.8 ns   10087.87         128
countParallelN 29              3140785.9 ns   14470.74         128
countParallelNLocal 29         3136006.0 ns   15872.83         128
countParallelN 30              3193558.0 ns   10662.66         128
countParallelNLocal 30         3181362.4 ns    9108.27         128
countParallelN 31              3240491.7 ns    8547.12         128
countParallelNLocal 31         3237648.7 ns   11895.09         128
countParallelN 32              3287461.6 ns   14005.81         128
countParallelNLocal 32         3344364.3 ns   90453.98         128
```

The results are expected. local performs similarly to non local version. As thread count increases. So does the processing speed.

2

```
countSequential               11703886.3 ns   57343.70          32
countParallelN  1             12100872.8 ns   19933.32          32
countParallelNLocal  1        12086893.4 ns   28708.33          32
countParallelN  2              7959489.4 ns  143266.73          32
countParallelNLocal  2         7898216.6 ns  101455.89          32
countParallelN  3              5643043.1 ns   88690.94          64
countParallelNLocal  3         5845986.4 ns   77859.81          64
countParallelN  4              4356121.3 ns   30577.17          64
countParallelNLocal  4         4607973.0 ns   79186.38          64
countParallelN  5              3727472.6 ns   48816.47         128
countParallelNLocal  5         4149992.0 ns   68915.79          64
countParallelN  6              3602013.3 ns   64637.03         128
countParallelNLocal  6         4235735.2 ns  114634.56          64
countParallelN  7              3731275.0 ns   42040.70         128
countParallelNLocal  7         4019863.4 ns   31631.51          64
countParallelN  8              3565424.9 ns   34966.73         128
countParallelNLocal  8         3846094.7 ns   37755.66         128
countParallelN  9              3493096.1 ns   23805.28         128
countParallelNLocal  9         3665514.1 ns   29062.45         128
countParallelN 10              3203062.7 ns   25826.80         128
countParallelNLocal 10         3563074.3 ns   21620.77         128
countParallelN 11              3046929.8 ns   14238.58         128
countParallelNLocal 11         3410708.1 ns   21623.13         128
countParallelN 12              2945262.1 ns   17530.46         128
countParallelNLocal 12         3334143.7 ns   12406.27         128
countParallelN 13              2878275.2 ns   27167.13         128
countParallelNLocal 13         3281218.9 ns   12647.10         128
```

