Oblig 3 – IN 3030

Marcusti@student.matnat.uio.no

Introduction

This Oblig was divided into two parts:

a) Parallelize an implementation of the sieve of Eratosthenes, find all primes upto N.
b) Factorize the 100 Largest number less than N*N, where N varies between 2 million and 2 billion. Both sequentially and in parallel.

User Guide

To compile the program, run the command 'javac *.java'. To run the program, run the command "java Oblig3 'N' 'K' ". So if I want to generate all primes up to 2 million and use all available cores of my current machine, I would write – Java Oblig3 2000000 0.

Parallel sieve of Eratosthenes

The file SievePara.java includes my implementation of almost everything, except a monitor which is located at Monitor.java.

My implementation begins by sequentially finding the first couple of primes. If the given N is larger than 100 000, I take the square root of N and find the first primes up to this. This means if N = 2 000 000 000, I choose to sequentially find the first 44 700-ish primes.

My idea for parallelizing the sieve was to let each of the cores have their own sieve, go through each their own primes, and cross out all multiples of the given primes up to N. After this process has been handled I merge each of the sieves back with the main sieve. Finally we can sequentially go through the sieve, adding all numbers not crossed out to a list of primes. I did not implement a parallel version of collecting the primes, even though this could be implemented by letting each thread collect primes from each their own sub-index of the sieve.

Parallel factorization of a large number

I begin by dealing each thread a list of primes, in a round-robin fashion. This is to deal with the load imbalance of lower primes being used more frequently than the larger ones. I then use a Monitor for communicating which number we should factorize next. Let's call the main thread for Master, and the other threads Workers.

My idea is that the Master puts a big number into the Monitor and each worker grabs this number and begins factoring. The factoring is done by trying to divide the current big number with each threads' primes, until we have a rest of zero. This signals we have found a factor, and it is communicated to the Monitor. The Monitor handles all queues, and makes sure that the factors found are stored correctly.

In my implementation each thread only checks through all their primes once, and if they didn't find a factor, wait until a new updated currentNumber exist in the Monitor. This is done by the usage of queues inside the monitor, where workers await an update in the currentNumber. I can now increment a counter in the Monitor for each thread that fails finding a new factor. If this counter becomes equal to the number of workers, no one found a new factor and the current number is a new prime larger than N.

I've updated the logic behind the queues, and I no longer run into deadlocks.

Implementation

The sieve

The parallel implementation is heavily inspired by the sequential solution given. Many of the methods are here either exact or almost exact copies of the methods given by the TA's.

Worker

This class is used as a thread in finding the primes up to N. For each of the primes given in an array, cross out all multiples in the local sieve. Finally, 1 by 1, merge the local sieves with the Master's sieve.

Monitor

The monitor handles the communication between the master and the Factory Workers. It is also responsible for printing/saving the factors for each base.

putNum() - The Master waits until the currentNumber is 1. If this is the case, update currentNumber to the next big number, and let all workers know we have a new number.

The FactoryWorkers communicate through either getNum() or updateNum(). The getNum() method is used to update which number we currently want to factorize. If a Worker asks for a new number, but it already has the currentNumber, we want it to wait in a queue until the currentNumber is updated. If all Workers have communicated that they want a new number, but none has found a factor, we know that the current number is a prime.

updateNum() is used when a worker finds a factor. This method stores the factor found, and divides the currentNumber by the factor, updating the number we want to factorize and signaling all threads to begin factorizing again.

FactoryWorker

This class represents a factoring thread. Ask the monitor what the currentNumber is. Thereafter go through all of the local primes and check if one of these primes is a factor in the currentNumber we want to factorize.

Measurements.

The results below show the median of 7 runs on each of the following N values.

