3RD MANDATORY ASSIGNMENT

IN3030 Efficient Parallel Programming

Author: Pooria Mehregan
Username: pooriam

0. Introduction – *what this report is about*

In this report I am going to explain how I:

- parallelized the Sieve of Eratosthenes
- parallelized prime factorization of a big number N
- developed a sequential factorization of N

I am going to explain what strategies I used, and how did I implement these algorithms. I will also present outputs and graphs to visualize my results and to compare what speed-ups can be achieved by a parallel solution to both Sieve and Factorization of a big number N.

1. User guide – *how to run*

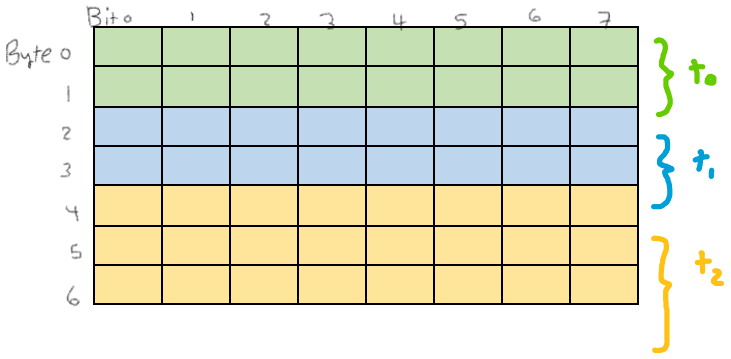
Make sure you are in the source directory and execute the command below at your terminal/shell:

- Option 1: User defined inputs where n is the number you want to factorize (and find primes less than it), and k is number of threads to be used. Set k to 0 if you want to use number of cores on your machine as number of threads:
  `javac MainClass.java && java -Xmx8096m MainClass <n> <k>`

- Option 2: an automated option to test and time the sieve for values:
  2 million, 20 million, 200 million and 2 billion. The result of factorization is then printed to a file called *Factors_<N>.txt*, and timing results are printed to a file called *timings.txt*.
  `javac MainClass.java && java -Xmx8096m MainClass`

2. Parallel Sieve of Eratosthenes – *how I did the parallelization*

The strategy I used to parallelize the Sieve is to divide the bytes between threads, in a way that each thread gets to work on bytes n to m. In the drawing below thread 0 works on bytes 0 and 1, thread 1 works on byte 2 and 3, and thread 2 works on bytes 4, 5, and 6.



NOTE: Be aware that parallel sieve does not return a sorted list, and it needs to be sorted if it's going to be used with parallel factorization.

3. **Parallel factorization of a large number – *strategy I used***

Strategy:

1. Gathering all primes less than sqrt(N) into array *primes*, by using class Parallel Sieve.
2. Let each thread get 2 primes, 1 from start of the array, and 1 from end of the array. Why:
    a. To avoid having threads that get lesser primes do much more work than threads which get bigger primes.
    b. Getting 2 primes at a time and not only 1, reduces the synchronization time, they need to access the method which deals the primes to them, in half.
3. After each thread founds factors for the prime it was given, it writes them to a global ArrayList<Long> called factors.

4. **Implementation - a somewhat detailed description of how the Java program works & how tested**

Parallel Sieve:

I parallelized the sieve in a separate class called ParallelSieve. The constructor of this class produces a prime table for primes less than sqrt(n). Then other threads go on to use this table for crossing off none primes in their own part of the byte array.

This class has a work() method, which creates and starts Worker instances. Each Worker represents a thread that works on the specific parts of the global byte array called odds. When workers are finished, each try to write their primes to a global ArrayList *allPrimes* by calling writeToGlobalArray(). This method uses ReentrantLock to ensure only one thread enters at a time.

Sequential Factorization:

The sequential factorization is located in the class SieveOfEratosthenes. This method, which is called *factorize(),* factorizes the 100 largest numbers less than N * N (exclusive: not including the N * N).

The strategy it uses is going through primes less than sqrt(m), and try to divide the number m on prime p.

- If m_n is not divisible by p, it tries the next prime until there are no more primes or prime is greater than sqrt(m).
- if divisible then it divides m on p, and lets the next divisor be the next prime and dividend to be rest of the division.
- If divisor becomes greater than sqrt(m) then method returns the current factors.
- Or when there are no primes and dividend is greater than 1, then it adds the dividend to factors, as the last factor, and returns.

Parallel Factorization

Class ParallelFactorization finds primes less than sqrt(N) in its constructor and determines the number of threads to be used. The work of this class is done by calling its public method work(). work() creates Worker classes and give each of them the number N. Each worker then asks for 2 primes from

dealNextPrimes(), and they get one prime from the beginning of the array and one from the end.

When the factorization is finished, workers adds their local factors to the global ArrayList<Long> factors. When all workers are done, the main thread multiplies all founded factors and divides N by their product to find the last remaining factor, and then returns the factors.

5. Measurements

Machine info:

- MacBook Pro (16-inch, 2019)
- 2,6 GHz 6-Core Intel Core i7 (12 with hyper-threading technology)
- 16 GB 2667 MHz DDR4

To see tables and graphs please see the pdf file called *SieveMeasurement.pdf* inside *materials* folder.

6. Conclusion

In this report, I tried to explain, comment on, and give visual measurements on how I parallelized the Sieve of Eratosthenes and factorization of a big number N.

The process may not look easy and requires an enormous amount of planning, but by looking at the measurements, one can see that much can be gained from parallelization of factorization of a big number like 2 billion.

7. Appendix – *the output of my program*

Output of the program is written to the file called *timings.txt* which can be found inside the *materials* folder.

Factorizations of each N can be found inside materials folder. They are called *Factors_N.txt* where N is replaced with the actual number. To achieve this level of organization I had to change the path in the Oblig3Precode (Line 98) a little.