Report

General approach-

When taking this problem into account with the I/O heavy lifting already being implemented most of the focus went to figuring out where the paralyzing would happen. It made sense to create a function that would do the serial addition and then have multiple threads at different moments call on that function to calculate how many of each letter appeared. After creating that part in main, the serial version was done. The next important step was then to figure out which sections of the code to write in parallel so that the overall code would run faster at certain times than the serial edition would. I settled on making an array of atomic integers of size 26 to represent where each one of the indexes would represent each letter in the alphabet. The reason that Atomic Integers were chosen was because when the first scalability assignment was done, they were the fastest as can be seen with this graph.

The graph below shows the time that the program took to run when compared to different ways of writing code in parallel.

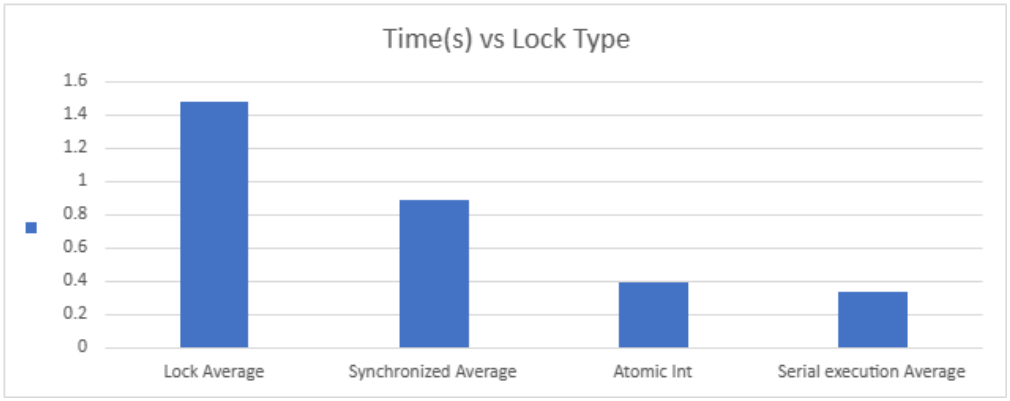


Figure 1: Speed of different locks on the same code

While in this case the serial execution was faster, the important thing to note is that Atomic Int was significantly faster than the other lock types and for that reason it was picked. From there the important thing was to determine how many threads to make in which to start I used the java code provided of "Runtime.getRuntime().availableProcessors()" to get the available number of processors and used that as the number of threads at first before trying to optimize this number. This means if there were 8 processors that were available then I would make 8 threads that way no matter on what computer it was run it would be able to create the threads required. Here is a diagram of the how I was going to run the program in parallel.