N = 2 Million.

```
mvrcus@Mvrcus:~/Documents/IN3030/Oblig3$ java Oblig3 2000000 0
--------Threads Available-------
                 4
--------Sequential Sieve------------
27.732234ms
12.783624ms
11.43059ms
11.281146ms
12.307295ms
11.343914ms
13.444915ms
-------------------------------------
      Median Time Sequential
           12.307295ms
-------------------------------------
---------Parallel Sieve-------------
29.741656ms
10.552308ms
5.994924ms
5.901506ms
6.856156ms
6.232023ms
5.830213ms
-------------------------------------
      Median Time Parallel
           6.232023ms
-------------------------------------
-------------------------------------
      Speed Up S/P
          1.9748
-------------------------------------
```

```
-------------------------------------
Errors: 0
Primes: 148933
-------------------------------------
-------Sequential Factoriazation------
174.325461ms
157.898073ms
160.890579ms
156.351258ms
156.295111ms
157.620551ms
155.972082ms
-------------------------------------
      Median Time Sequential
           157.620551ms
-------------------------------------
-------Parallel Factoriazation------
117.807477ms
96.589214ms
102.084545ms
95.603043ms
101.205714ms
122.316636ms
103.045457ms
-------------------------------------
      Median Time Parallel
           102.084545ms
-------------------------------------
-------------------------------------
      Speed Up S/P
           1.5440197240434386
-------------------------------------
```

N = 20 Million.

```
mvrcus@Mvrcus:~/Documents/IN3030/Oblig3$ java O
---------Threads Available--------
                 4
---------Sequential Sieve-------------
142.185411ms
122.085456ms
115.439683ms
115.104936ms
114.853727ms
118.166667ms
114.733693ms
--------------------------------------
      Median Time Sequential
           115.439683ms
--------------------------------------
----------Parallel Sieve--------------
88.190687ms
60.425803ms
55.500385ms
85.300114ms
55.221619ms
55.556733ms
57.176713ms
--------------------------------------
      Median Time Parallel
           57.176713ms
--------------------------------------
--------------------------------------
      Speed Up S/P
          2.0190
--------------------------------------
```

```
--------------------------------------
Errors: 0
Primes: 1270607
--------------------------------------
-------Sequential Factoriazation------
1284.117514ms
1257.146157ms
1257.908769ms
1258.240088ms
1257.844638ms
1265.336599ms
1255.810123ms
--------------------------------------
      Median Time Sequential
           1257.908769ms
--------------------------------------
-------Parallel Factoriazation------
840.171878ms
807.97278ms
819.100853ms
817.248188ms
810.397079ms
790.066479ms
799.314489ms
--------------------------------------
      Median Time Parallel
           810.397079ms
--------------------------------------
--------------------------------------
      Speed Up S/P
           1.5522128615668418
--------------------------------------
```

N = 200 Million.

```
mvrcus@Mvrcus:~/Documents/IN3030/Oblig3$ java Oblig3 200000000 0
---------Threads Available--------
                 4
---------Sequential Sieve-------------
1511.946161ms
1685.470147ms
1806.769347ms
1490.879325ms
1470.9478ms
1475.106322ms
1474.930643ms
--------------------------------------
      Median Time Sequential
           1490.879325ms
--------------------------------------
----------Parallel Sieve--------------
906.056032ms
892.664037ms
932.551664ms
900.067453ms
808.548993ms
808.119738ms
806.517828ms
--------------------------------------
      Median Time Parallel
           892.664037ms
--------------------------------------
--------------------------------------
      Speed Up S/P
          1.6701
--------------------------------------
```

```
--------------------------------------
Errors: 0
Primes: 11078937
--------------------------------------
-------Sequential Factoriazation------
11055.033717ms
10810.856852ms
10808.038272ms
10819.6128ms
10812.192022ms
10808.464709ms
10823.552305ms
--------------------------------------
      Median Time Sequential
           10812.192022ms
--------------------------------------
-------Parallel Factoriazation------
6259.610246ms
6253.033398ms
6175.649557ms
6296.454462ms
6476.505352ms
6290.796979ms
6246.789545ms
--------------------------------------
      Median Time Parallel
           6259.610246ms
--------------------------------------
--------------------------------------
      Speed Up S/P
           1.7272947670997851
--------------------------------------
```

N = 2 Billion.

