Lab 2: Multithreaded Quicksort using POSIX Compliant Threads

By: Alexander N. DePaolo

Class: EECS 3540 Operating Systems & Systems Programming

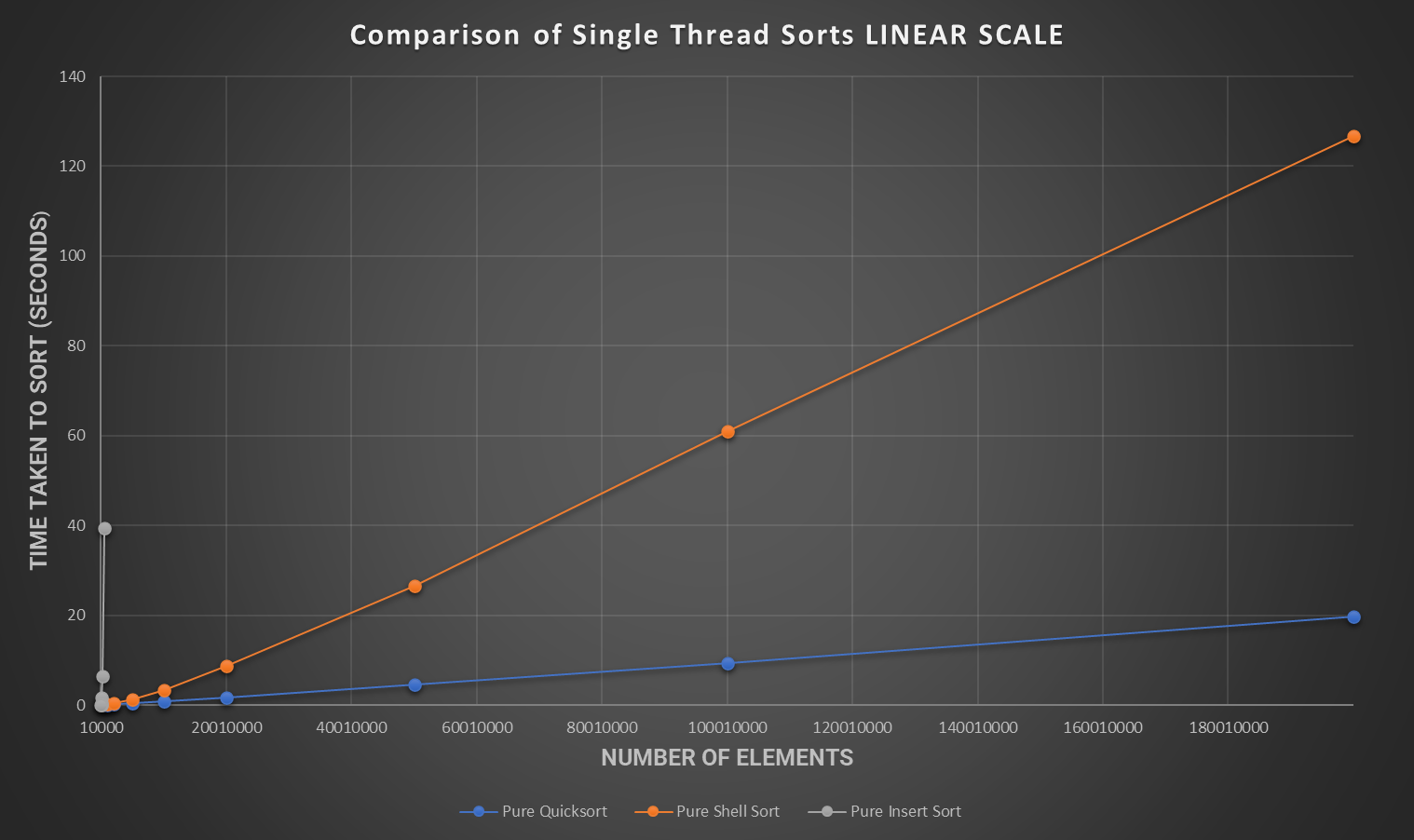
Teacher: Dr. Thomas

3/25/2023

Script 1 and 2: Single Threaded Testing on Quicksort, Shell Sort, and Insert Sort

