# IN3030 Oblig3 report - hermagst@uio.no

## User guide

All 4 algorithms have a separate main function, but these were mostly for debugging purposes. To validate the algorithms and generate more meaningful benchmarks please run the TestAll.java file. This file will run all the algorithms. It takes 3 arguments:

- n: the number of primes to find and the base for factorization
- k: the number of elements below n to factorize(defaults to 100 like in oblig text)
- threads: the number of threads to use for the parallel algorithms(defaults to all available processors).

Example usage:

```
$ javac *.java

$ java TestAll 200000000
```

## Parallel Sieve of Eratosthenes

My solution here had a couple iterations. I first tried a more direct mapping from sequential to parallel, with each bit in every byte of the oddNumbers were a flag for whether a number was composite or not. This worked for smaller values of n. However for bigger values i noticed that sometimes the parallel version would skip over a couple primes. I figured this was because simultaneous updating of the same byte would result in a race condition where sometimes one (or more) of the threads changes would be overwritten by the other thread(s).

After that i tried a very simple ArrayList approach where i would add all composite numbers to the arraylist and then synchronize with the results at the end. However this resulted in some quite ridiculous memory usage (even allocating 20GB of ram would cause the program to crash for larger n values).

Finally I arrived at my final solution which is quite similar to the sequential, just using a full byte for each composite number flag. This is quite memory inefficient (7-8x worse than for the sequential), but this seems to be the only way to transfer this version of the algorithm to a parallel version.

I also noticed that a significant portion of the execution time was spent inside the sequential collectPrimes() method, so I experimented with different approaches to this. I ended up with a solution quite similar to the sequential.

## Parallel factorization of large numbers

To parallelize the factorization of large numbers I made two runnable classes. One for factorizing a single number and for starting the threads for each number in an interval. The main thread makes one interval class for each core assigned to the program which then in turn start a new thread for each individual prime number factorization.

I also tried just calculating all the factors in the interval objects instead because that would seem more effective than creating more threads than we have on our computer and then having to deal with more context switches than necessary and synchronization overhead. However this approach seemed to have approximately the same/very slightly slower runtimes than my current approach.

For storing the factors I went with the arraylist datastructure because we cannot know how many factors a given number will have before we finish factorizing it. Therefore we would have to use some sort of dynamic datastructure or alternatively do some sort of logic to emulate this ourselves, but the easier and more effective solution is to use the java library here.

Synchronization regarding the `factors` list of lists is done by first sequentially instantiating empty lists at the start of the algorithm, and then at the start of each number factorization we index into the list of empty lists to get that numbers factors list, and then update that list while iterating.

Oblig says that the algorithm must parallelize each factorization, but I do not see a way of doing this that would be faster than the sequential version as this seems like an inherently sequential problem and since we are already utilizing all our threads to factorize other numbers this will not give a performance improvement.

## Testing

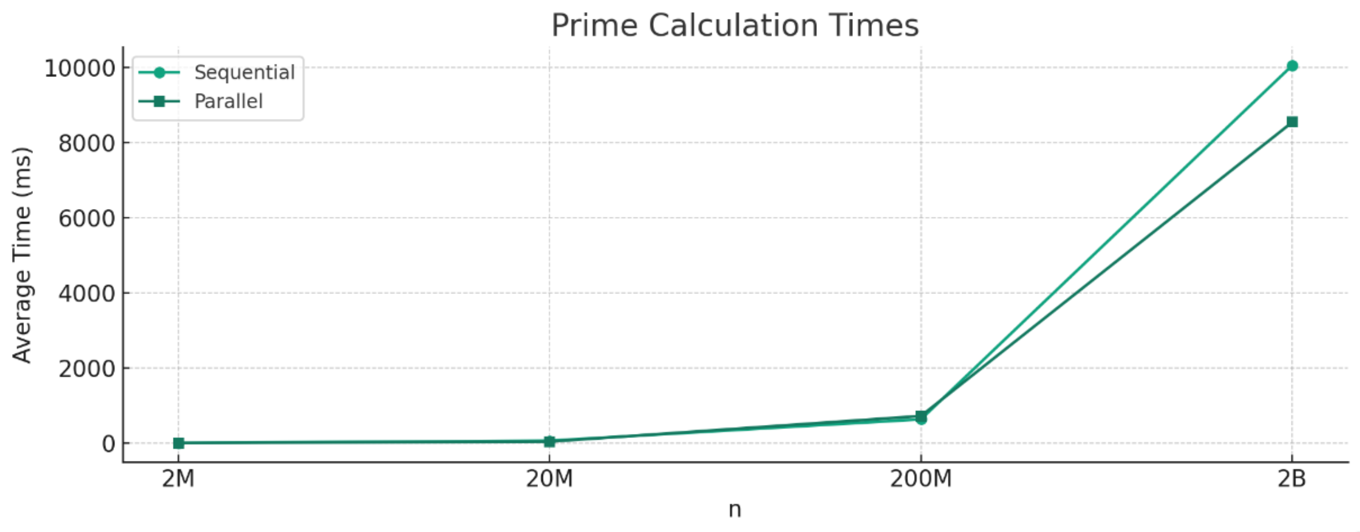
While first making the classes testing was primarily done by running that class' main method with the different parameters specified in the oblig text. After the first implementations of each of the classes i made the TestAll class which i from then on used to verify the outputs of the algorithms and also to compare runtimes when making changes. I also continuously checked that the outputs of the factorization had not changed by comparing the new Factors_... files with the old ones by using the linux `diff` command.