```
countParallelN 14              2916695.3 ns   35479.97        128
countParallelNLocal 14         3265353.1 ns   13798.08        128
countParallelN 15              2916283.8 ns   43304.92        128
countParallelNLocal 15         3241645.9 ns   17594.84        128
countParallelN 16              2938420.6 ns   22740.48        128
countParallelNLocal 16         3319951.4 ns   26361.08        128
countParallelN 17              2984937.7 ns   21154.31        128
countParallelNLocal 17         3361832.3 ns   25361.83        128
countParallelN 18              3005573.0 ns   75792.27        128
countParallelNLocal 18         3412850.3 ns   15950.94        128
countParallelN 19              2988674.1 ns   43624.01        128
countParallelNLocal 19         3472205.9 ns   12672.34        128
countParallelN 20              2957489.9 ns   14573.33        128
countParallelNLocal 20         3496729.8 ns   16070.63        128
countParallelN 21              2982809.0 ns   19988.25        128
countParallelNLocal 21         3549494.2 ns   16644.19        128
countParallelN 22              2978384.3 ns   11167.81        128
countParallelNLocal 22         3558898.8 ns   17252.63        128
countParallelN 23              2968538.0 ns   14089.87        128
countParallelNLocal 23         3571487.0 ns   15119.27        128
countParallelN 24              2990088.2 ns   21096.87        128
countParallelNLocal 24         3613582.2 ns    8845.62        128
countParallelN 25              2996263.3 ns   13426.63        128
countParallelNLocal 25         3654489.4 ns   11760.10        128
countParallelN 26              3012673.8 ns   16671.81        128
countParallelNLocal 26         3676822.1 ns   14854.22        128
countParallelN 27              3043794.0 ns   21279.20        128
countParallelNLocal 27         3714385.9 ns   21381.94        128
countParallelN 28              3068495.9 ns   22246.37        128
countParallelNLocal 28         3738555.2 ns   15053.48        128
countParallelN 29              3112514.3 ns   13019.32        128
countParallelNLocal 29         3769508.9 ns   12382.42        128
countParallelN 30              3168635.0 ns   22397.47        128
countParallelNLocal 30         3825757.6 ns    7793.18        128
countParallelN 31              3221237.7 ns   13632.07        128
countParallelNLocal 31         3858665.4 ns   14360.88        128
countParallelN 32              3259932.1 ns   19068.94        128
countParallelNLocal 32         3891021.9 ns   17891.31         64
```

After rewriting the classes to use executors. There is no noticeable difference in terms of speed. But variance seems to be slightly more stable. This is strange and a bit unexpected as logic dictates that by utilising Executors, the execution speed should be increasing.

6.3

1

See `Histogram2.java` for the actual implemenation.

Total and the count need to be made final and private so they have no chance to escape. Total could be switched from int to Atomic int to avoid race conditions but is not needed. Get total should utilise a read lock to avoid stale data. Increment should have a write lock implemented to execute the critical section of adding to total and count. Get count should have a read lock implemented so values are not accessed when being written to. Get percentage should also have a read lock to not get stale data from either of the methods its accessing. Only method that does not need a synchronisation is getSpan as the size of count has no chance of increasing. However, it has to be noted that there could be a wrong data when accessing this method before count is fully initialised.

2

See `Histogram3.java`

3

See `HistogramPrimeThreads.java`

4

```
Histogram2                  3288550500.0 ns 406554238.21          2

Histogram3  1               2140301245.0 ns 72288735.52          2
Histogram3  2               1622910770.0 ns 21403862.67          2
Histogram3  3               1597729520.0 ns 11342022.08          2
Histogram3  4               1566118895.0 ns 22459939.49          2
Histogram3  5               1547482995.0 ns 5399088.49          2
Histogram3  6               1556947505.0 ns 15460783.83          2
Histogram3  7               1547371640.0 ns 10973632.85          2
Histogram3  8               1551295330.0 ns 22065662.29          2
Histogram3  9               1573640115.0 ns 15060293.60          2
Histogram3 10               1559738100.0 ns 31545280.47          2
Histogram3 11               1584976185.0 ns 34390176.58          2
Histogram3 12               1556109705.0 ns 32122596.86          2
Histogram3 13               1557255360.0 ns 11153815.06          2
Histogram3 14               1573382615.0 ns 40911443.66          2
Histogram3 15               1584244000.0 ns 42605485.21          2
Histogram3 16               1581638505.0 ns 41767737.05          2
Histogram3 17               1590000425.0 ns 32653825.66          2
Histogram3 18               1588993485.0 ns 30703854.60          2
Histogram3 19               1572040715.0 ns 21103488.32          2
Histogram3 20               1579909875.0 ns 12310053.02          2
Histogram3 21               1587189160.0 ns 25681333.18          2
Histogram3 22               1562478140.0 ns 33150232.57          2
Histogram3 23               1627102715.0 ns 62493697.85          2
Histogram3 24               1588442185.0 ns 35742298.14          2
```

Histogram 3 is a significant improvement over Histogram 2. As it takes less time for each thread to get control of the array lock. However, there is barely any increase in speed when increasing the lock count on Histogram 2. This is due to the fact that it is very unlikely that 2 threads would get the same prime factor when calculating the result. Even if they would, the wait time for the lock to be released is very small. Thus, there is minimal speed gain. However, it has to be noted that there is some speed gain from 1-3 locks. Due to them locking up the increase execution more often.