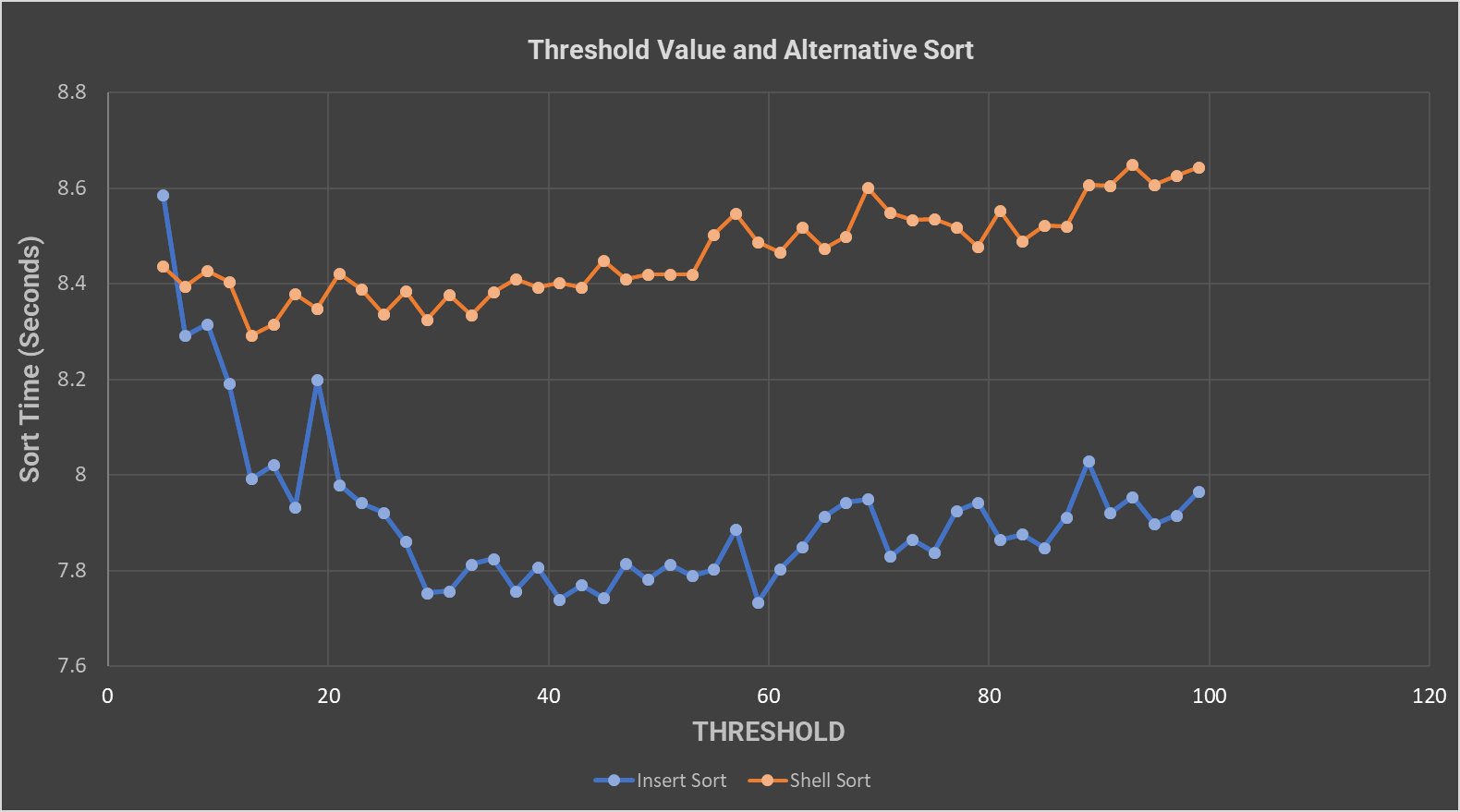
Chart, line chart

Description automatically generated



In comparing the three sorts purely on their own and in a single threaded application, it is clear that Quicksort at these massive numbers of elements reigns supreme. The speed and efficiency as expected of Quicksort allows it to process large arrays of numbers like this at a fraction of even shell sort or insert sort times. Examining the graphs above, Quicksort’s time growth with larger and larger amounts of elements is much smaller than Shell Sorts and Insert Sort. Shell sort’s time growth, although paling in comparison to Quicksort’s, is marginally smaller than that of the least efficient Insert Sort. Insert sort, if ran on the values that Shell and Quicksort ran on, wouldn’t even fit on the graph with its time values.