## Timings

In the graphs below we can see the runtimes for the different algorithms matching the times in the appendix below. All runs were on an AMD Ryzen 5 3600x, 6 cores 12 threads @3.8GHz, with the same command line arguments as in the appendix.

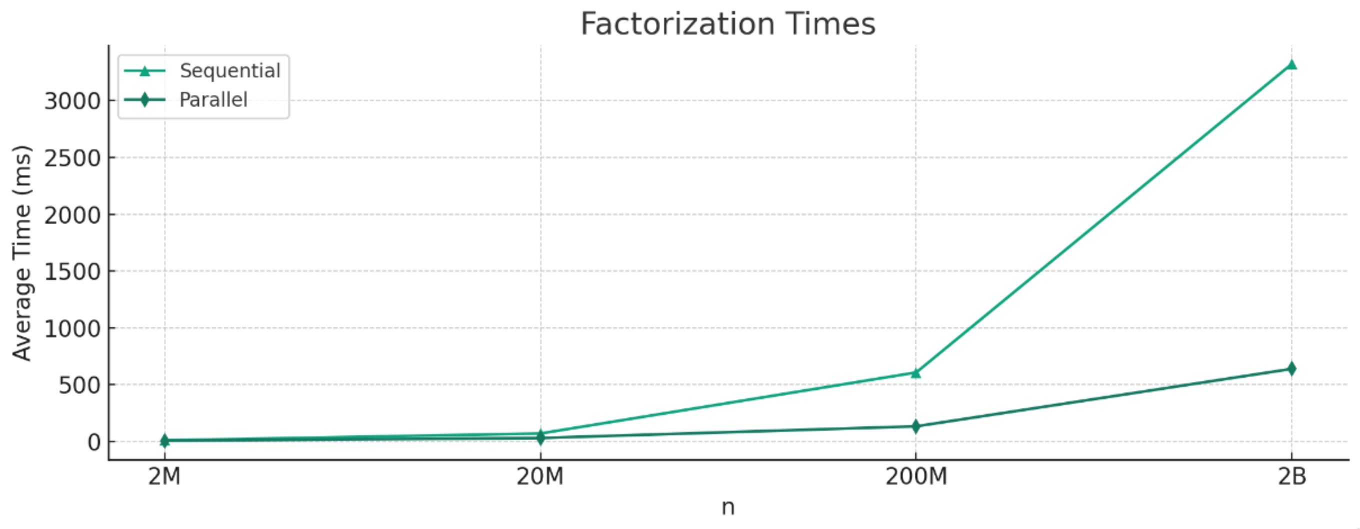
NOTE: primes calculated are for n * 2, not n * n. See comment inside TestAll.java for more explanation.

| n | Sequential (ms) | Parallel (ms) |
|---|---|---|
| 2M | 6 | 4 |
| 20M | 64 | 41 |
| 200M | 634 | 722 |
| 2B | 10046 | 8536 |



The parallel version of the sieve compared to the sequential one is not much faster, even for larger values for n. I mentioned above that a significant portion of the parallel algorithm was spent doing sequential operations (collectPrimes), but for it to be this slow there may be something other than that and general threading overhead that slows it down.

| n | Sequential (ms) | Parallel (ms) |
|---|---|---|
| 2M | 9 | 6 |
| 20M | 70 | 30 |
| 200M | 607 | 133 |
| 2B | 3323 | 639 |



The parallel version of the factorization algorithm is quite a lot faster than the sequential, the rate of which it is faster than the sequential also grows with n (proven by 6/9 > 30/70 > 133/607 > 639/3323). This makes sense as this problem is relatively easy to parallelize.

## Appendix

Following are the terminal outputs of running the program with the different n values specified in the oblig and the other fields left blank. The numbers are also added in a more readable format on the lines above each terminal input.

```
$ javac *.java

# n = 2 000 000
$ java TestAll 2000000
Average time to find primes sequential:       6ms
Average time to find primes parallel:         4ms
Average time to calculate factors sequential: 9ms
Average time to calculate factors parallel:   6ms

# n = 20 000 000
$ java TestAll 20000000
Average time to find primes sequential:       64ms
Average time to find primes parallel:         41ms
Average time to calculate factors sequential: 70ms
Average time to calculate factors parallel:   30ms

# n = 200 000 000
$ java TestAll 200000000
Average time to find primes sequential:       634ms
Average time to find primes parallel:         722ms
Average time to calculate factors sequential: 607ms
Average time to calculate factors parallel:   133ms

# n = 2 000 000 000
$ java TestAll 2000000000
Average time to find primes sequential:       10046ms
Average time to find primes parallel:         8536ms
Average time to calculate factors sequential: 3323ms
Average time to calculate factors parallel:   639ms
```