Judah Goldring
Algorithms 2545
Professor Leff

Serial Primes:

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?

   O((n)^sqrt(n))

2. What is the actual order of growth?

   If end parameter = n

   And start parameter = m

   O((n-m)^sqrt(n))

3. Explain the observed behavior.

   Serial primes loops through all of n, and for each n it loops through a second loop runs until its sqrt.

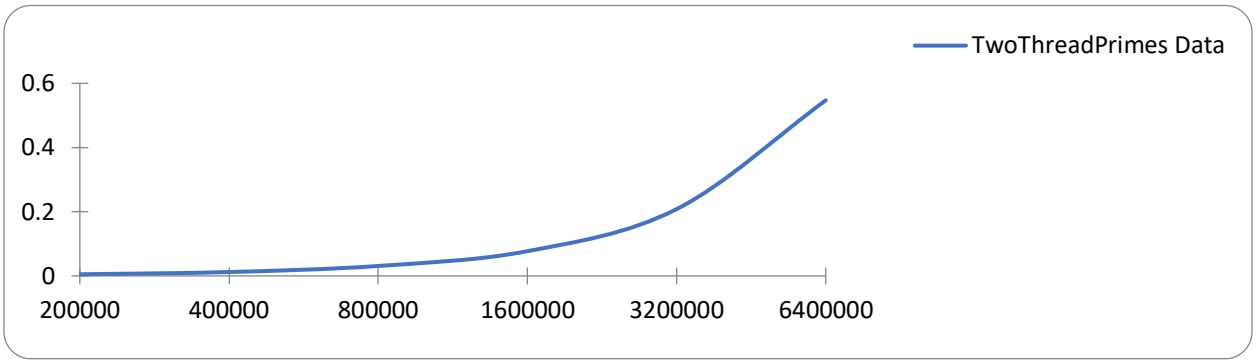
   ```
   for(i = m; i < n; i++)
       for(j = 0; j < sqrt(n); j++
           if is prime
               counter++
   ```

Two Thread Primes:

1. What is the theoretical best performance for TwoThreadPrimes?

   Half the time it took for serial primes, since the work is being split in two "simultaneously".

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.

   If you split the number of primes to process in two, meaning instead of doing x—n in one go, you did x – n/2 as one thread and n/2+1 – n as the second thread. The work for the second thread would be done on much larger numbers making everything more expensive since it would have to check every number less than it's square.

   Also creating threads in of itself is expensive so would also effect run time.

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

   The order of growth is still O(n^(sqrt(n)) but its done in "half" the time because of two threads.

PrimesFJ:

1. How many cores does your laptop have?

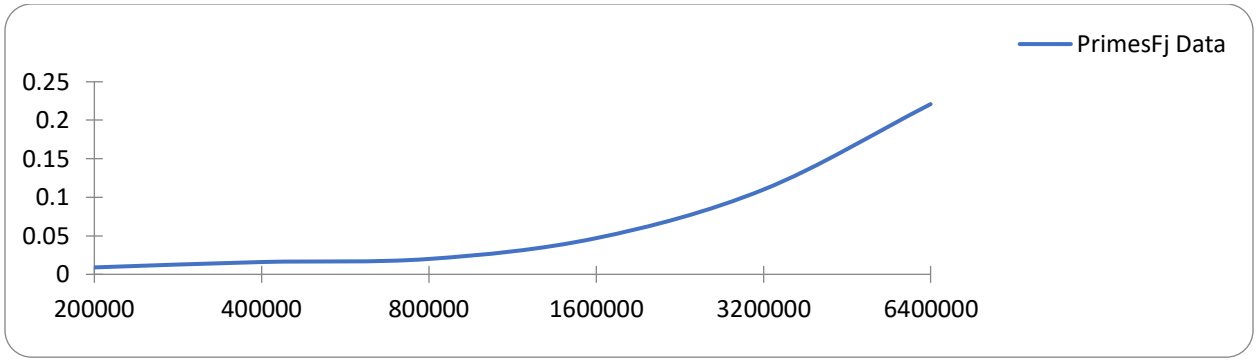
   8 cores.

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

   The PrimesFJ should perform around n00% better than the sequential algorithm.

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 thresh- old?

   My algorithm isn't super sensitive to threshold change since the change in threshold doesn't give that much of an improvement in algorithm performance compared to the sequential algorithm. The difference between a threshold of 500 to 1000 to 1500 to 2000 to 2500 to 3000 was all within 50% when input reached millions of numbers.

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

   Different cores on computer can be used to complete different tasks on the computer that’s not running the algorithm so their full power isn’t necessarily used.

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

   Same as the other two algorithms, just completes the required task in quicker time since its broken into smaller parts simultaneously.

Summary:

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).?

The order of growth doesn’t differ (non-trivially) because the sequential algorithm used remains the same. The only difference between the three algorithms is the quantity of n the sequential algorithm performs on.