1. INTRODUCTION

1.1. This program uses the Erastothenes Sieve algorithm to generate all prime numbers from 0 to N.

1.2. After all primes are generated, the 100 largest numbers (max N*N) are factorized.

1.3. The program does this twice: First sequential, then in parallel. The objective is to create a parallel solution which yields a speedup > 1x the sequential solution

1.4. All factorizations created by the parallel solution are printed into a file.

2. USER GUIDE

2.1. Run this program by using the command:
java Oblig3 <N> <K>

N is the maximum prime to be generated
K is the amount of threads to be used in the parallel solution
Set K=0 to automatically choose all threads on your processor

The global variable LOOPS can be changed to whatever number you see fit. By default it is set to 7. This is the amount of times the program is run.

Note: For N = 2 billion, additional heap space is needed. Run with the -Xmx4096m flag.

3. PARALLEL SIEVE OF ERASTOTHENES

3.1. The sieve is split into 3 parts. First we find the initial √N primes, so that we can mark all numbers from 0 to N. Then we count all non marked numbers. Finally we store these numbers in a primes[] array.

3.2. FIND √N PRIMES

3.2.1. Before starting, we need to first find the √√N primes sequentially. Doing this will make initializing start values for every thread easier and allow us to use synchronization a lot less.

3.2.2. Now that all √√N primes have been found, we start every thread with current prime = findInitials(). This method will assign every thread with its own prime. Every thread will mark all the values of the initial primes, and then find the next *num_threads* primes before continuing marking.

3.3. COUNT PRIMES

3.3.1. In this part we need to count all the bits that have not been marked with a 0. As we've already counted the first √N primes, we need to start from √N and count to N. We split this gap into *num_threads* parts. This gap should be of size *(N - √N)/num_threads*. Then every thread has a *low*, starting from *√N + 1 + gap*id*, and a *high* going to *√N + gap*(id+1)*. The final thread has a *high = N.*

3.3.2. From here, we just need to count all zeros locally. Once we pass the limit, we add the localCounter to the global primesCounter, by using locks.

3.4. GATHER PRIMES

3.4.1. The final part of the sieve is creating the primes[] array. By using primesCounter, thread 0 initializes primes[] to this size.

3.4.2. Every thread creates a local copy of this array.

3.4.3. Using a slightly modified version of flip, every thread loops the whole byteArrray[], and finds all the zeros in every bit. The bit is then calculated into the respecting value, and then stored into the local array.

3.4.4. Every value in the local array is then copied into the global primes[] array, using locks.

4. PARALLEL FACTORIZATION OF A LARGE NUMBER

4.1. We are only interested in the 100 largest numbers that we can find, using all primes from 0 to N. These numbers range from N*N-101 to N*N-1.

4.2. All these numbers are stored in two long arrays of size 100. One of these arrays are operated on, such that we know which number is the original and which is the remaining number. If the remaining number is not 1, then it must be a prime.

4.3. In this solution we make use of the primes[] created by the sieve. Every thread is given a piece of this array, where every thread starts on primes[thread_id] and jumps num_threads for every loop.

4.4. We have a match when our current % primes[i] == 0. The current number is divided by primes[i]. Every thread has to update the global array as well, so in order to prevent data loss, a lock is used.

4.5. Once no more matches is found, we need to loop the remaining numbers array. If the remaining number is not 1, we add this number to the factors. Finally we print the results.

5. IMPLEMENTATION

5.1. The program is split into three files. The main file is Oblig3.java. This controls the amount of times the program is run, checks for correct input, and generate a new instance of the parallel and sequential files for every loop. All runs are timed and printed out.

5.2. The SequentialSieve.java is responsible for generating all the prime numbers from 0 to N. This file uses mostly code from the TA sessions, uploaded by Magnus Espeland.

I added the factorization algorithms, which uses the primes[] generated by the sieve to factorize the 100 largest numbers, where N*N is max.

