1) This report is about how I implemented a parallelization of the Sieve of Erethosthenes and factorization to primes, as well as how the implementation performed.

2) You run the program with: “java Oblig3Main <n> <c>” where <n> is a positive integer, and where <c> is the number of cores you want to use.

   Example: “java Oblig3Main 1000 4”

   My processor is an Intel Core i7 with 4 cores, 8 threads, and 2 Ghz clock frequency.

3) I parallelized the Sieve by:
   a. Sequentially finding the primes up to sqrt(N).
   b. Divide the byte-array into <c> partitions, where <c> is the number of threads available.
   c. Giving each thread 1 partitions, which includes roughly 1/<c> of the bytes.
   d. Running the Sieve on the partition for each of the primes up to sqrt(N).

4) I parallelized the Factorization by:
   a. Dividing the primes into <c> partitions, where <c> is the number of threads available.
   b. Giving each thread a partition of the primes, and having each thread check if N can be divided by their assigned primes. If they could, they would be added to a shared Array of ArrayLists containing all the factors.
   c. Going through each ArrayList in the shared Array, dividing N by each factor, as many times they can be divided, and putting the components into a new Arraylist, containing all instances of a component being part of the answer.
   d. If N is not 1 when the last component from the array is accounted for, that means N is a prime-number, and N gets added to the new ArrayList of all the components.

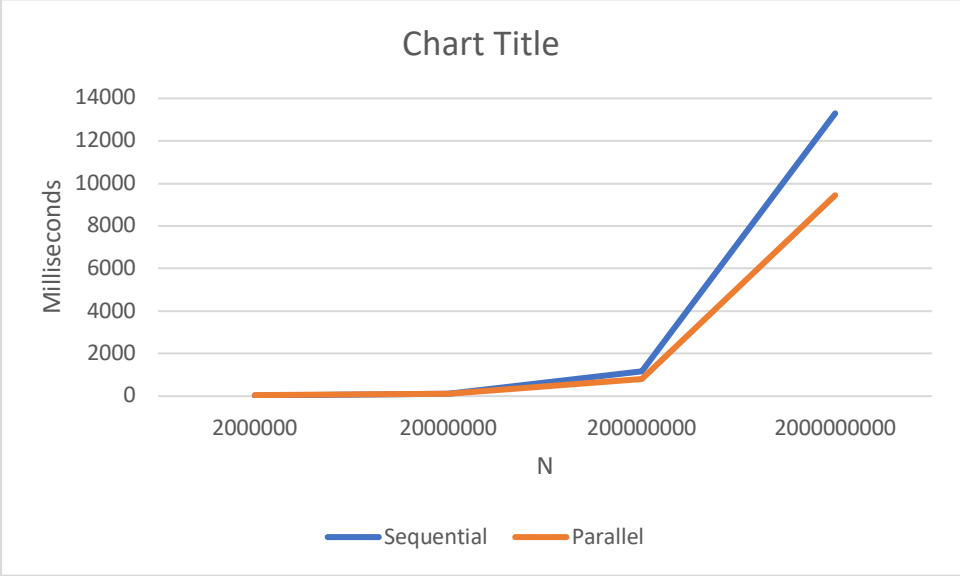
5) My program:
   a. Checks if there are given any arguments, and decides n and c by those arguments.
   b. Runs and times the sequential and parallel implementations of the Sieve of Erithosthenes, and checks if they return the same answer.
   c. Runs and times the sequential and parallel implementations of the Factorization, and checks if they return the same answer.
   d. Runs the factorization for the 100 highest numbers less than N*N, adds those factors into the precode-class, and writes the results to a file.

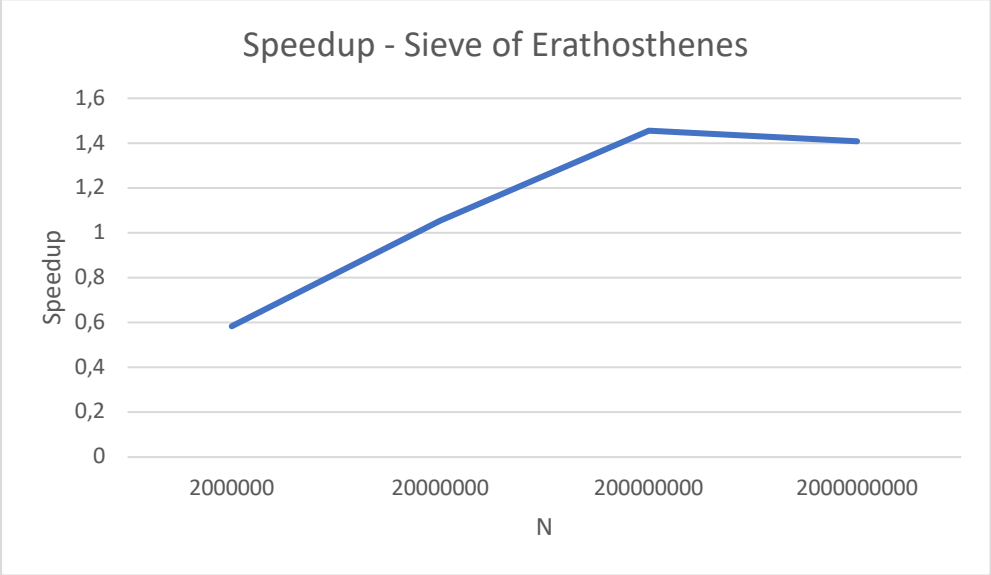
6) Sieve of Erithosthenes:

