IN3030 – oblig 3

UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

Daniel Sharifi – danish

Contents

1. Introduction

This report is about parallelization sieve of Eratosthenes and prime number decomposition (factorization). It discusses and compares the sequential and parallel version of the algorithms, and aims to see if there are benefits to gain by parallelizing.

2. User guide

java Main [mode] [n] [threads]

Sequential:

- java Main 0 [n]
- java Main 0 125000 n = 125 000.

Parallel:

- java Main 1 [n] [threads]
- java Main 1 200000 10 n = 200 000, threads = 10.

Benchmarking (for performance results). This will run all 4 numbers specified in the oblig 9 times and print out the median time of each n.:

- java Main 2 [threads]
- java Main 2 10

3. Parallel Sieve of Eratosthenes

The algorithm can be split into four parts. Assume 'n' is the maximum number we are searching primes for.

1. Find all primes up to $\sqrt{\sqrt{n}}$, and cross out all multiples of these primes (but not itself). We now have all primes up to $\sqrt{n}$.

The threads are each given a partition from $\sqrt{n}$ to n of the byte array where they are going to flip multiples of all primes from 3 to $\sqrt{n}$. The sieve is now finished and all the unflipped numbers are primes. The threads will again go through their partition and store all unflipped numbers in a local arraylist. When this is done the thread meet a barrier.

After this the thread with id 0, will go through all these local arraylists and find the count of primes, to initialize an array where all the primes will be merged. When this is done the thread meet a barrier.

At last the array which was initialized by thread-0 is filled up with the local primes from every thread (in parallel)

4. Parallel factorization of a large number

The parallelization of the factorization is simple. Each thread check every "number of threads"th prime and store their local factorization in an local ArrayList. All the threads do this for 100 numbers without a barrier until they are all done.

```
    void factorize_partly(long n) {
        int index = id;
        int nIndex = (int) (base - n);

        // factors sotred locally
        ArrayList<Integer> factors = nestedFactors.get(nIndex).get(id);

        while (index < primes.length && Math.pow(primes[index], 2) <= n) {
            if (n % primes[index] == 0) {;
                factors.add(primes[index]);
                n /= primes[index];

            } else {
                index += threads;
            }
        }

    }
```

The threads will continue to take every number of threads'th number of the 100 numbers to prime factorize and merge all the factors into one array.

5. Implementation

Sieve:
The sieve returns an int array with all the primes up to a specified argument n.

Factorizer:
The factorizer is somewhat more complex, at least the parallel. It stores all the threads' local prime factors for all the hundred numbers in a 3 dimensional arraylist. In this fashion:
n ---> threads ---> factors

where index 0 has all the threads factors for n*n, and index 1 has all the factors for n*n-1, and etc.

By having it global the threads can see each others factors to be able to also parallelize the merging of the prime factors.

The factorizer returns an two dimensional int array where the primes are merged from the arraylist, with this structure: n ---> factors.

Main:

