Oblig 3 IN3030 – Report

1. Introduction

In this report I will explain my solution to the parallel version of Sieve of Eratosthenes in addition to the parallel factorization of a large number. Then I will discuss how I did the implementation, in addition to measurements, including tables and graphs of runtime and speedup for different values of N.

2. User guide

Compilation:

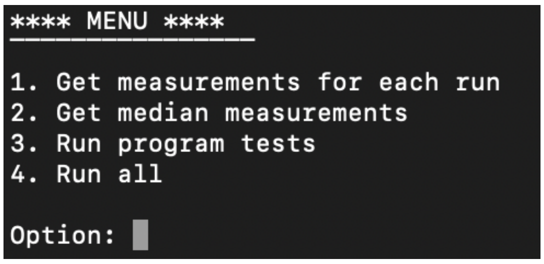
- Javac *.java

Run program:

- Java Main <n> <threads>
- E.g `java Main 200000000 7`

If you choose 0 threads the program will automatically use the number of cores on the machine as a starting point for creating the threads.

The user will then be presented with a menu and 4 options:



```
**** MENU ****
--------------

1. Get measurements for each run
2. Get median measurements
3. Run program tests
4. Run all

Option: 
```

1. Get measurements for each run

- For option 1 the user will be asked to choose number of runs for each algorithm.
  `Choose number of runs:`
- The program will then print measurements, including runtime and speedup for all algorithms.

2. Get median measurements

- Program uses a default number of runs and measure runtime and speedup for each run.
- The program then calculates and prints median runtime and speedup for each algorithm.

3. Run program tests

- Run program tests for each algorithm.

4. Run all

- Run full program, both measurements for each run, median measurements and tests.

3. Parallel Sieve of Eratosthenes

The parallel version of Sieve of Eratosthenes generates alle prime numbers up to N. The problem is solved in the ParallelSoE class. The constructor takes two arguments:

- An integer N used to determine the limit of primes to be generated.
- An integer k used to determine the number of threads to use

First the sieve is performed and all prime numbers up to √n are generated.
Each thread is then given a segment of the primes and then cross out alle multiples of the primes, generating alle prime numbers up to N. The total number of primes is then counted and the final array 'primes' is initialized. The threads are now done and the getPrimes() method sequentially collects the prime numbers from a two dimensional array to a single 'primes' array and returns it.

4. Parallel factorization of a large number

The parallel version of factorization of a large number calculates the factorization of the 100 largest numbers less than N x N. The problem is solved in the ParallelFactorize class. The constructor takes three arguments:

- An array of primes[] generated by sieve.
- An integer N used for getting the numbers (N x N) to factorize.
- An integer k used to determine the number of threads to use.

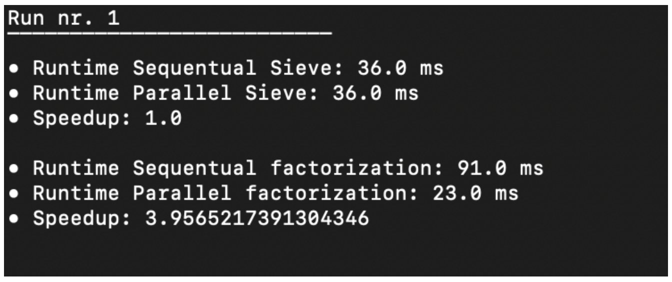
The factorize function will then create k number of threads for doing the parallelization. Each thread will then partly factorize the prime number on each index equal to id of thread + k up to length of the array. The factors are then stored in a local ArrayList before they eventually are gathered together. The factorize method uses a Cyclic Barrier to wait for the threads to synchronize and then returns the array with the factorized numbers.

5. Implementation

Sieve of Eratostenes generate all prime numbers up to n, which is decided by input from user. The factorization algorithm will then factorize the 100 larges number less then n * n using the prime numbers from sieve. The program includes both a sequential and parallel version of each algorithm. As presented in the user guide the program has 4 options presented to the user when running the program.

1. Get measurements for each run:

The first option gives the user the opportunity to choose number of runs for timing the algorithms. The program has a function timeMeasurements() which then runs the number of runs chosen by the user. The function will for each run store the runtime data in a separate array for each algorithm. When finished the function will then call the function getMeasurements() with the arrays as arguments, and then print out the measurements for each run. Belove you can see an example output for 2 runs.