Now my laptop began to struggle, so I had to do the calculation on my desktop pc with more ram.

```
C:\Users\Macos\Desktop\Oblig3>java -Xmx14000m Oblig3 2000000000 0
---------Threads Available--------
                 8
---------Sequential Sieve-------------
20933.068892ms
21745.712759ms
21140.940185ms
21019.659888ms
21188.510623ms
21367.463357ms
21714.308022ms
--------------------------------------
      Median Time Sequential
           21188.510623ms
--------------------------------------
----------Parallel Sieve--------------
8467.495525ms
8399.364135ms
7962.861212ms
8961.290036ms
9258.116199ms
8993.599536ms
8708.801513ms
--------------------------------------
      Median Time Parallel
           8708.801513ms
--------------------------------------
--------------------------------------
      Speed Up S/P
          2,4330
--------------------------------------
```

```
--------------------------------------
Errors: 0
Primes: 98222287
--------------------------------------
-------Sequential Factoriazation------
105636.862211ms
104055.933164ms
105165.757306ms
115425.342571ms
108402.565931ms
103875.057988ms
104076.320617ms
--------------------------------------
      Median Time Sequential
           105165.757306ms
--------------------------------------
-------Parallel Factoriazation------
33811.682503ms
35484.911283ms
34390.214114ms
34610.073442ms
34321.040866ms
35892.730583ms
35149.166671ms
--------------------------------------
      Median Time Parallel
           34610.073442ms
--------------------------------------
--------------------------------------
      Speed Up S/P
           3.0385881001448354
--------------------------------------
```

