Assignment: "Calculate Primes": (*Fork/Join* Practice)

Avraham Leff

November 10, 2022

COM 2545 Assignment

1 Assignment-Specific Packaging

The general packaging is unchanged from the basic "Homework Requirements" (see slides from first lecture and "Homework Policies for COM 2545" on Piazza).

This assignment's "DIR" **must be named** *PrimesFJ*.
You will implement **multiple Java classes** (see below).
Your write-up file **must be named** *PrimesFJ.pdf*.

2 Motivation

The assignments has multiple purposes:

- Give you practice using the JDK's "fork/join" APIs to implement the parallel decomposition algorithm technique discussed in lecture.
- Encourage you to think about the benefits and limitations of thread-based parallelization.
- Encourage you to think more deeply about the notion of "order of growth" and problem size.

You are *encouraged* to research how fork/join works in general and the JDK implementation in particular. You are *encouraged* to use similar examples as a template and **forbidden** to view code that implements this specific problem (assuming that it exists).

You are encouraged to learn more about prime numbers, but I suggest that you don't spend too much time with the topic because it won't help that much with the code ☺.

Note: the assignment is "about"

- Getting experience with "self-education", as you research how to translate the concepts you learned in lecture into concrete API calls.
- Getting "software engineering" experience as you experiment with your implementation, and produce a useful summary of your results.

3 Requirements

The problem you'll be solving is: "determine the number of primes below a given bound". The API for this problem is specified by `PrimeCalculator.nPrimesInRange`.

Your task is to solve this problem in multiple ways:

- Straightforward, "naive", fashion: `SerialPrimes`. Do not use clever algorithmic techniques such as Sieve of Eratosthenes! We're only using this class as a baseline measurement.
- `TwoThreadPrimes`: create exactly two threads, partition the problem space between them, then use the SerialPrimes algorithm to solve its part of the problem. The implementation then combines the results of its two threads.
- `PrimesFJ`: use Java's *Fork/Join* framework along the lines discussed in lecture.

3.1 Non-Programming

This portion of the assignment is worth 30%, and will be graded for "completeness" and "clarity".

Your writeup must contain the following subsections, in the specified order below.

3.1.1 SerialPrimes

Your writeup must address the following questions (no more than two paragraphs, ideally one).

1. If we set the `start` parameter to a constant of 2, we can characterize "problem size" in terms of the magnitude of the `end` parameter. From this perspective, what would you expect the SerialPrimes *order of growth* to be?

2. What is the actual order of growth?

3. Explain the observed behavior.

3.1.2 TwoThreadPrimes

Your writeup must address the following questions (no more than two paragraphs, ideally one).

Define algorithm performance as *"performance relative to SerialPrimes algorithm"*.

1. What is the theoretical best performance for TwoThreadPrimes?

2. What did you observe? Create a graph showing your results with *"increasing values of the `end` parameter (holding `start` constant at 2) on the x axis, and algorithm performance (see above) on the y axis"*

3. If there is a non-trivial discrepancy between "expected" and "actual", provide a convincing explanation.

4. What is the observed order-of-growth?

3.1.3 PrimesFJ

As you know, the value that you set the *threshold* parameter is an important determinant of the performance of any *Fork/Join* algorithm. Optimize your threshold parameter to have your *Fork/Join* implementation perform as best as it can for large values of the range $2..end$ where "large" means "hundreds of millions".

Your writeup must address the following questions (no more than two paragraphs, ideally one).

Define algorithm performance as *"performance relative to SerialPrimes algorithm"*.

1. How many cores does your laptop have?

2. What is the theoretical best performance on an n-core machine?

3. What did you observe? Create a graph showing your results with *"increasing values of the `end` parameter (holding `start` constant at 2) on the x axis, and algorithm performance (see above) on the y axis"*

4. Pick a "very large" value of the `end` parameter (holding `start` constant at 2). Graph the performance of your *Fork/Join* algorithm on the y axis against various threshold parameters on the x axis (smaller to larger). How sensitive is your algorithm to changes in the threshold?

5. If there is a non-trivial discrepancy between "expected" and "actual", provide a convincing explanation.

6. What is the observed order-of-growth?

Finally:

- Consider the order-of-growth of the three algorithms. Do they differ (non-trivially)? Report your results and explain them (not more than one paragraph).

3.2 Programming

This portion of the assignment is worth 70%, and will be graded for "correctness" and ability to meet the "Big-O" requirements at scale.

To be explicit: your algorithm implementation can use whatever set of clever techniques you choose, but **must conform to the stated design**. For example, "sequential must be sequential", "two threads can't be four threads", "no caching of results", you get the idea.

> Please review the general requirements for a programming assignment! I've tried to reduce the chances of "mistakes" occurring through the use of a "skeleton interface", but ultimately, **you are responsible** for ensuring that I can compile and test your code without incident.

Begin by downloading the interface and three skeleton classes from the `PrimesFJ` directory (git repository is here. Then, implement the stubbed methods.

3.3 SerialPrimes

This class provides a *serial (sequential)* algorithm for counting prime numbers. Your implementation should be straightforward: make sure it's correct!

3.4 TwoThreadPrimes

This class provides a *parallel* algorithm that uses **only two threads** to parallelize the computation.

You will be graded in terms of how closely your implementation approximates the theoretical performance improvement of two threads relative to one thread.

3.5 ForkJoinedPrimes

This class uses Java's *ForkJoin* framework to count the number of primes. You will be graded based on how closely your results perform relative to the theoretical maximum.