5.3. The ParallelISieve.java works just like SequentialSieve.java, except it uses multiple threads. Almost all the methods from the Sequential file has been paralleled. The output is created as expected.

6. MEASUREMENTS

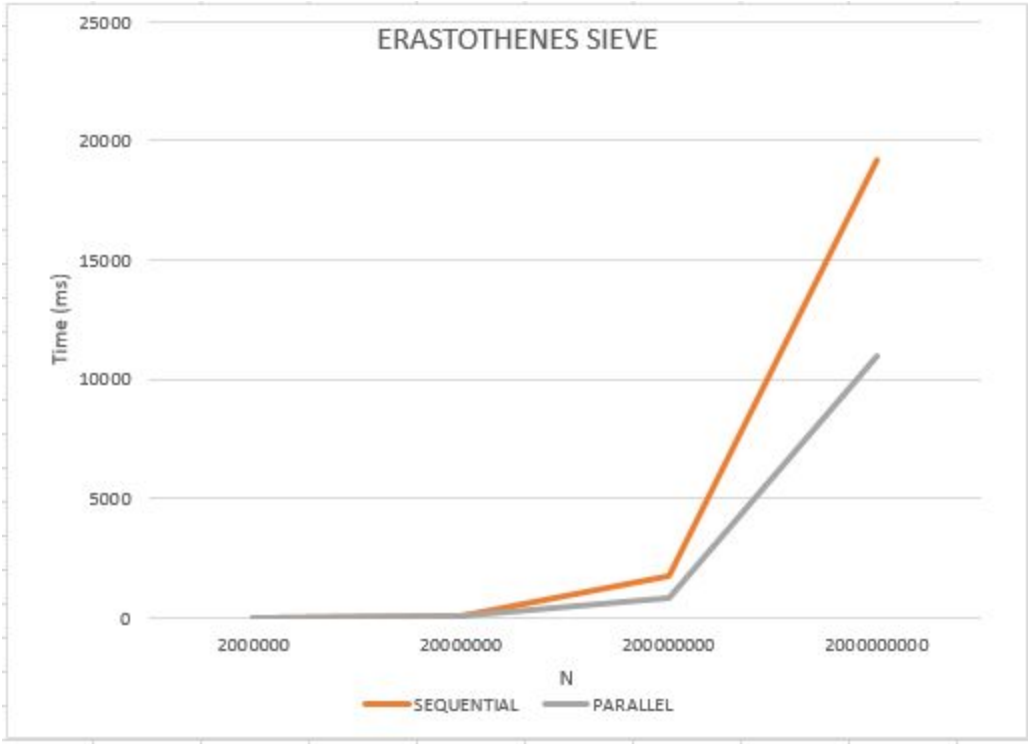
6.1. All measurements are used on a Intel Core i5-4670K @ 3.40 GHz (4 cores) CPU, running Windows 10 64 bit.