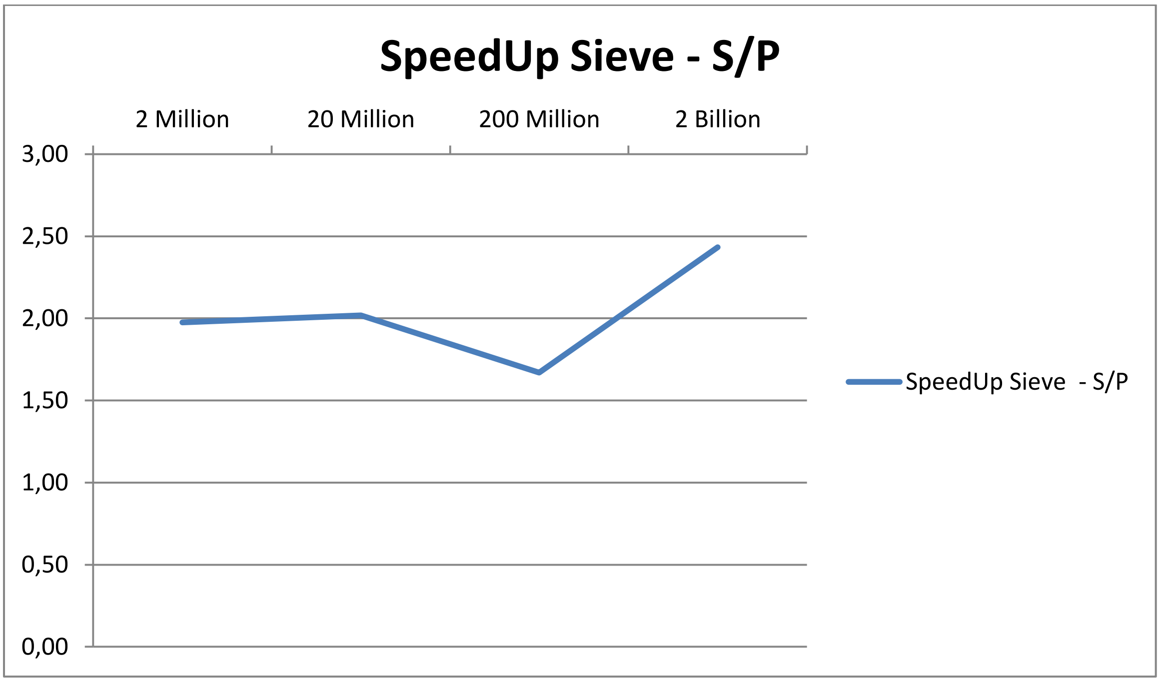

Figure 1: Y axis shows the speedup, S/P for the Sieve. It looks like it's increasing until Memory becomes a bottleneck, somewhere between N = 20-200 million. The parallel solution is here the superior implementation. Note that the final calculation done for 2 Billion is done on another computer, and does therefor not suggest anything interesting beyond proving that the parallel is indeed faster than the sequential solution for n = 2 billion.