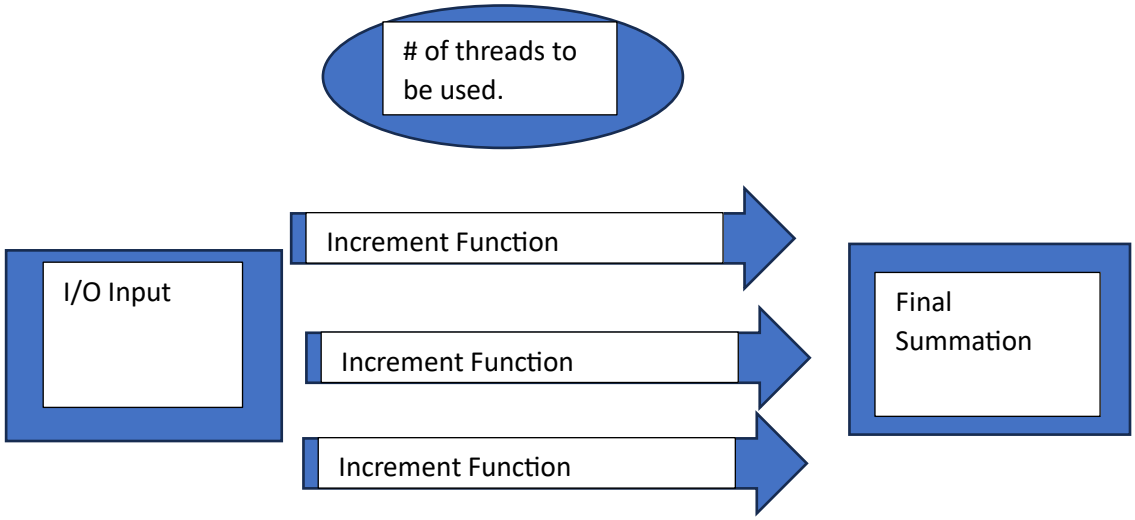
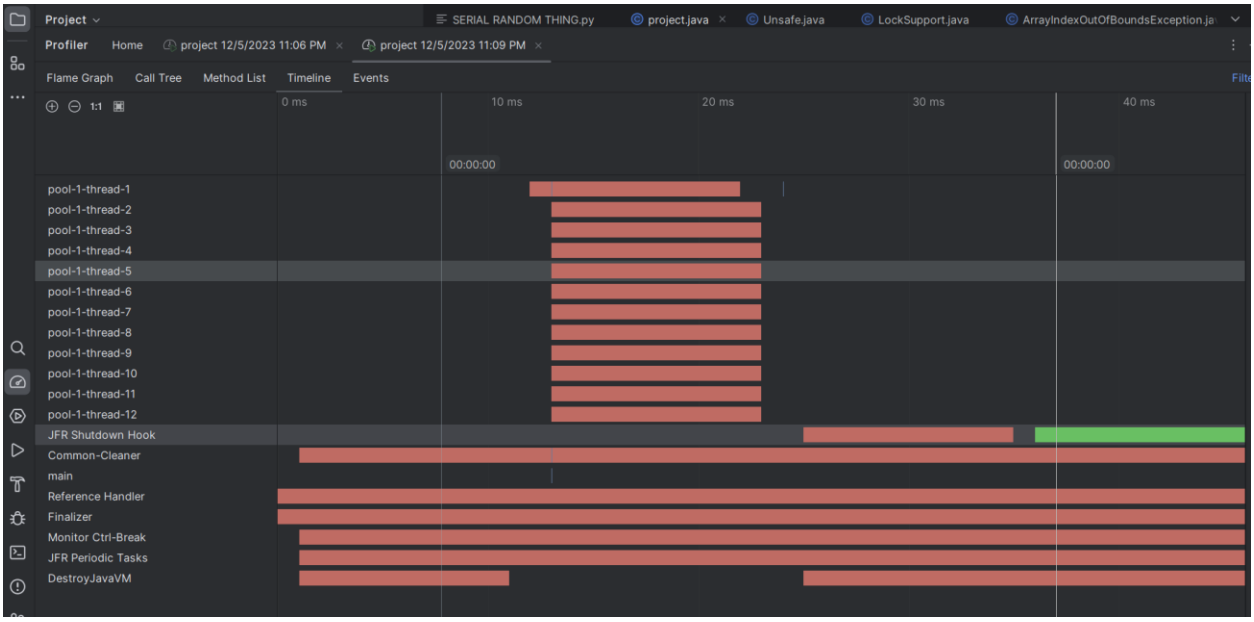


