Marked Lab: CPU-Intensive Application

*This lab must be prepared in teams of **1 to 2** students and submitted on Campus. Check the due date in the submission area.*

Contents

1 Work to Do

You have to develop a Java method that computes prime numbers as fast as possible by leveraging the processing power of all available CPUs. This method is named `computePrimes` and it is defined in the `PrimeComputer` class. You may add to this class any field, method or inner class you need to implement the `computePrimes` method.

The template of the `PrimeComputer` class and a complete test program are provided as a NetBeans[1] project. Please read carefully the Javadoc of the project before you start developing. You can find the HTML version of the Javadoc here: `dist/javadoc/index.html`.

If you have any question regarding the lab, please post it on the Questions & Answers forum. I will not reply to emails. Thank you.

2 Beware

I will test the `computePrimes` method using some test programs of my own. Therefore, **you must not alter** the interface of the method, including its `throws` clause.

I will test the `ComputePrimes` methods in batches. Therefore, you must ensure the method **releases all the resources it allocated** before it returns. This applies in particular to the threads, whether daemon or not, that your method created.

[1] you may use another IDE since the only deliverable is the PrimeComputer.java file

This lab is concerned with finding the best parallel algorithm to compute primes, not finding the best sequential algorithm. Therefore, for the sake of fairness, your code **must use the provided** `PrimeComputerTester.isPrime` method. Only two variations are allowed: (i) your code can optimize the selection of prime candidates, (ii) it can also reuse the computed primes to reduce the number of divisions needed.

3 Test Bed

I will use the **Java SE 8** platform to test your code. Please make sure it runs correctly on this version.

I will most probably run your code on the following hardware configuration: Core™ I7-8550U 2.4 GHz processor, 4 cores (i.e. 8 logical processors total), 16 GB RAM.

However, your code must not make any assumption about the number of available processors. Instead, it must retrieve this number by calling the `Runtime.availableProcessors` method.

I will test your method on very large [1 – max) ranges, typically [1 – 10,000,000): do not optimize your code for small `max` values. The elapsed time will be averaged over several iterations, typically 10, to mitigate noise and to allow for JVM warmup.

4 Directions

The key idea to get the best performance is to keep all the processors busy from the beginning to the end of the computation. To that end, you must split the workload evenly across the processors.

Some data-partition schemes are known to be inefficient. For example, assume you have to compute the primes in the range [1 - 10,000,000) on a two-processor computer. If you have processor #1 compute primes in the range [1 - 5,000,000) and processor #2 compute primes in range [5,000,000 - 10,000,000), then processor #1 will complete its task well before processor #2, resulting in processor #1 being under-used.

Moreover, keep in mind that synchronization between threads comes at a cost. Algorithms that requires little or no synchronization usually yield better performance.

5 Marking Scheme

The tentative marking scheme is as follows.

Item	**Marks**
Primes are correct	3
Primes are sorted	3
Performance	6
Algorithm quality	4
Use of libraries	2
Code quality	2
Total	*20*

Items are detailed below. Please note that the grades of the first three items all depend on the recall[2] metric of your algorithm. This enables you to trade correctness for performance, as constrained algorithms usually do. You can check the grade formula in the `PrimeComputerTester.java` file, lines 188 – 195.

Primes are correct. False negatives are tolerated, false positives are not. If the algorithm is not 100%-precise, the grade is 0.

Primes are sorted. The grade is proportional to the sort rate of the computed primes. This rate is 100% if primes are all sorted is ascending order.

Performance. The grade is proportional to the speedup of your algorithm compared to the speedup of the solution algorithm.

Algorithm quality. Your algorithm must be parallel, including the sort part. It must implement a sound load-balancing scheme. Thread synchronization must be as low as possible, if any.

Use of libraries. Your code must not duplicate methods of the standard Java library. Moreover, it must not use any other library than the standard Java library, e.g. Apache Commons.

Code quality. Your methods must be less than 30-line long. The rate of non-Javadoc comments must be over 20%.

6 Submission

6.1 Deliverable

The deliverable consists of the `PrimeComputer.java` file alone. In order to identify the team members, you must rename this file as `LASTNAME1.FirstName1.LASTNAME2.FirstName2.java` **without any space character** before submitting it.

6.2 Submitting

Drop your Java file into the submission area, **only once per team**. You may submit the file again as often as you wish until the deadline, provided you **always do so under the same Campus user**.

[2] Wikipedia – Precision and Recall: https://en.wikipedia.org/wiki/Precision_and_recall