Script 3: Comparison of Optimal Threshold and Alternative Sorts in Single Threaded Quicksort

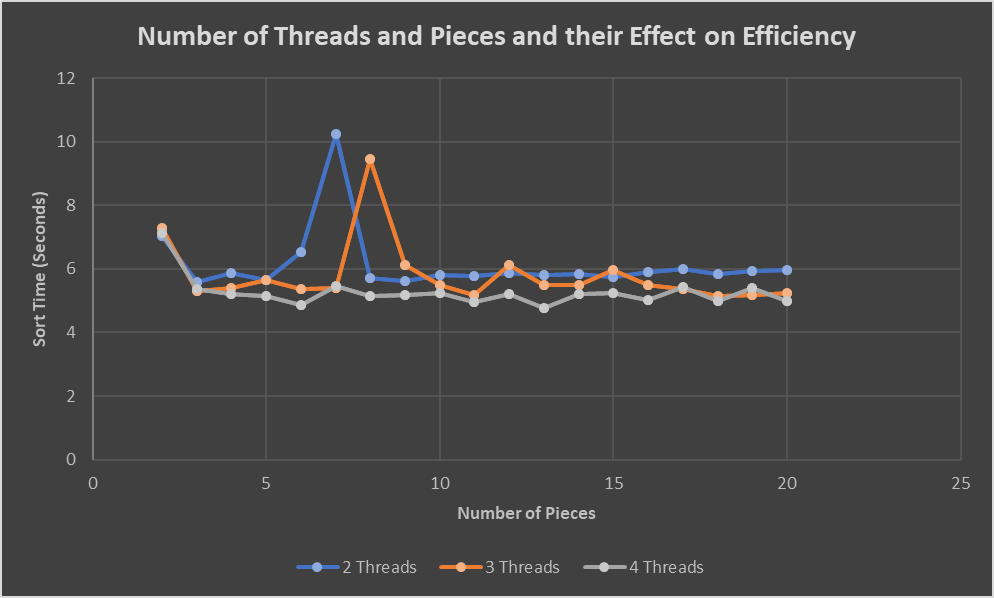


After running a handful of various increments of the threshold value, I settled upon incrementing it by a value of 2 with each test on each alternative sort. Examining the graph in closer detail, two distinct observations come to mind. First, shell sort ends up with much more consistent times over various thresholds as compared to insert Sort, which can drastically shift time wise in either direction depending on the increase or decrease of the threshold. Second, when paired with quicksort, insert Sort works quicker than shell sort. Thinking through how much more setup shell takes as compared to the simple comparison of insertion, it makes sense how it saves more overhead in the long run.

Though, something that was not expected was the range of lowest threshold values. As discussed in the previous paragraph, changing the threshold when shell sorting does not affect its time as much as insert sort. However, its lowest time came with a threshold of 13 while insert’s lowest was tied between 41 and 59.

Why do these threshold values produce the smallest times? I have a bit of a theory for the shell sort one. The 13 may be a sweet spot of sorts for shell sort. Since it deals with powers of 2 minus 1. Within a threshold of this size, the highest comb/tooth value it deals with is 7 and it only has to go through 3 different tooth values: 7,3, and 1. This value isn’t so small that shell sort is unnecessary(I.E, a threshold of 4 would be so small that I’d be better to use insert) but also it isn’t so obnoxiously large that shell sort takes longer than it should. As for insert sort’s threshold, it may be another sweet spot of sorts. It’s just big enough to where insert can handle it on its own but not so small that it doesn’t the total amount needing to be sorted.

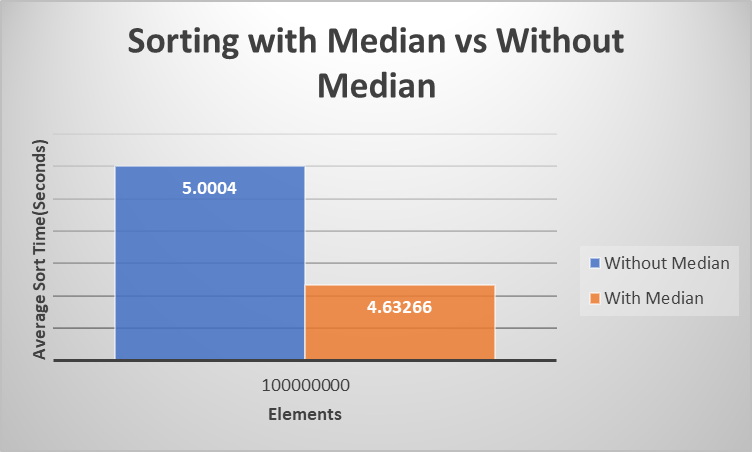
Script 4: Comparison of Threads, Pieces, and their Effect on Efficiency when Manipulating them