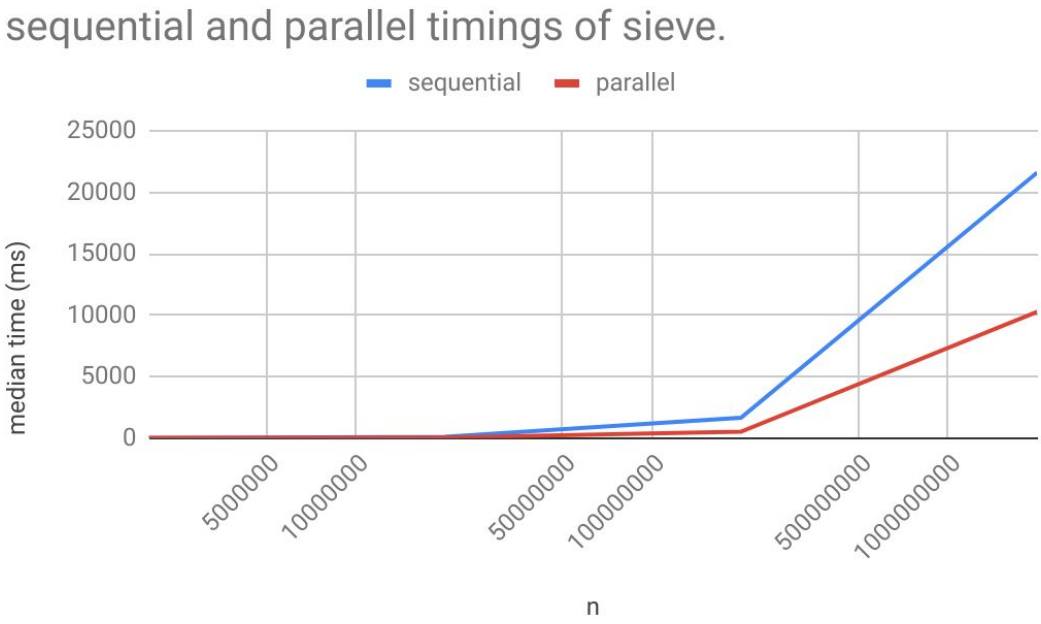
The java program takes two or three parameters depending on the mode to be selected. The benchmarking mode will run values of N: 2 million, 20 million, 200 million, 2 billion 9 times to allow for JIT compilation, on both the sequential and parallel version. After doing the all the function calls, the output of the sequential and parallel versions of the sieve and the factorization are compared and asserted to be equal. At last the factorization is also checked by multiplying all the factors to check if it is correct.

All the modes will save the results to an output by using the Oblig3Precode attached at the course website..

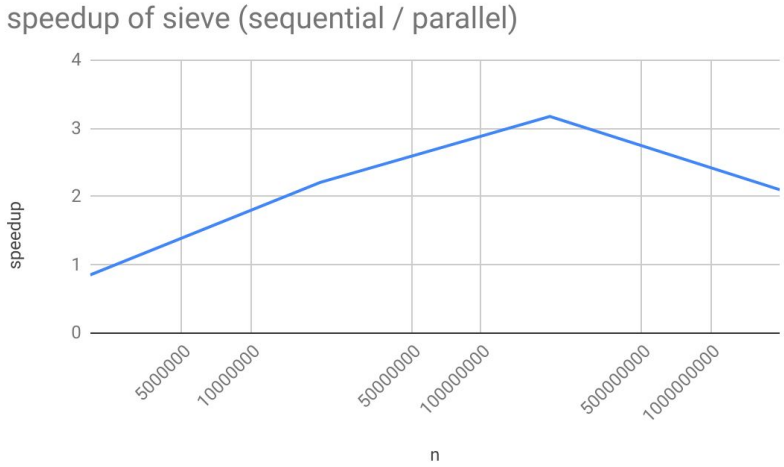
6. Measurements

Timings in milliseconds for the sieve.

sequential	parallel	n
11.593341	13.597139	2000000
112.363646	50.855977	20000000
1659.379423	522.476226	200000000
21600.6649	10277.75416	2000000000

