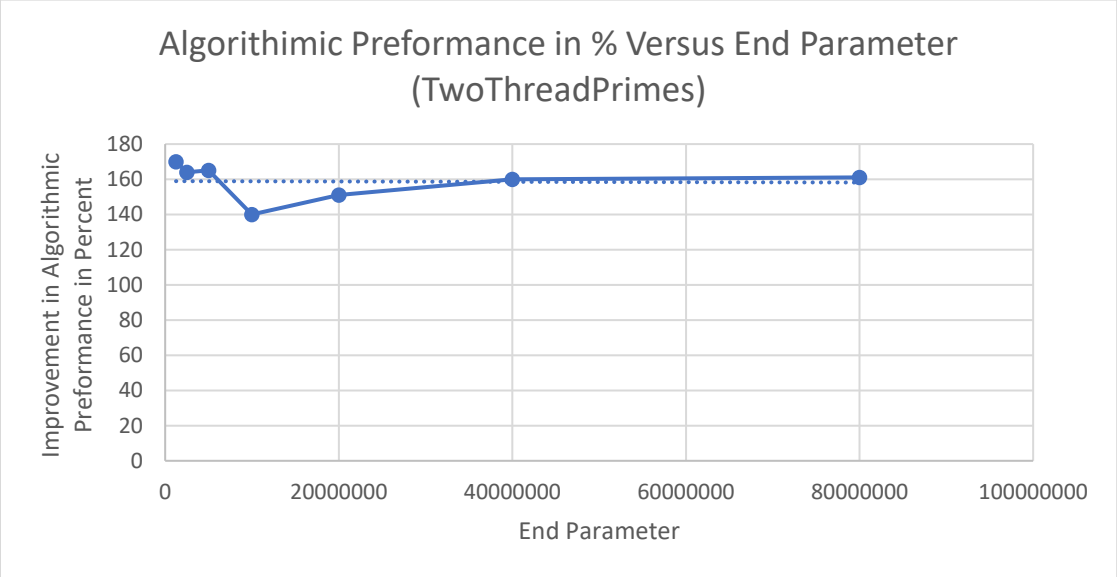
Elad Ohayon

3.1.1 SerialPrimes:

If we set the start parameter to a constant of two and look at problem size as a factor of the end parameter, the expected order of growth would be "very" sub-O(n^2). The actual order of growth is "quasi" linear with the doubling ratio measured to be ~2.7. The algorithm is closer to nlogn and O(n) than O(n^2) because of its behavior. If the number is divisible by 2 it is an O(1) operation to determine it isn't prime; this is the best case and is O(1) * O(n). The worst case is that we must check from 2 through $\sqrt{n}$ (in situations when the number is prime). The best case is far more common (occurring ½ of the time) as opposed to the worst case which happens when the number is prime. We can estimate the number of primes from 2 to n using the equation $\frac{n}{ln\, n}$. This means a prime occurs roughly $\frac{1}{ln\, n} \times 100\%\ of\ the\ time$. Assuming that if a number is not divisible by two it is prime (since we cannot predict how many numbers we must check for each potential prime), the time complexity would be O(n) * $(\frac{1}{2}O(1) + \frac{1}{2}\ sqrt(n))$.

3.1.2 TwoThreadPrimes:

The theoretical best performance of TwoThreadPrimes is ½t where **t** is the time complexity of SerialPrimes. Meaning TwoThreadPrimes should have a 200% improvement compared with SerialPrimes.