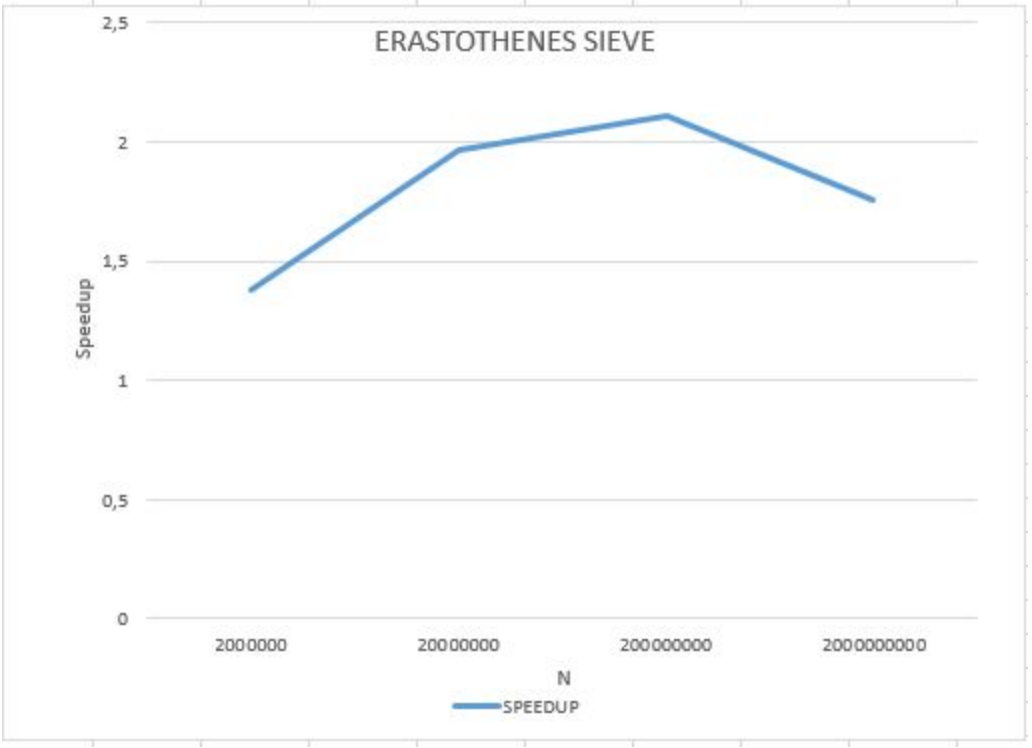
6.2. Runtime values showing the median result after 7 runs:

SIEVE				
N	2000000	20000000	200000000	2000000000
SEQUENTIAL	11	114	1709	19181
PARALLEL	8	58	810	10935
SPEEDUP	1,375	1,966	2,110	1,754

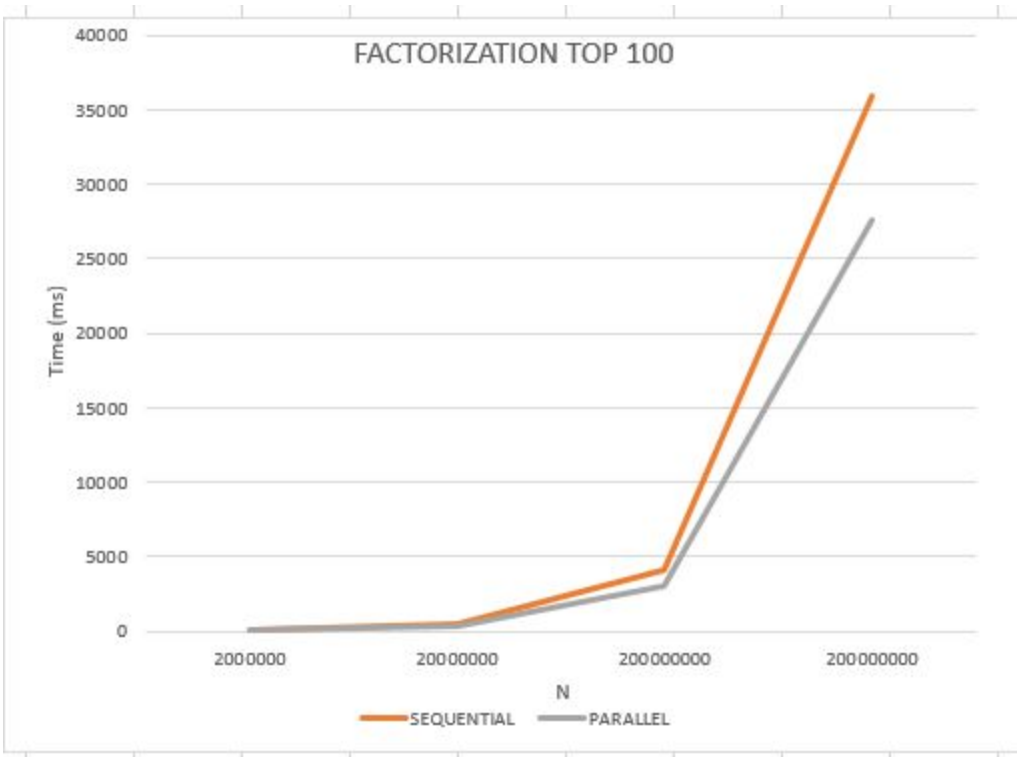
TOP 100 FACTORIZATION				
N	2000000	20000000	200000000	200000000
SEQUENTIAL	53	455	4178	35874
PARALLEL	58	349	3076	27607
SPEEDUP	0,914	1,304	1,358	1,299