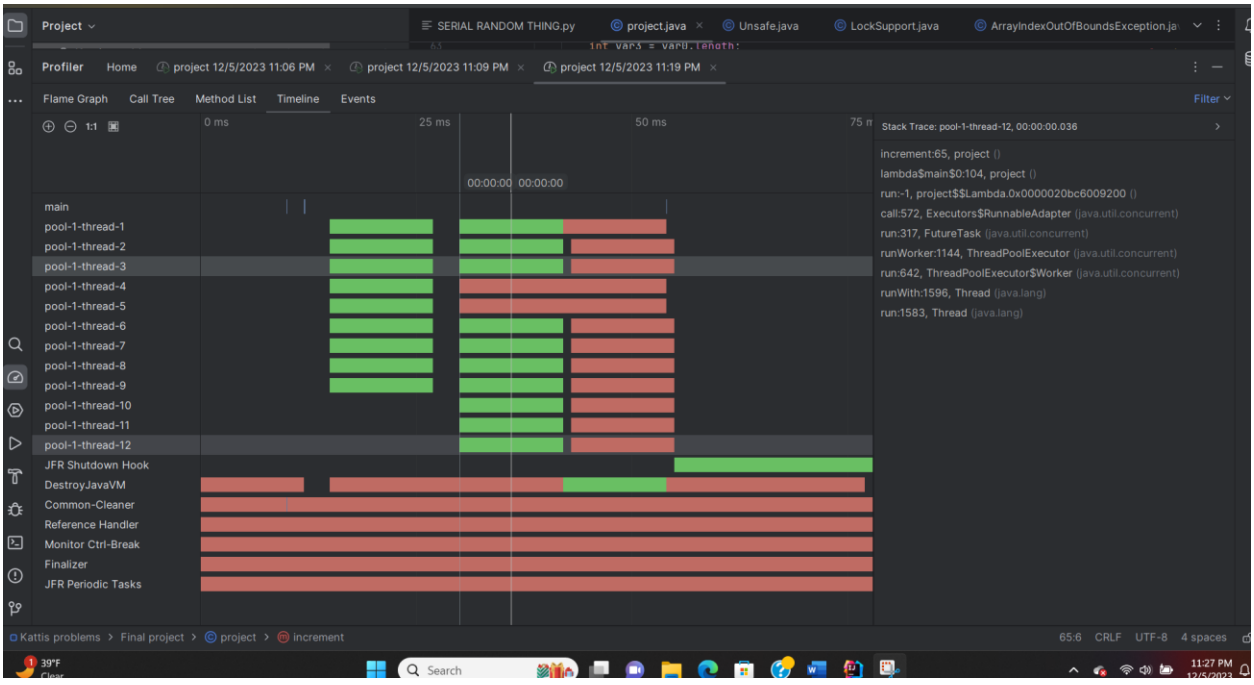
Figure 2: Plan of how the code would work.

One important thing to note is that because we are using atomic integers there is no final piece of code that must combine the different sections together. Each is simply going through the increment function and adding to each element where it applies without having to worry about concurrency issues. However, when deciding on how many threads to make this was something that I didn't run that many tests on to determine, and just set it to be the max value of the available processors. Knowing Amdahl's law, some logic needs to be considered to determine at what point the cores start to reach diminishing returns. To do this I used the current amount of cores in a profiler to see where there program ran slowly and what would be factors holding it back.

After a painful amount of time, I set up a profiler on this computer and here is the updated diagram of the how to program ran with size 1,000.



As we can see there the total run time of this example was 40 ms overall however something super interesting to note is from my understanding the red means it was in parked state of each thread so as if it isn't working in parallel but temporarily suspended. If we then look at the example of 1 billion characters:



The obvious thing is we notice greener and essentially what that means is that we notice that t.

In the first example, thread one almost instantly completes the summation or there is green there which I assume is the increment function running and the thread not being in park and the rest of threads just being created and never get the ability to actually do any summation towards the answer while in the large 1 billion file they all of them use the increment function at some point. One interesting thing to note is that when working on efficiency the IDE told me how long each section of the function increment took to run and here and the program told me that 247 ms were spent on the increment function which it said was 86% of all time this leaves the non-threaded parts to be running the remainder of the time which finalSummation, the function that prints out the answer was clocked in at 13 ms which was 5% of the time. However, it is important to note that the increment function 86% of the time does not mean that F_p = 0.86. It simply means that most of the time was spent trying to increase and waiting. I started with a simple break down that then I used tests to then build for calculating F_p.

