Brian Rodriguez

Assignment 2

1.

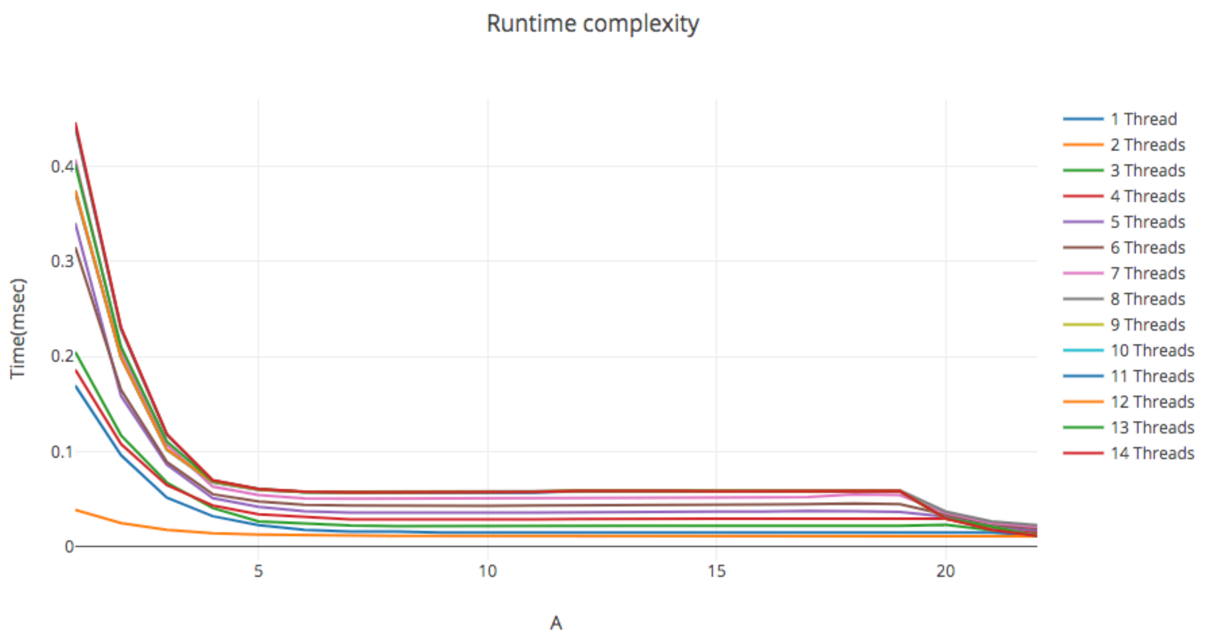
a. Using a vector of Booleans would be a quick and dirty way of reporting the primes found. One way I could think of is using the indices of the vector report whether or not a prime was found. Unfortunately, since a vector of Booleans is not thread-safe due to bits being shared across the whole vector then it cannot guarantee the validity of the Booleans that are being stored inside. I have tested by implementing such a method. With a prime range of 100 the only numbers being reported are in the range of 1-6 inclusive. Running other tests there are numbers being reported that are not prime; the range does not affect the reports coming back from the program. It is safe to say that this method cannot be truly used unless dealing with the vector a bit level, which I do not think is possible

b. In order to realize a solution like this, you must first install tbb onto your machine using homebrew or the source code. Using this concurrent vector with tbb or openmp libraries is quiet simple. The concurrent vector essentially does all of the overhead of worrying about pushing new objects and resizing of the vector for the user. This allows allows the user to reduce the amount of space needed to store all of the time numbers. Instead of having to allocate an array or vector the size of primes range, using concurrent vector allows to just push all of the prime findings and report them in the order that was received. If we were to go further, we would use the standard library algorithm class to sort the primes in ascending order, allowing us to report the primes in sorted order. This of course adds more overhead to the matter.

c. Using an array of size primes range allows us to keep track of which prime numbers were found using indexing. Reporting that a prime number was found is achieved by making the initialized values as zeros and changing the zeros to 1 if a prime number was found. After the parallel work has been done we are able to report the primes in order by checking if the index has been changed to a 1. The only drawback to this method is having to use #pragma omp critical when writing to the array. This adds a considerable amount of overhead, since all of the threads must stop working when they reach this point in the code until the OpenMp lock has been freed by the current thread writing to the array.