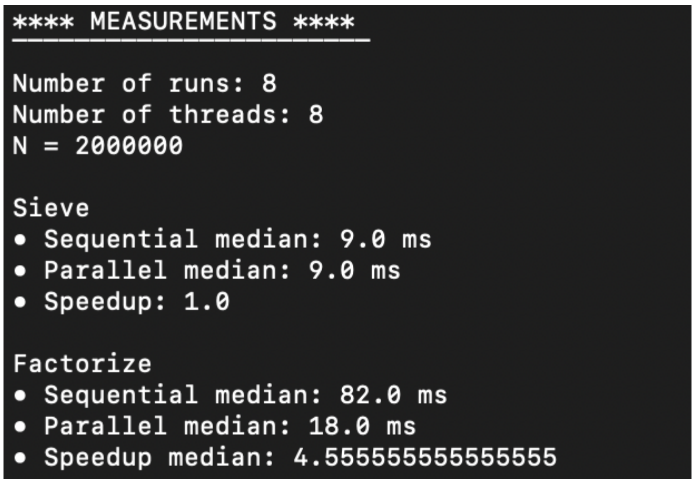
```
Run nr. 1
─────────────────────────

• Runtime Sequentual Sieve: 36.0 ms
• Runtime Parallel Sieve: 36.0 ms
• Speedup: 1.0

• Runtime Sequentual factorization: 91.0 ms
• Runtime Parallel factorization: 23.0 ms
• Speedup: 3.9565217391304346
```

2. Get median measurements:

This option gets median measurements for the algorithms. The option uses a default number of runs, in this case 8, to measure the runtime for each algorithm. It also uses the method timeMeasurements() to time each run. It then calls the function getMedianMeasurents() with the arrays as arguments. It then sorts each array and prints out the median measurements. Example output:



```
**** MEASUREMENTS ****
----------------------

Number of runs: 8
Number of threads: 8
N = 2000000

Sieve
• Sequential median: 9.0 ms
• Parallel median: 9.0 ms
• Speedup: 1.0

Factorize
• Sequential median: 82.0 ms
• Parallel median: 18.0 ms
• Speedup median: 4.555555555555555
```

3. Testing:

Calls the function testProgram() which then calls all the test-methods which is implemented in the following functions:

testSieveOutput();
- Test sieve output for all prime numbers less than 100. The test is implemented with an array including all the prime numbers less than 100 and will checks that all the numbers generated from the sieve less than 100 equals to the numbers in the array.

testSeqAndParaSieve();
- Test that sequential and parallel sieve produce the same results. Check all indexes in the arrays created by the sieve and compare each element.

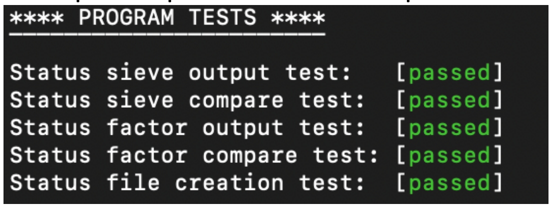
testFactorization();
- Test that factorization produce correct output.

testSeqAndParaFactorization();
- Test that sequential and parallel factorization produce the same results.

writeFactorization(n);
- Write factorization using Oblig3Precode method writeFactors().

The program will run all tests and give feedback to the user if test is passed or failed.

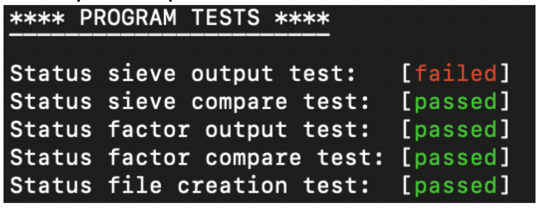
Example output when all tests passed.



```
**** PROGRAM TESTS ****

Status sieve output test:   [passed]
Status sieve compare test:  [passed]
Status factor output test:  [passed]
Status factor compare test: [passed]
Status file creation test:  [passed]
```

Example output when one test failed:



```
**** PROGRAM TESTS ****

Status sieve output test:   [failed]
Status sieve compare test:  [passed]
Status factor output test:  [passed]
Status factor compare test: [passed]
Status file creation test:  [passed]
```

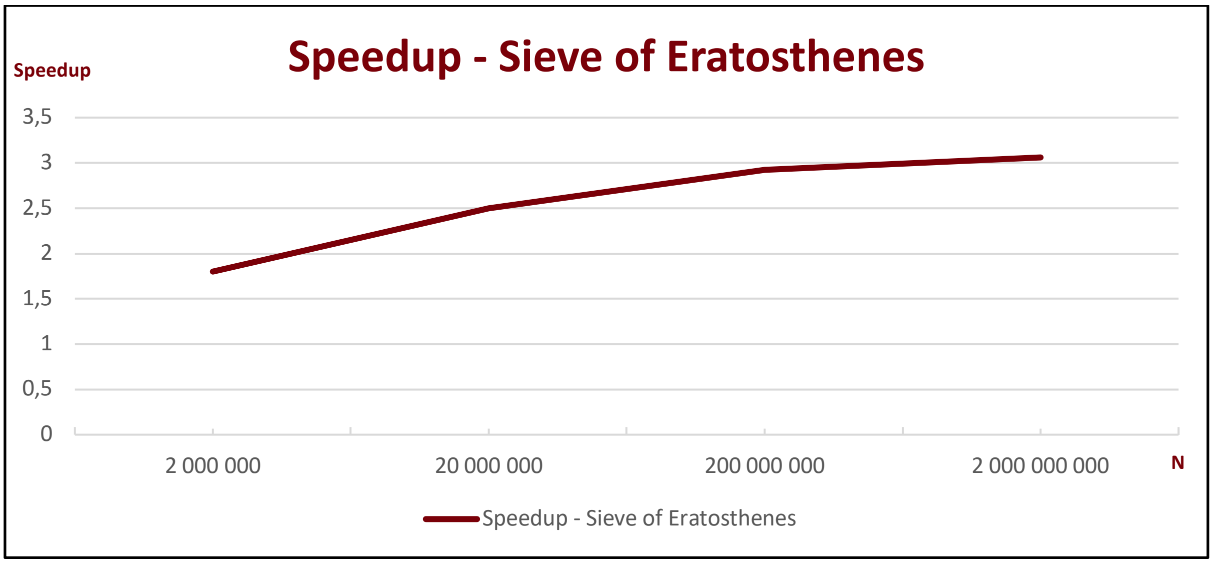
4. Run all:
Run the whole program in one operation, including measurements of all runs, median measurements and tests. The user will not be able to decide number of runs and it will use the default number of runs and print measurements of each run and median measurements.

