IN4330 mandatory 3

Erlend Kristensen

27.03.2022

1 Introduction

This report is about prime numbers and factorization. For finding prime numbers we are using the Sieve of Eratosthenes algorithm, which quickly described it starts off by saying every number is a prime, and for every prime number marks all numbers which is a product of this prime number as a non prime, so we end up with a list of only prime numbers. We will parallelize both the sieve algorithm and the factorization algorithm to get speedups, and then test the times both manually and with maven benchmarking. For all testing purposes of this report, the code was run on a laptop with a AMD Ryzen 5 5500U cpu with 6 cores and a base speed of 2.1 GHz.

2 User guide

User guide can also be found in readme.txt.

To compile:
javac Oblig3.java; java Oblig3 $<n><k>$
where n is size of array, and k is number of cores you want to use.

Can also manually test Sieve time by compiling:

javac SievePara.java; java SievePara $< n >< k >$

for benchmarking:

go into folder "myBenchmark" and run: java -jar target/benchmarks.jar

if it does not work, first run: mvn clean install, then run the command above.

3 Sieve of eratosthenes

When parallelizing sieve, we have to keep in mind the overhead required. I chose to parallelize the traversing, so for each prime number, each thread has different values that they mark out as non prime. This will not give a great speedup, since we only parallelize the innermost for loop, but its a clean implementation. I also changed from bytes to integer list, since i might get overhead with different threads updating same byte index (happened for larger n values). with integers, I initialize the list to only contain enough indexes for each odd number, so i had to change up the traversing slightly (as can be seen in SievePara.java).

4 Factorization

For factorization, we only need prime numbers up to the square root of the number we want to find. Therefore, we use sieve to find the prime numbers up to the numbers square root, then we use these primes to factorize. Factorization is a lot simpler to parallelize than sieve. The sequential algorithm

works by checking if a number is divisible by the prime numbers we have found, and if they are we divide our input number with this number and add the prime number to a list. We keep checking primes and dividing until certain conditions are met. Either we divide until our input number is 1, or we try to divide until we reach the end of our list of prime numbers. If we reach the end, then we know that the number we are left with is a prime as well, and add it to the list.
For parallel we simply do the same, but each thread checks different primes, but they divide a shared variable until it is no longer divisible as stated above.
The implementations can be found in Factorize.java and FactorizePara.java. The resulting factorizations are found in their respective text files.

5 Implementation

Our program starts by taking input variables from the user, which is n (number which we want to find all primes up to) and k, number of cores we want to use. If k is ¡= 0, we use all cores available on the computer.
Then, we find the primes using both the sequential and parallel sieve implementation, and print out the times used for these. Further on we use these to factorize using both the sequential and parallel factorization methods. The numbers we want to factorize are the 100 largest numbers bellow n^2, so $n^2 - 1, n^2 - 2, ...$, then we print out the times used for both methods. We do all these calculations 7 different times and store the times in different lists. We then sort them and print out the median time to get a more accurate representation of the time needed for the sequential and parallel implementations.

6 Measurements

These are the median times i got for sieve after 7 runs:

time in seconds my own laptop (6 cores)		
Size of array (n)	Sieve sequential	Sieve parallel
2 000 000	0.0113883	0.0096715
20 000 000	0.117093	0.1213761
200 000 000	1.4150084	1.3005843
2 000 000 000	15.1008787	13.2315276

We get a clear speedup here, slightly above 1, which is expected for a "simple" implementation like this.

These are the times we got for sequential and parallel Factorization:

time in seconds my own laptop (6 cores)		
Size of value (N)	Factorization sequential	Factorization parallel
2000000^2	0.1050634	0.1383578
20000000^2	1.3399365	0.8238873
200000000^2	4.0494827	2.631601
2000000000^2	36.6587431	16.2858921

These speedups are great, with over twice the speedup for $N = 2000000000^2$.

Here are the benchmarking tests:

```
Benchmark Mode Cnt Score Error Units
MyBenchmark.testFactorize thrpt 3 0,010 ± 0,001 ops/s
MyBenchmark.testFactorizePara thrpt 3 0,018 ± 0,004 ops/s
MyBenchmark.testSieve thrpt 3 0,043 ± 0,004 ops/s
MyBenchmark.testSievePar thrpt 3 0,051 ± 0,003 ops/s
```

For benchmarking i used the same amount of tests as with the main code, so 100 different values for factorization. I used $n = 2000000000$ (2 billion). We see that the factorization scores almost twice as high for parallel than sequential, and sieve scores 18% higher, which indeed is a speedup and is similar to the speedup of our printed times.

7 Conclusion

To conclude, we managed to get a speedup on an already fast sequential algorithm (sieve), and the factorization also ran fast, especially with parallelization.