At first I took the approach that I really only had three functions:

1. Input
2. Increment
3. Final Summation

Only function two was the section that is using threads so therefore taking heavy lifting, overhead and all other factors and throwing them out the window, I settled on setting F_p = 1/3, with the goal to further refine it as more testing went on. As a result of this F_s is then set to 2/3. We can then graph speedup as a function of P where P is the amount of Processors to try to maximize how many processors to have.

$$s = \frac{1}{(F_S + \frac{F_p}{P})}$$

There are several ways to maximize a function but the easiest way is to take its derivative set it equal to 0 and solve for a critical point.

$$\frac{d}{dP}\frac{1}{(F_S + \frac{F_p}{P})} = 0$$

However, we run into a problem with this approach. When we take the derivative with respect to P of the equation we get the following:

$$\frac{3}{(2p+1)^2}$$

No value of P will make this equation reach 0 which means there is no absolute maximum value that we can set P to will give us significant speed up in this fashion. This is an important consideration about Amdal's Law. Paralyzing will only get us so far, and as we can see with the examples the effectiveness of the amount of threads seems to be mostly dependent on the size of the file that we are parsing. This calls for a new approach to determine the best amount of threads to use. Due to the structure of the original curve what we can do is take a limit as P->∞ what is the max possible performance boost given by this law.

Limit Work:

$lim_{x\to\infty} \frac{1}{\frac{2}{3}+\frac{\frac{1}{3}}{x}}$ if we evaluate this limit using limit rules, we get that the absolute max that x will ever be is 2/3 or 1.5x multiplication. This needs to be clear, however, this is the most perfect and impossible of worlds. Technically it is more accurate to say that every boost that comes from the use of threads will always be under 1.5X speedup unless you are using infinite threads. So, if the number of threads is highly dependent on the size of the input file (i.e. it would not make sense to have 20 threads for a file of size 3 since by the time all the pre-loading is done the computer will have finished if you had just ran it in serial). After spending seeing these charts and it made me realize that the most important consideration of how many threads to make is *the size of the input file*. From here I then went on to test large input files with different number of threads to see which performed better. To do this I graphed various time plots while trying a variety number of threads versus the time it took to finish running an input file of size one billion.

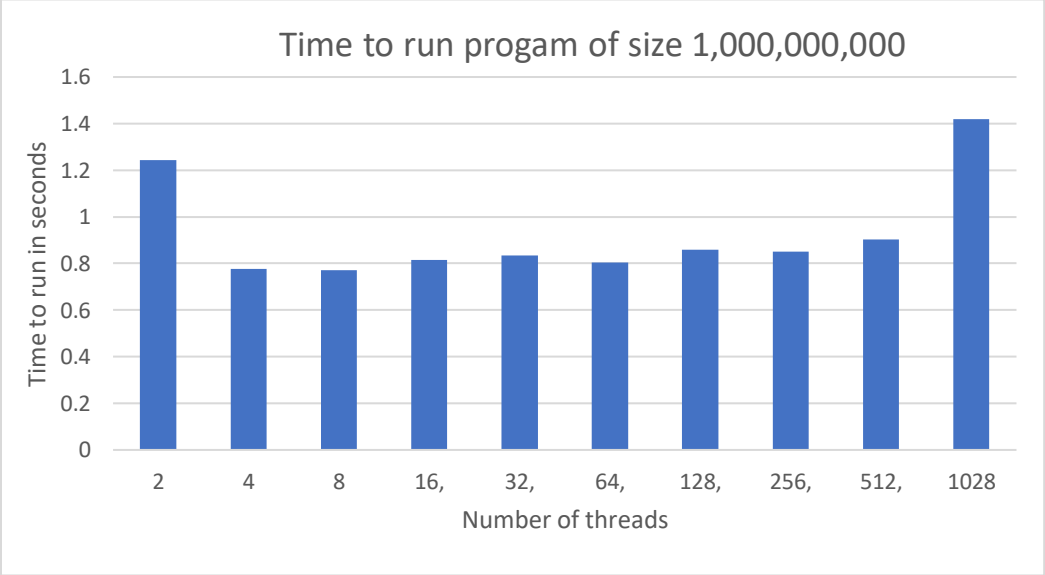
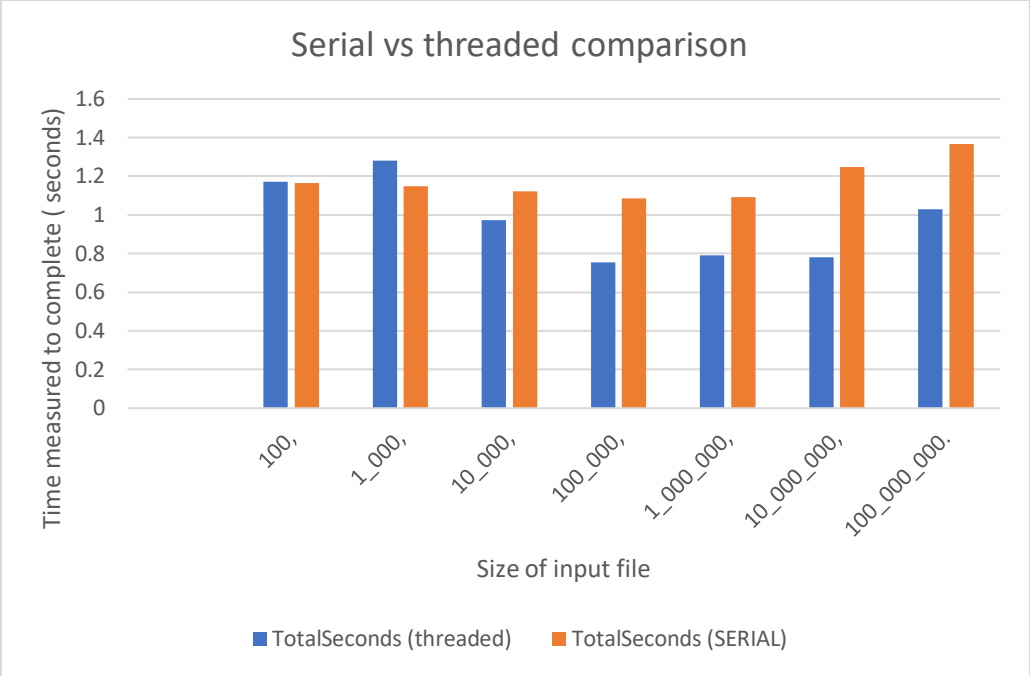