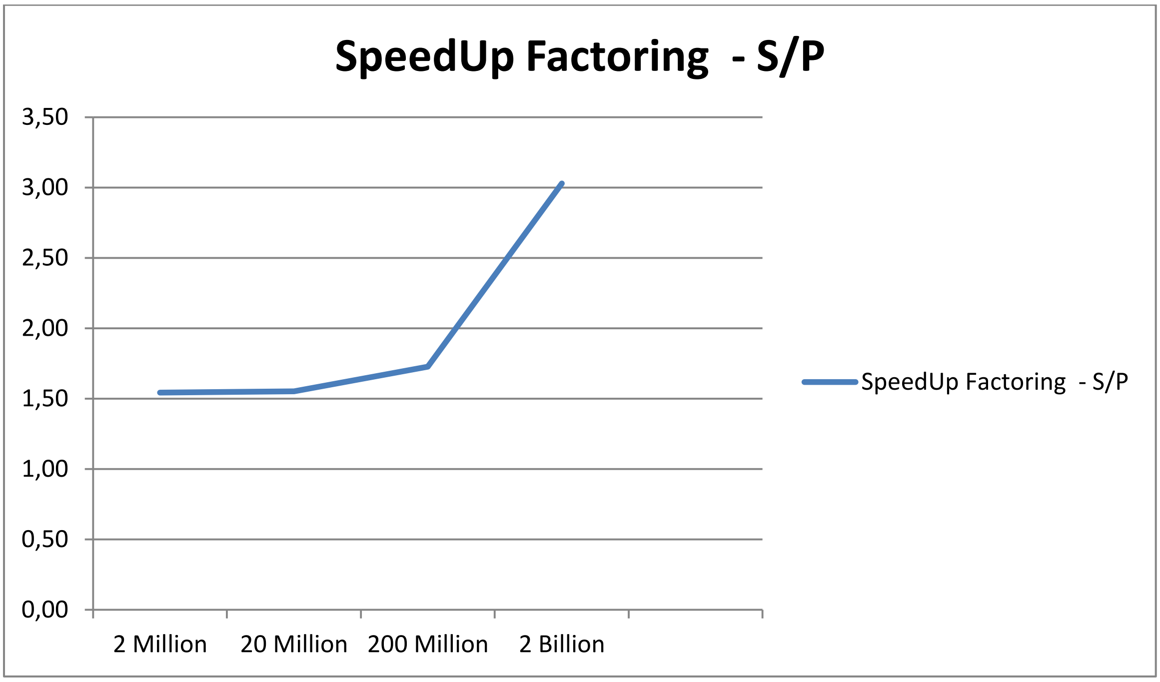

Figure 2: Speedup of the factorization. The speedup seems slowly increasing for n valued between 2 million and 200 million. Again, the run for 2 billion is done on my desktop pc and cant be directly compared with the other results. I am quite pleased with a speedup of over 3 for my parallel factoring, running with 8 cores!

```
3999999999999999945 : 3*5*31326653*8512453171
3999999999999999946 : 2*278347*1649303*4356553
3999999999999999947 : 7*229*1531*1629863665979
3999999999999999948 : 2*2*3*64169297*5194592257
3999999999999999949 : 42667*360053*260376299
3999999999999999950 : 2*5*5*257*24953*12474814519
3999999999999999951 : 3*3*29*449*1327*51971*494927
3999999999999999952 : 2*2*2*2*11*22727272727272727
3999999999999999953 : 227*17621145374449339
3999999999999999954 : 2*3*7*19993*4763572012109
3999999999999999955 : 5*799999999999999991
3999999999999999956 : 2*2*999999999999999989
3999999999999999957 : 3*13*17*36791*202717*808937
3999999999999999958 : 2*19*61*1725625539257981
3999999999999999959 : 37*16703*599009*10805141
3999999999999999960 : 2*2*2*3*3*5*2071723*5363222357
3999999999999999961 : 7*7*7*7*78031*103651*205981
3999999999999999962 : 2*109*173*1787*59351656159
3999999999999999963 : 3*11*6947*17448124544713
3999999999999999964 : 2*2*23*71*307*2251*6257*141623
3999999999999999965 : 5*73*683*1712017*9372131
3999999999999999966 : 2*3*666666666666666661
3999999999999999967 : 83*2207939*21827039191
3999999999999999968 : 2*2*2*2*2*7*31*127*127*103561*344863
3999999999999999969 : 3*3*3*19141747*7739531201
3999999999999999970 : 2*5*13*3259*320011*29503081
3999999999999999971 : 21319*1396141*134389049
3999999999999999972 : 2*2*3*333333333333333331
3999999999999999973 : 263*1847*3803*2165264831
3999999999999999974 : 2*11*17*10695187165775401
3999999999999999975 : 3*5*5*7*59*113*2857*19801*20201
3999999999999999976 : 2*2*2*313*1033*1546412477693
3999999999999999977 : 19*41*79*839*35521*2180963
3999999999999999978 : 2*3*3*222222222222222221
3999999999999999979 : 383*10443864229765013
3999999999999999980 : 2*2*5*29*599*31139*369743471
3999999999999999981 : 3*8861*150472106233307
3999999999999999982 : 2*7*593*481811611659841
3999999999999999983 : 13*41732101*7373036591
3999999999999999984 : 2*2*2*2*3*43*691*983*3943*723589
3999999999999999985 : 5*11*72727272727272727
3999999999999999986 : 2*199*199*293*196853*875617
3999999999999999987 : 3*3*23*508637*37991084993
3999999999999999988 : 2*2*47*1283*949261*17469877
3999999999999999989 : 7*89*503*12764504465981
3999999999999999990 : 2*3*5*170809*780598992637
3999999999999999991 : 17*53*211*233*2393*37735849
3999999999999999992 : 2*2*2*223*208513*10753058401
3999999999999999993 : 3*139*4024357*2383567397
3999999999999999994 : 2*112957699*17705743103
3999999999999999995 : 5*159059*303539*16569799
3999999999999999996 : 2*2*3*3*3*3*7*11*13*19*37*52579*333667
3999999999999999997 : 421*9501187648456057
3999999999999999998 : 2*432809599*4620969601
3999999999999999999 : 3*31*64516129*666666667
```

Figure 3: Results of parallel factoring, N = 2 Billion.

Conclusion

As we can see from the graph, it is clear that the parallel Sieve is significantly better than the sequential one for large N, even considering the overhead caused by extra layers of communication, synchronization and creation of threads. The overhead quickly pays of for larger N, until the memory becomes the bottleneck and the speedup slows down. In conclusion the parallel version is the clear winner when it comes to performance.

The parallel factoring is also the better implementation when it comes to factoring. I am quite pleased with a speedup of over 3.0 when run on my 8-thread desktop computer!

Appendix

The output of my program is the result of running each algorithm 7 times. Currently the sequential and parallel factoring both print a output.txt file, n.txt and n+1.txt. Where the sequential writes to n+1.txt, just so we can compare them without overwriting the files.