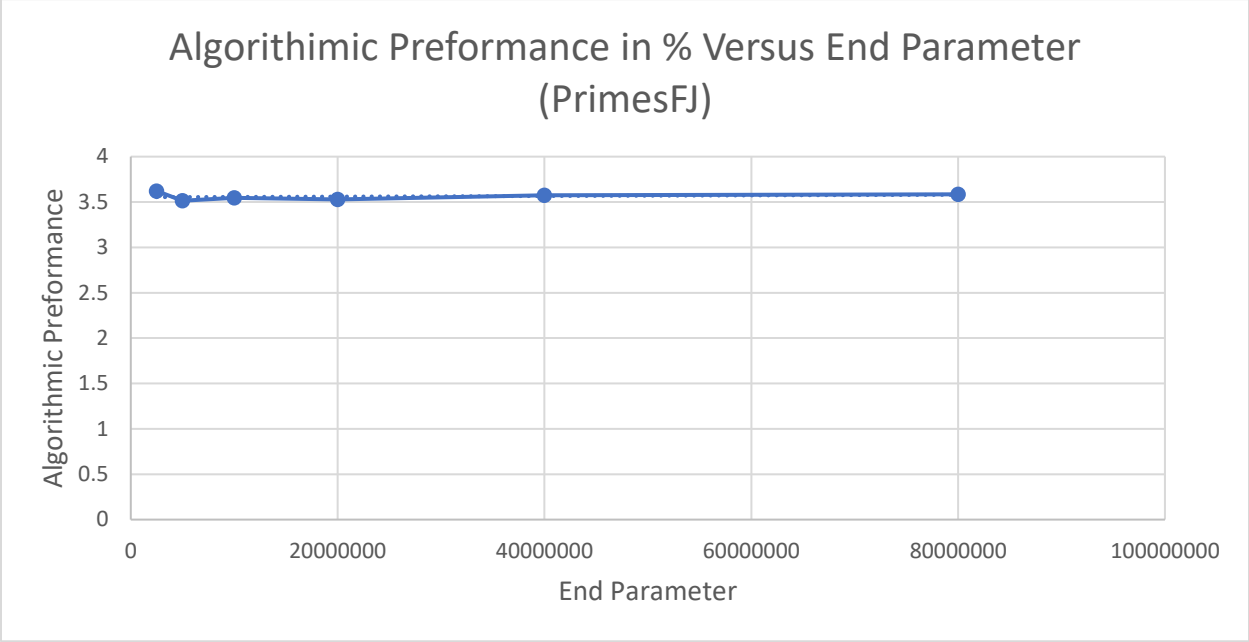


Looking at the chart above, the observed improvement was ~ 160% (and not 200%). TwoThreadPrimes was not able to reach the theoretical limit because the amount of computation done by each thread is not equal. As the range is split into two threads where **left** takes 2…end/2 and **right** computes end/2+1… end, the **right** thread calculates larger numbers which takes more computations because of the $\sqrt{n}$ discussed in SerialPrimes. This means the left thread will finish its computations a lot before the right thread, and sit idly and wait for the right thread to finish, causing TwoThreadPrimes to behave similarly to SerialPrimes once the left thread dies out. The observed order of growth is the same as SerialPrimes

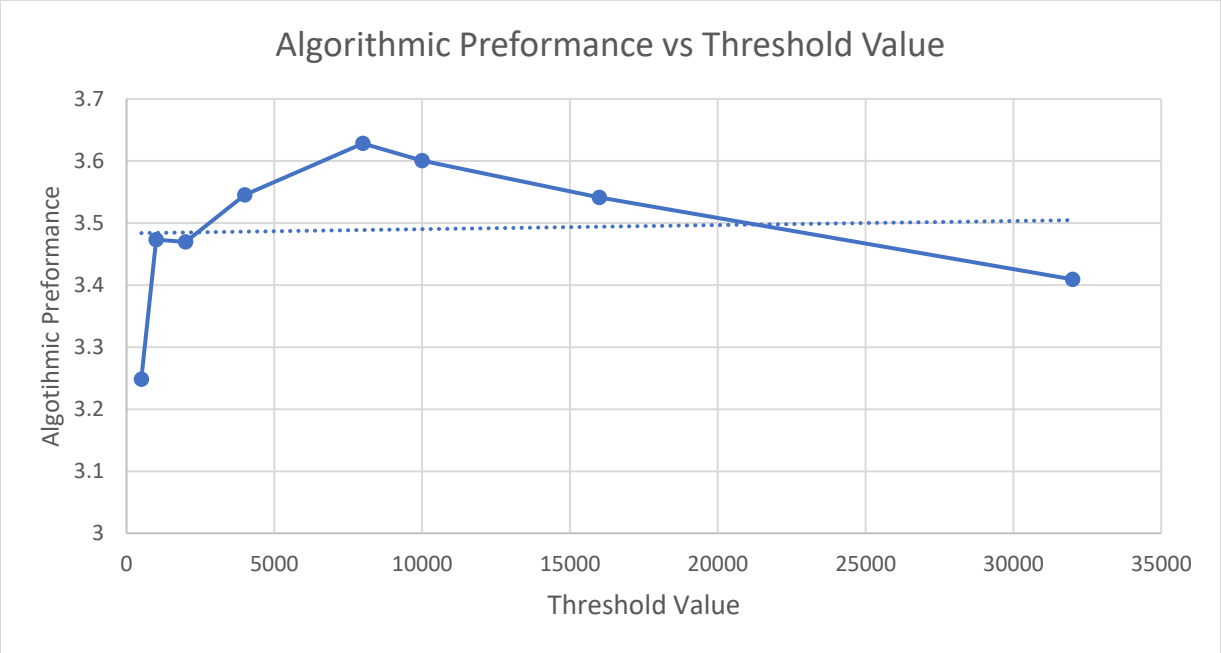
as multithreading merely decreases the time complexity by a constant, in this case by ½, but doesn't affect the "Big O".

3.1.3 PrimesFJ

My computer has 4 cores which through hyperthreading, has 8 "logical processors". Java, however, says my computer has 8 available processors. In terms of experimental observation, it seems the most improvement my machine could have is 400% (see below). The best theoretical performance an n-core machine is a n*100% improvement assuming that there are no sequential parts to the algorithm.



From the chart it can be observed that Java's ForkJoin Framework resulted in roughly a 350% improvement in performance, resembling the number of physical cores on my computer.



From this graph, it can be observed that algorithmic performance increases with the threshold parameter until about 8000 in which case algorithmic performance begins to slowly decay until it hits a too large of a threshold value. In other words, there is a large improvement when the threshold is raised from 1 to 1000. For example, from a threshold value of 500 to 8000 there is an improvement of 50%, but from 8000 to 10000 there is a very slight decrease of ~15%. There is roughly a difference of 50% between the theoretical best and algorithmic performance due to the cost of creating threads and the sequential parts of the algorithm. The observed order of growth is roughly $\frac{1}{n}$ **T** where **T** is order of growth in SerialPrimes, and n is the number of cores.

Overall, the order of growth of each algorithm varies by a constant. Where this constant represents the number of cores. As such the order of growth is $\frac{1}{C}$ O(n) * $(\frac{1}{2}O(1) + \frac{1}{2}\ sqrt(n))$. Where for SerialPrimes C=1, for TwoThreadPrimes C=2 and for FJPrimes C=number of cores. While this difference is trivial at small runtimes, such as cases when it takes 1 second for SerialPrimes to return a value, it becomes significant at larger ranges like 100,000,000 where FJPrimes takes 28 seconds and SerialPrimes takes roughly 108 seconds. Scaling this, something that takes a sequential algorithm 1 year to run would only take 365/number of cores time to do; and with 365 cores, it would take only a day.