Figure 3: Finding optimal number of cores with test size of 1,000,000,000.

Takeaways:

The optimal number of threads to use is 4. On this system there is no significant change between using two threads and using 1028 threads in terms of speed which means that using that many threads is not worth the overhead because two threads run just as fast. Another key takeaway that was surprising was that at least on this machine, for size 1,000,000,000, the fastest number of threads to have been just four. However, this led to an important consideration. The number of threads is highly dependent on the size of the input file. At certain sizes it is better to just do it in serial, and other times it's faster if you thread the program. To see when this would be I test exactly that as we can see in the following graph.



Based on the following graph we can deduce that anything under the input of 10,000 should be just done in serial. Everything else should be done in threaded using 4 threads for this system. I am not sure if this speed will apply to other computers equally as different OS implementations and different machines could operate threads differently, so it is hard to say, but in general the number of available processors even if it is more than 4 is going to be faster than the serial version in the majority of cases.

Optimization:

There were many optimizations that were made along the way when writing this program in parallel. The most important one was figuring out how to take the input and send it to multiple threads without having to worry about double counting. One thing I did first was break up the sections into a range and then used the copyofRange function to copy that set of ranges of characters over to be a new byte array for a thread to then run that part exclusively. However, there is a big problem with this. CopyOfRange is an O(n) function in time complexity ***each*** time it runs. If I make n copies and each one is n time complexity we just successfully created an O(n^2) time complexity. What had to then be modified was the way the Byte array was passed into main that would then create threads. By changing the byte array to be a double array that then broke it up into the partitioned sections in each sub array of the 2D array when I was sending threads I just had to go through and set certain indexs of the 2D array to be a thread and run it. This significantly reduced the time my processes took.

Optimizations left:

The rest are potential optimizations that would help the program run faster. One thing is that I would check the size of the input file first after making it a double byte array and seeing if the size of the file is less than 10,000 and if so, run the program in serial. Other optimizations could be run to then determine the best number of threads to run on a particular machine, maybe having a function that

simply checks the number of threads compares is to the size of the file and then decides which to run the program in. Some problems with that are that pre-work could just end up making the overall program run slower because by increasing the pre-work there's going to be times where it would just have been faster to just run it without checking any of that information.