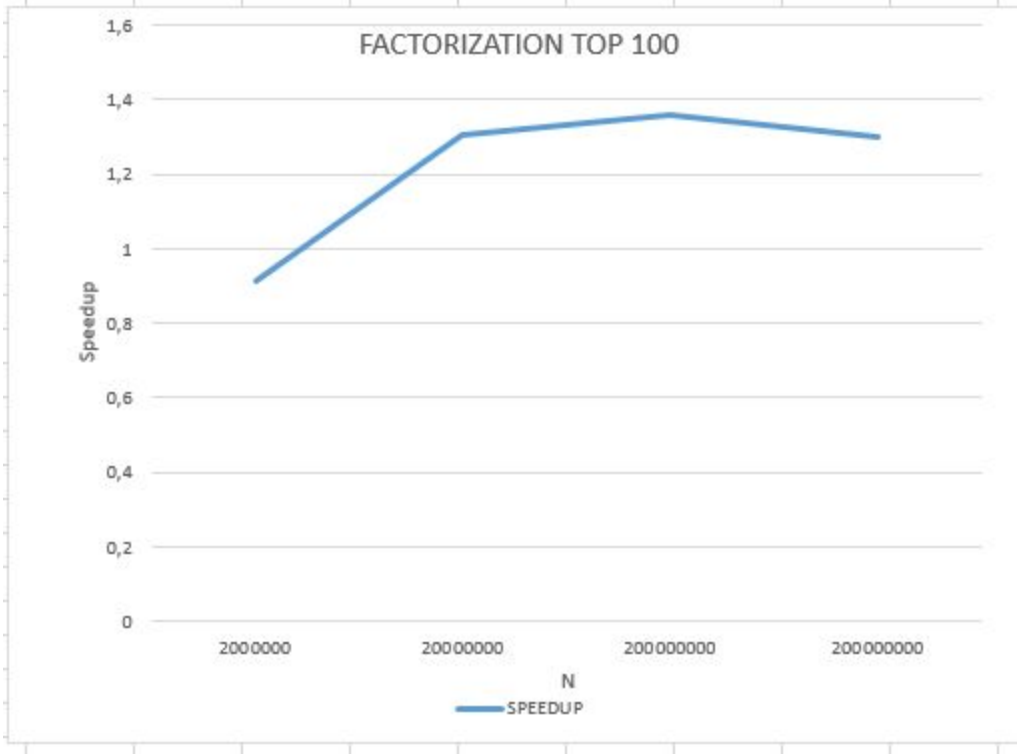
6.3. Sieve runtimes values as a graph:





6.4. Top 100 factorizations runtime values as a graph:





7. CONCLUSION

7.1. The program works as expected and doesn't seem to crash.

7.2. The parallel sieve results are compared with the sequential sieve after all primes are generated. No error occurs.

7.3. Our program works with an acceptable speedup > 1.

8. APPENDIX

8.1. The program is run LOOPS amount of times (default: 7). Every run creates an output like this:

Run i
Erastothenes Sieve: SEQUENTIAL: <a> ms PARALLEL: <b> ms SPEEDUP: <a/b> ms
100 Factorizations: SEQUENTIAL: <x> ms PARALLEL: <y> ms SPEEDUP: <x/y> ms