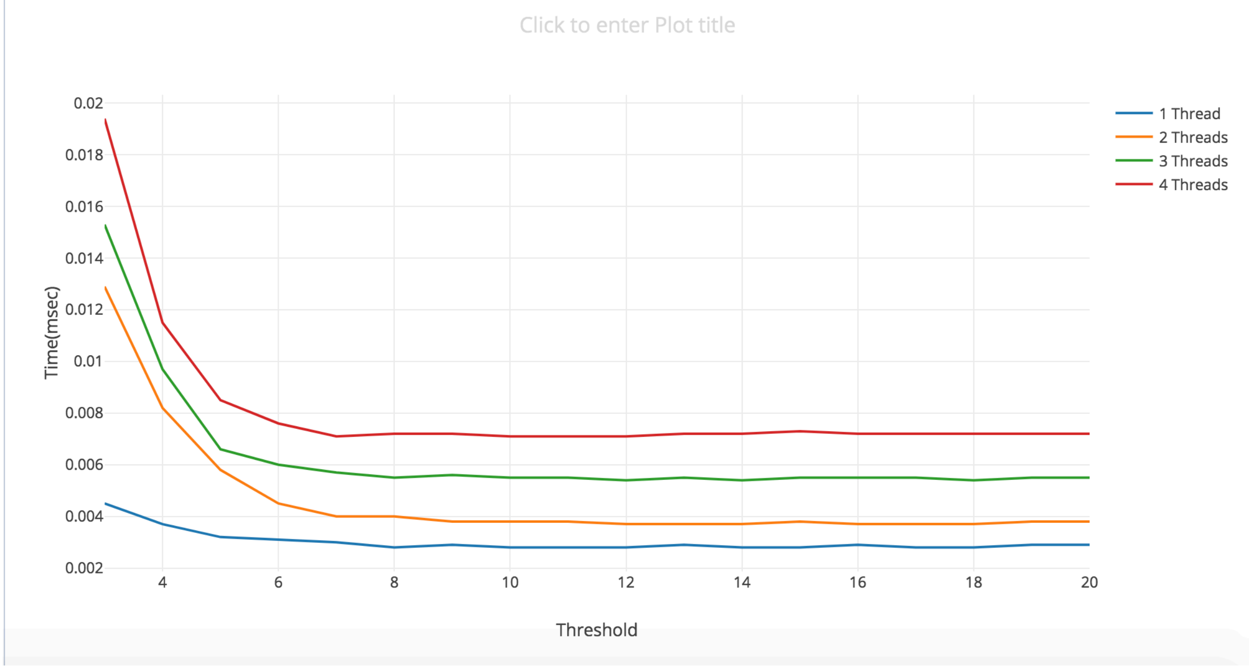
2.

f) The following graph depicts the runtime complexity with respect to the number of threads and threshold size increasing in powers of 2.



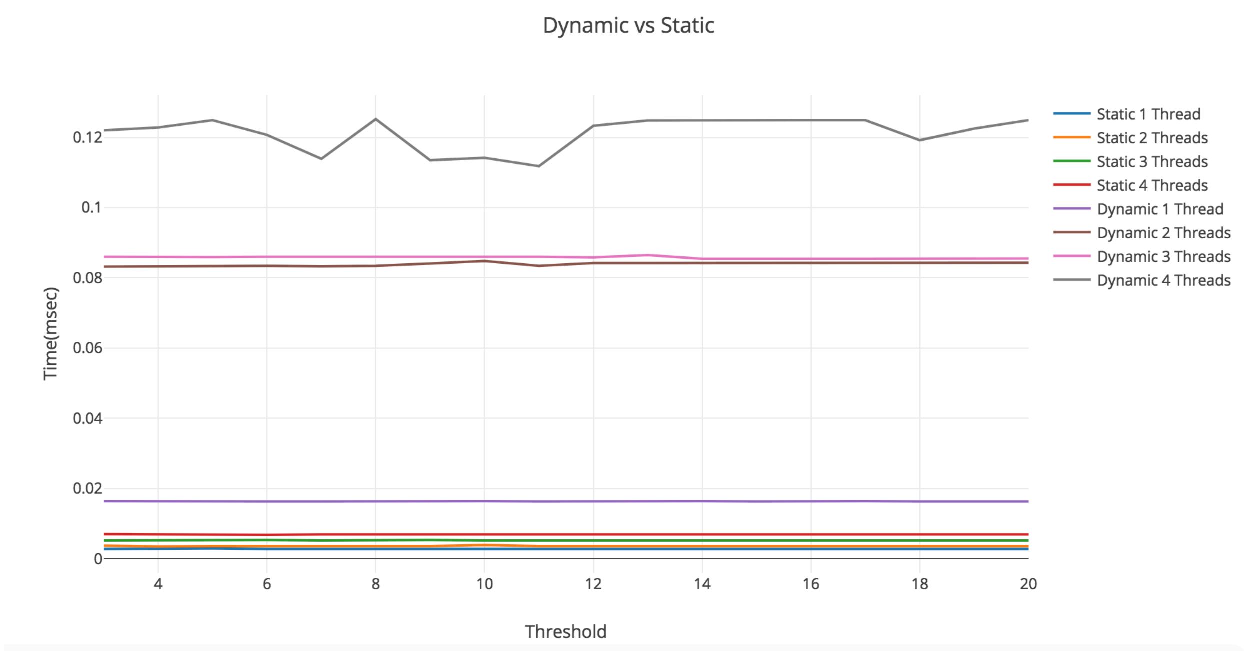
I am unsure what to think of these findings. It seems that the speed up of using multiple threads even to the number of cores is zero to none. It seems as if the single thread is considerably faster than any of the number threads that were tested. This can also be a wrong implementation of the dot product using openmp but I do not think this is the case. Perhaps a flaw in my timing code?

g) The following graph depicts the areas of interests.



The areas of interest that I saw to have the most speed up were between 2^3 threshold to 2^8, with the number of threads being from 1 to 4. Anything passed these boundaries a slowdown will start to occur. Judging from this graph it still seems as if though one thread is the winner followed by 2 threads. It still does not make sense to me and more testing needs to be done on the matter.

h) The following graph depicts the runtime complexity for both Dynamic vs Static OpenMP scheduling.



Judging from the graph, it is easy to see the considerable slowdown that the Dynamic scheduling interface gives. This is because of the reassigning of work that OpenMp gives per thread created. This means that the thread is blocked after every end of the iteration loop to be assigned new bounds to work on. The pros are that you always know that a thread will be assigned work at the end of its iteration loop. Static on the other hand gives a considerable amount of speed-up. In the Static representation, each thread is assigned boundaries to do work on as opposed to dynamic where bounds are assigned at the end of the workload.