After nearly 7 hours of manually running each set of 20 runs individually and filtering the results, I finally got a decent graph. Examining closely, it’s no surprise that when the number of threads is increased, performance does improve, although only by a little bit, once again any bit of an increase in efficiency helps for the long run and for larger sets of variables. Another interesting find was the plateauing of time values when reaching a certain number of pieces. For all three different variations of the number of threads, when increasing the pieces, they eventually reached a semi-asymptotic time value. There were some dips here and there, but nothing major like the very beginning of each thread.

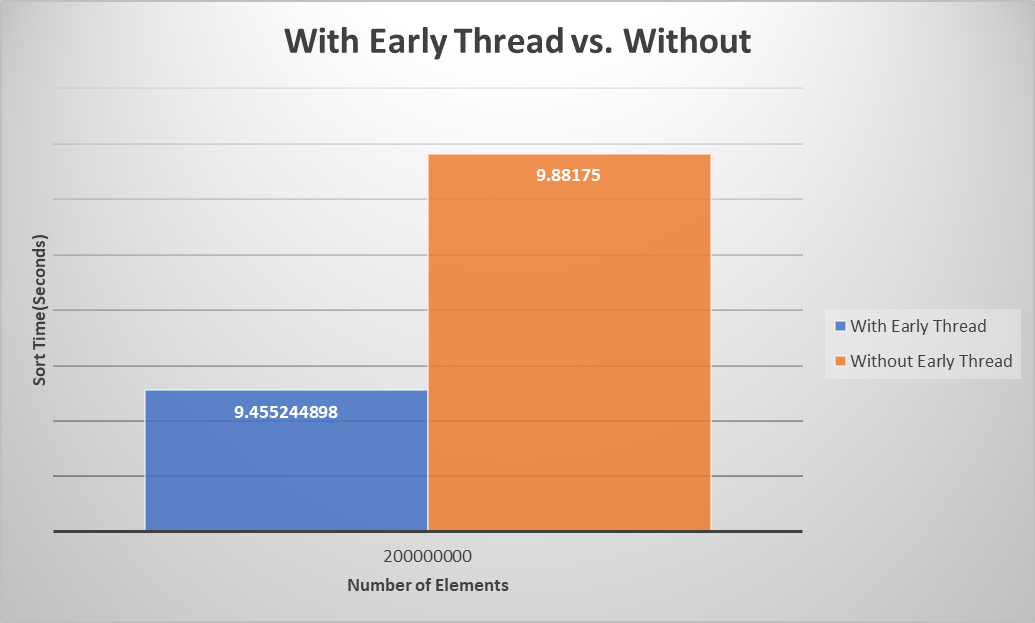
This brings me to my second observation, that of the initial peak then the huge jump all of the threads made right between the 5-10 pieces range. These may be due to slight outliers that occurred during running that I might have missed during my filtering. That or it’s one of the sweet (or in this case “sour”) spots of sorting. It may be that partitioning into those numbers of pieces with that many threads causes weird partitioning and partition handling and thus increases overhead. Then again, when examining the 4-thread series line, it does not have as major a jump as the others, thus I’m personally leaning towards a missed outlier or two within the average calculations. Overall, the changing of pieces didn’t affect the efficiency too much, but changing the threads did help the speed to a certain degree.

Script 5: Comparison of using Median of Three on Multithreaded Quicksort



The median of three method of partitioning both implemented in the quicksort and the multithreaded part of the program produces a significant boost in speed. It improves our partitioning to give more even splits so that one side or one thread isn’t doing more work than it needs to be efficient. Although it only gives an approximate average speed up of 0.371 seconds, this difference is substantial as the input sizes get larger and larger.

Script 6: Comparison of Early Thread and Without Early Thread Multithreaded Quicksort.



Examining the effects on the average wall clock times, it is safe to conclude that the early thread does indeed help cut down on overhead times. Between the averages, approximately 0.4265 seconds were saved when using the early thread. It makes sense as to why this would be, since we always run the early thread on the smallest initial partition piece, we effectively cut out what might be the unnecessary splitting of a small part that can be sorted much faster on a singular thread. Overall, the results for this test were expected.