6. Measurements

Speedup Sieve of Eratosthenes

N	Sequential runtime (ms.)	Parallel runtime (ms.)*	Speedup
2 000 000	9,0	5,0	1,8
20 000 000	93,0	37,0	2,5
200 000 000	1106,0	379,0	2,92
2 000 000 000	13314,0	4339,0	3,06

***Number of cores used in parallel version: 8**

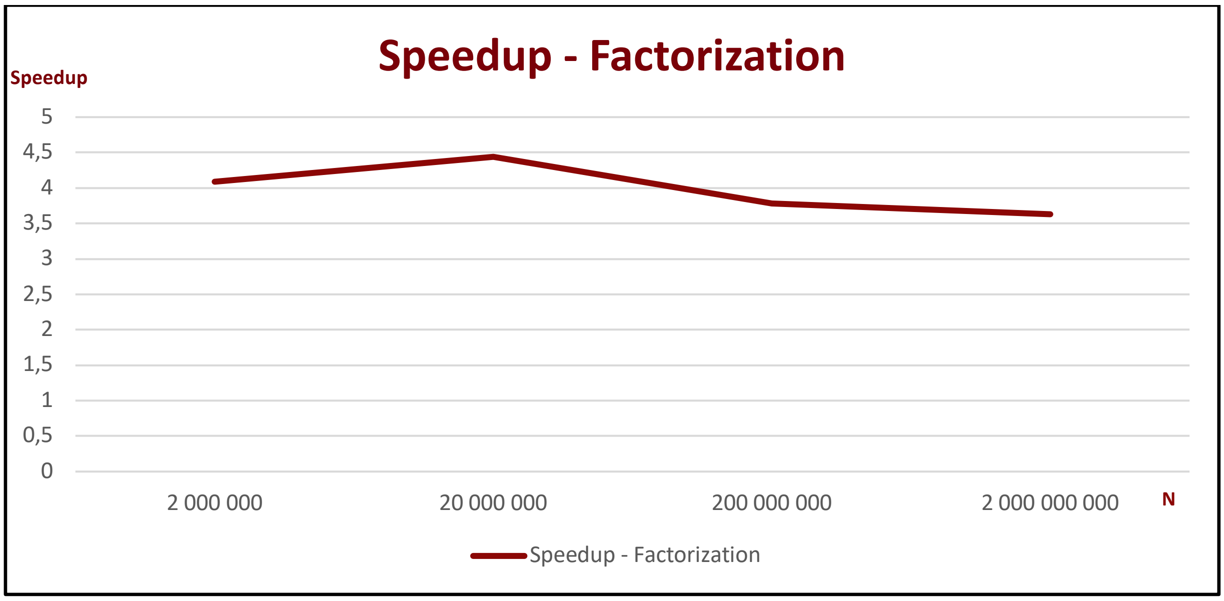


From the graph we can observe a steady increase in speedup as N gets bigger, but that the curve flattens out a bit when N gets very large.

Speedup Factorization

N	Sequential runtime (ms.)	Parallel runtime (ms.)*	Speedup
2 000 000	86,0	21,0	4,09
20 000 000	782,0	176,0	4,44
200 000 000	5812,0	1536,0	3,78
2 000 000 000	52482,0	14422,0	3,63

***Number of cores used in parallel version: 8**



The graph shows speedup for the factorization, comparing the sequential and parallel version. We can observe a relative stable speedup, which seems to decrease a small amount after N exceeds 20 000 000.

7. Conclusion

For the sieve we can see a steady increase in speedup as values of N gets bigger, at the same time as the relative increase in speedup decreases for very large values of N. For the factorization we can see a good speedup for all values of N, with a slightly decrease in speedup after N exceeds 20 000 000. In both cases I find it difficult to say exactly what causes the decrease in speedup when N becomes larger. One of the reasons could be related to the large values of N which the threads are operating with, and that it can take time for them to synchronize. It could also be a result of the sequential parts of the parallel algorithms, especially the sieve, where large values of N could make these parts less efficient.