   The measurements were made by using System.nanoTime(). The measurements are in milliseconds

	Sequential	Parallel	Speedup
N	ms	ms	

2000000	28	48	0,58333333
20000000	119	113	1,05309735
200000000	1163	799	1,45556946
2000000000	13296	9445	1,40772896





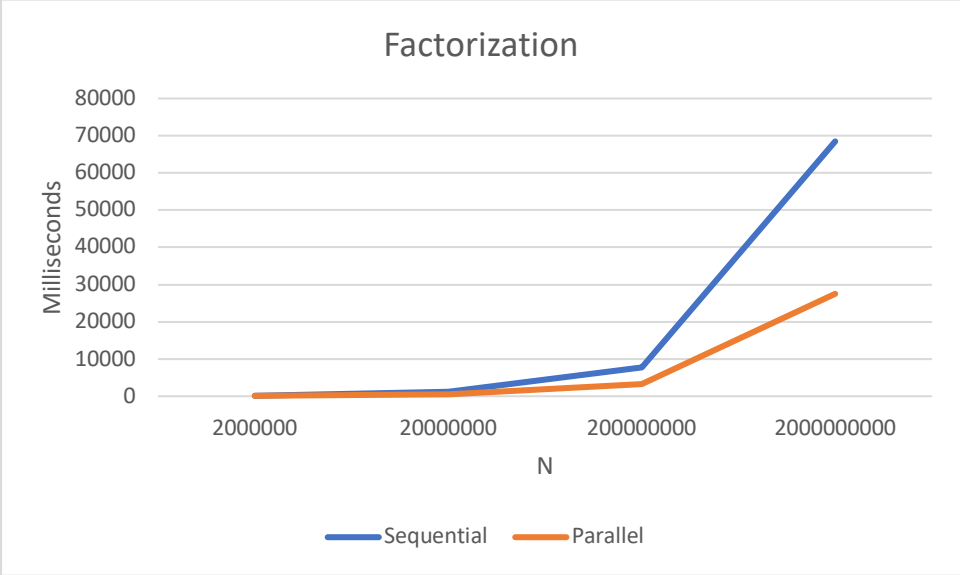
The Sieve gets a speedup > 1 right around N = 200 000 000.

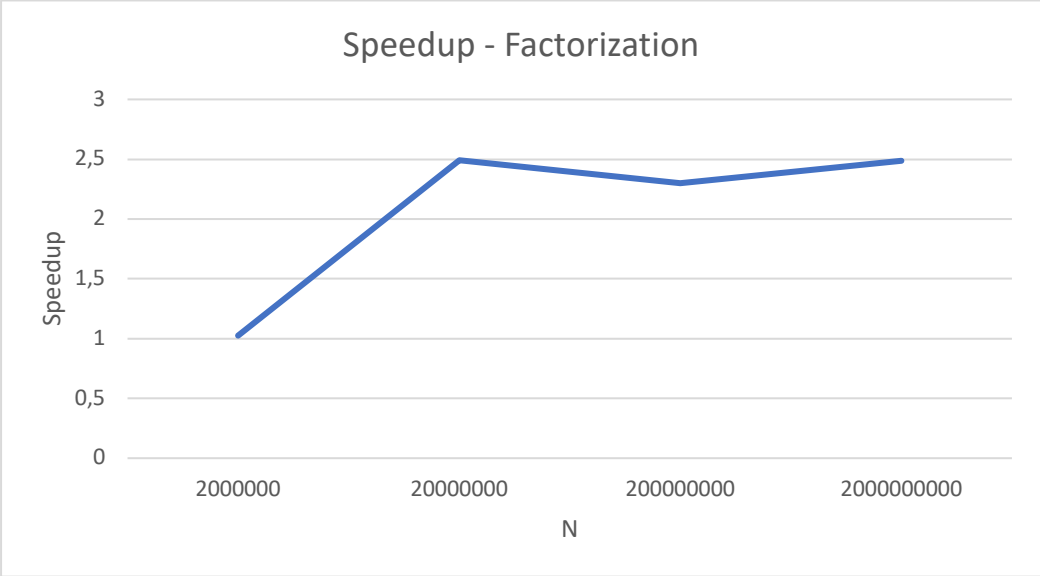
I think the speedup gets noticeable around there because the cost of starting new threads is too high for lower numbers.

Factorization:

The measurements were made by using System.nanoTime() on 100 factorizations each. The measurements are in milliseconds.

	Factorization		
	Sequential	Parallel	Speedup
N	ms	ms	
2000000	124	121	1,02479339
20000000	1131	454	2,49118943
200000000	7685	3342	2,29952124
2000000000	68474	27527	2,48752134





The Factorization gets a speedup of >1 right about N = 2 000 000.
There is still a speed-down if I run factorization of a single number, but it gets significantly faster when factorizing 100 numbers, since it allows for the JVM compile the bytecode to machine code instead, lowering the cost of starting the new threads.

7) I have managed to implement a quite efficient parallelization of the Sieve of Erithosthenes, and Factorization of multiple, large numbers.

8) The program outputs:
    a. Nano-second duration of Sequential Sieve
    b. Nano-second duration of Parallel Sieve
    c. Whether the Sieves return the same answer.
    d. Nano-second duration of Sequential Factorization
    e. Nano-second duration of Parallel Factorization
    f. Whether the Factorizations return the same answer.