IN3030 - Obligatorisk Innlevering 3

Aleksander Rosseland Kvåle

April 14, 2023

1 Introduction

In this experiment we are working with the Sieve of Eratosthenes algorithm in order to efficiently find all prime numbers up to a certain value. In addition the second part of our program factorizes very large values using the list of prime numbers generated. Here I've created my own algorithm not based in any previously discovered mathematics. It is a "trial and error" method in order to factorize the numbers. We have also included parallel versions of these two algorithms in order to measure potential speedup from multi-core use.

2 User Guide

Running this program is as simple as first compiling all files using § javac *.java. Then run the program by using § java Main ¡value to compute¿ ¡number of threads¿.

3 Parallel Sieve of Eratosthenes

The parallelization of the Sieve of Eratosthenes is implemented using several worker threads in which the work is split into. The algorithm is based on the same principle as the sequential method, and marks numbers that are composite. Each worker thread takes a range of numbers to sieve and stores its local results in an array. This is then synchronized by making sure each worker only works on a certain local range set by the program. This way multiple workers can mark composites in a wide variety of values. After sieving, the worker thread counts the number of primes and stores them in a local array. They all wait for eachother before merging all the results into the final result of the algorithm.

4 Parallel factorization of a large number

As in similarity with the previous parallelization this method also splits the work between worker threads. Again we store values locally and combine them in the end to find the final product of each value.

5 Implementation

The Sieve of Eratosthenes is implemented as given in the precode. From there I've created the parallel version of this algorithm which is it's own class. Every algorithm has gotten it's own class to keep the code clean and readable, especially if one only wants to look at one aspect/algorithm. The parallel sieve algorithm reuses some of the methods from the sequential algorithm, and then adds some aspects of synchronization to make it work over multiple threads.

The Factorization algorithm is rather straight forward and follows a trial and error principle which tests the primes we have in our list. Then stores these locally before returning them to the main program.

All methods are put in each individual class and the Main method only serves as a tool to run the program and print the information we need to measure speedup.

6 Measurements

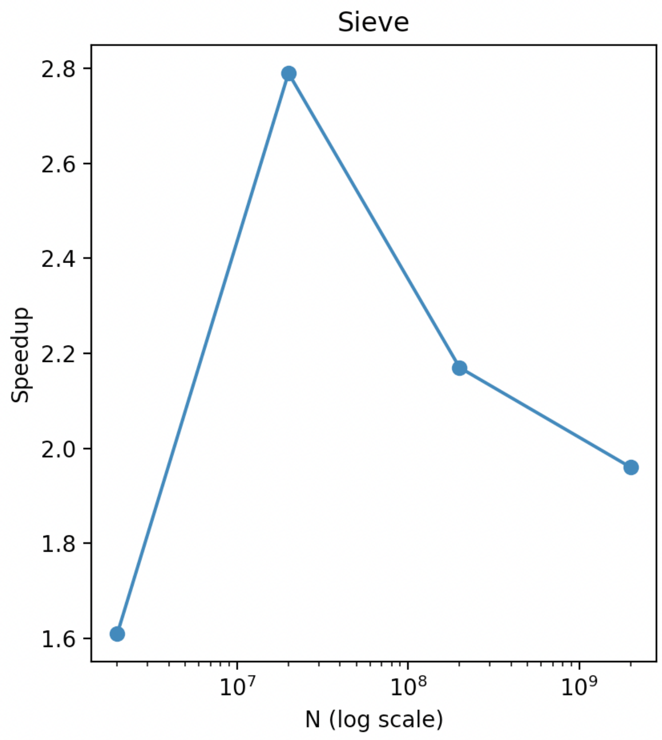

Figure 1: Sieve speedup

Lets first look at the Sieve speedup which we can see has a decent speedup already from the first value of n of 2 million. From here we reach our top at 20 million and then from there the speedup decreases significantly. The speedup overall is good on an 8 core machine. We have not reached the full potential of 7 speedup, but there will be time lost when we run the parallel algorithm as it has more steps than the sequential. We have to combine our results in the end and this results in some extra runtime. It seems as 20 million is the sweet spot for this implementation of the parallel algorithm. This could be directly linked to the amount of work the computer does with the different values. Normally we would expect the parallel algorithm to pull further and further away from the sequential as we introduce higher and higher values of n. However, in this case it is not so. I believe this has to do with the fact that for larger numbers we are handling more data at once and thus moving away from the faster L1 and L2 caches, as we exceeed the sizes of these our computer reaches for the much slower L3 cache or in worst case RAM as we store such large values at once. A computer with a larger L1 cache might prove to have better speedup for higher values of n than we managed here. Factorization seems to be struggling with

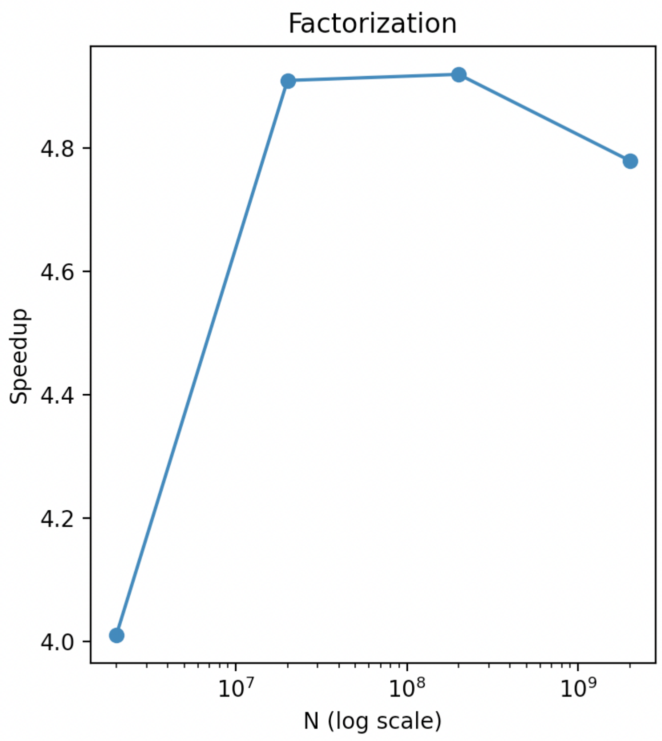

Figure 2: Factorization

the same problems as the Sieve parallel algorithm. Again we seem to reach a ”peak” in speedup, and then gradually losing some speedup for larger values of n. It does not hit as hard on the performance as the sieve, but we still see a decrease for the largest values of n. I do however believe this implementation is much better as we see speedups of close to 5, which is very good for our 8 thread processor.

7 Conclusion

We have managed to solve the task with a speedup larger than 1. I do believe the parallel Factorization is the strongest of the two implementations as this measured a much higher speedup than the Sieve.

8 Appendix

```
(base) kvale@Battlestation oblig3 % java Main 2000000000 8
Running Sieve with sequential algorithm..

Finished sequential run, time elapsed: 13592.8960ms.

Running Sieve with parallel algorithm..

Finished parallel run, time elapsed: 6904.7277ms.

Running Factorize with sequential algorithm..

Finished Factorization run, time elapsed: 32768.7001ms.

Running Factorize with parallel algorithm..

Finished Factorization run, time elapsed: 6851.1713ms.
```