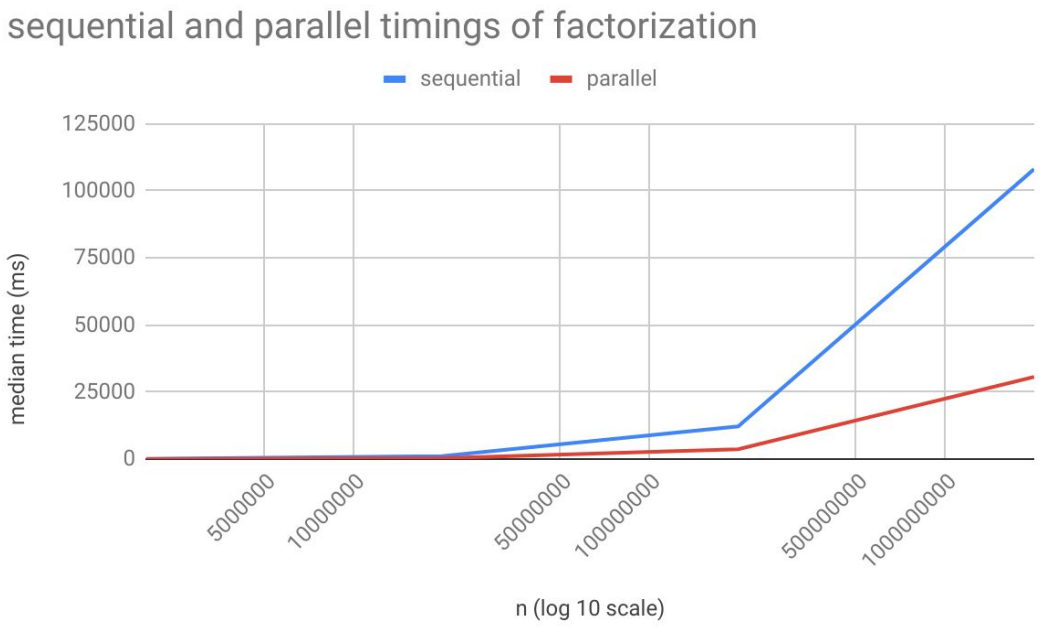


speedup	n
0.8526309101	2000000
2.209448183	20000000
3.175990295	200000000
2.101691145	2000000000

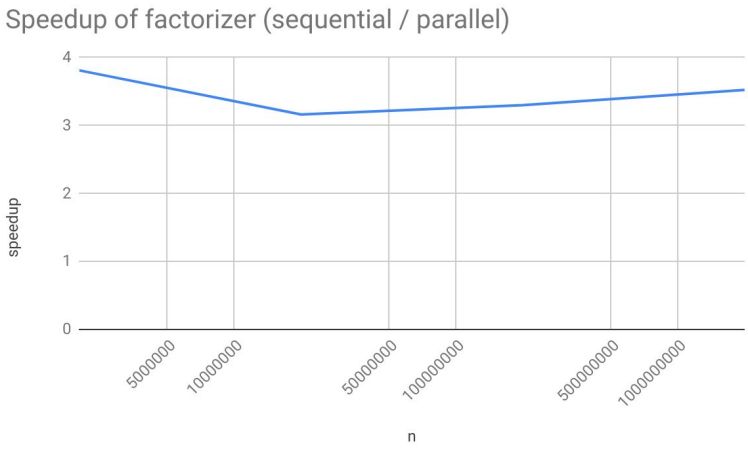


Factorization:

sequential	parallel	n
150.15894	39.39185	2000000
1184.753741	374.56569	20000000
12240.746	3710.240425	200000000
108115.9784	30671.15233	2000000000



speedup	n
3.811929117	2000000
3.163006577	20000000
3.299178651	200000000
3.525005425	2000000000



From the data (especially the tables) it is very noticeable that the speedup is decreasing significantly as n grows, and then it keeps increasing steadily as n grows. I think this is due to the garbage collector kicking in since the parallell versisons of both algorithms rely heavily on disposable objects such as ArrayLists while doing the computation. For the measurements I have used 10 threads on a machine with 8 cores.

However since the computation is the main chunk of work, even with the garbage collector kicking in the algorithm is able to keep a steady speedup greather than 3.

7. Output.

This is copy and pasted straight from the terminal after running the following command:

java -Xmx6G Main 2 10

Timings: (all times are in ms)

Sieve:

n:2000000

sequential: 11.593341

parallell: 13.597139

n:20000000

sequential: 112.363646

parallell: 50.855977

n:200000000

sequential: 1659.379423

parallell: 522.476226

n:2000000000

sequential: 21600.664898

parallell: 10277.754156

Factorization

n: 2000000

sequential: 150.15894

parallell: 39.39185

Speedup: 3.811929117317415

n: 20000000

sequential: 1184.753741

parallell: 374.56569

Speedup: 3.163006577030587

n: 200000000

sequential: 12240.746002

parallell: 3710.240425

Speedup: 3.2991786514751262

n: 2000000000

sequential: 108115.97835

parallell: 30671.152325

Speedup: 3.5250054254360332

8. Conclusion

We can conclude that parallelization of the two algorithm does indeed give good speedup. It would probably be even better if the parallelization was done in such a manner keeping disposable objects at a minimum and in general using primitive values instead of wrapper objects such as ArrayLists.