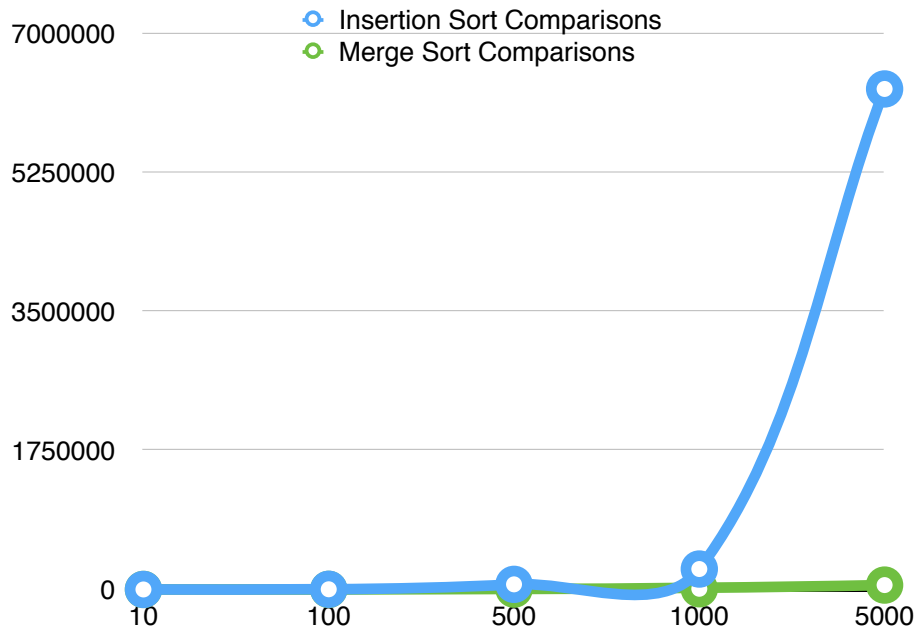
Experimentation:

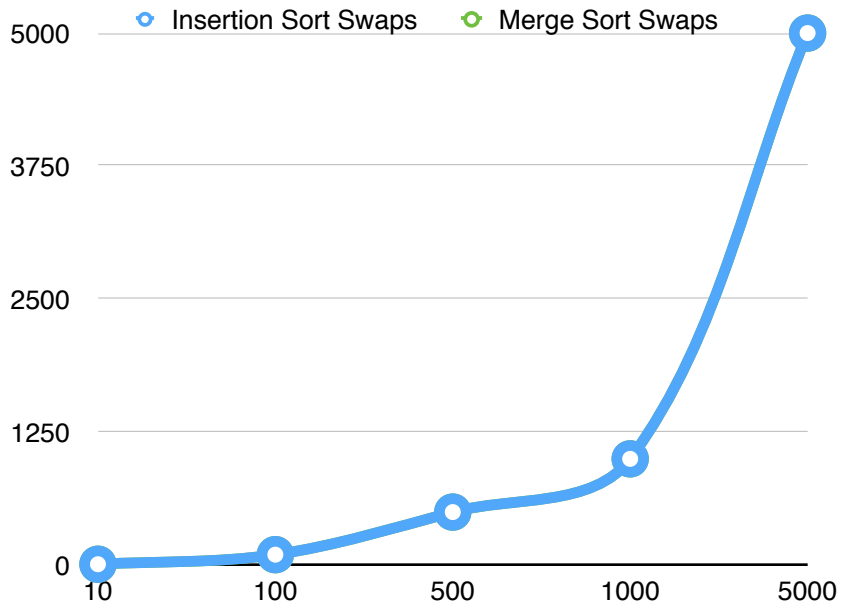
I have devised my experiments by using the traditional sequence.txt files we have been using for the duration of this project.

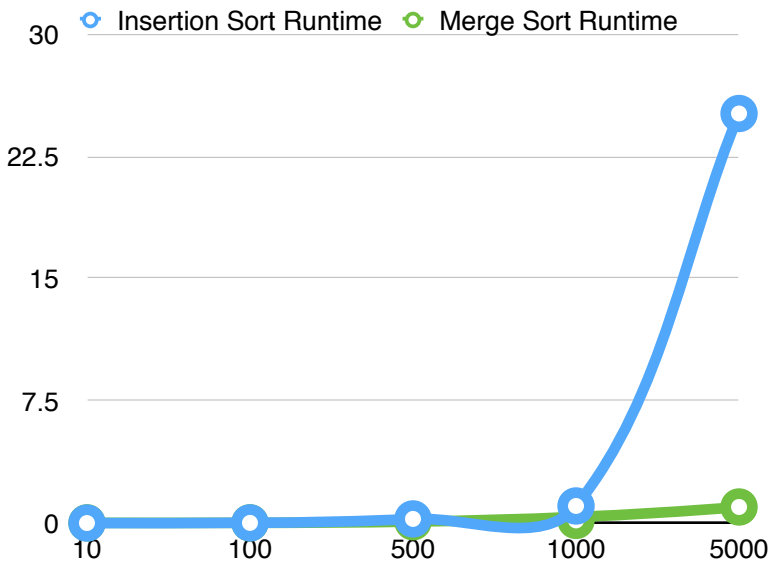
For usage in all experiments, I have written test fitness data similar to that provided, but in incremental lengths of 10, 100, 500, 1000, 5000, 10000 lines of fitness data. I feel that these are good increments that will test various experiments with different quantities of data length.

In order to perform experiment 1, I wrote a sequence file (experiment1.txt) to compare and contrast the comparison count, swap count, and runtime of different lengths of data using both merge sort and insertion sort.

Results:



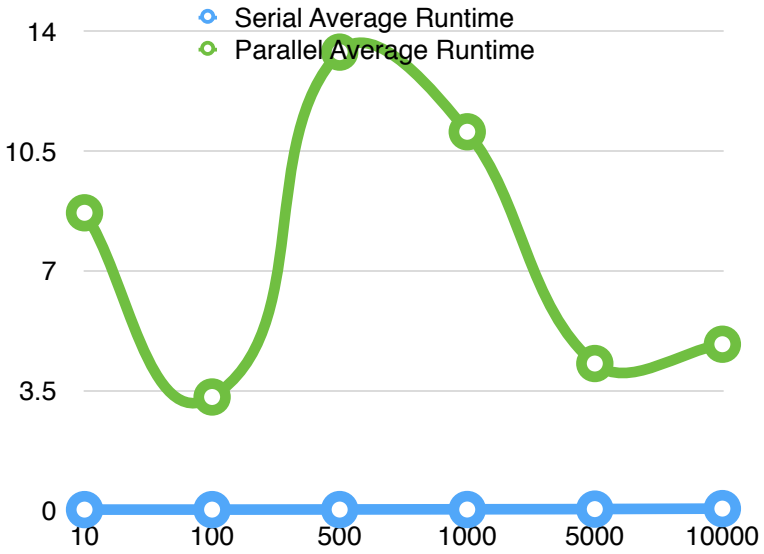




After observing my results, a few things are clear. Merge sort is the definite winner when it comes to large amounts of data. My results support the face that the number of swaps and comparisons by insertion sort is always equal to or larger than merge sort. This would make merge sort the winner; however, after observing runtime it is clear that we'd want to use insertion sort for sets of data around or lower then 100 elements. Otherwise, for sets of data larger then 100 elements, merge sort would be the most efficient method of the two compared.

In order to perform experiment 2, I wrote a sequence file (experiment2.txt) to compare and contrast the runtime of data using both the average and parallel average function.

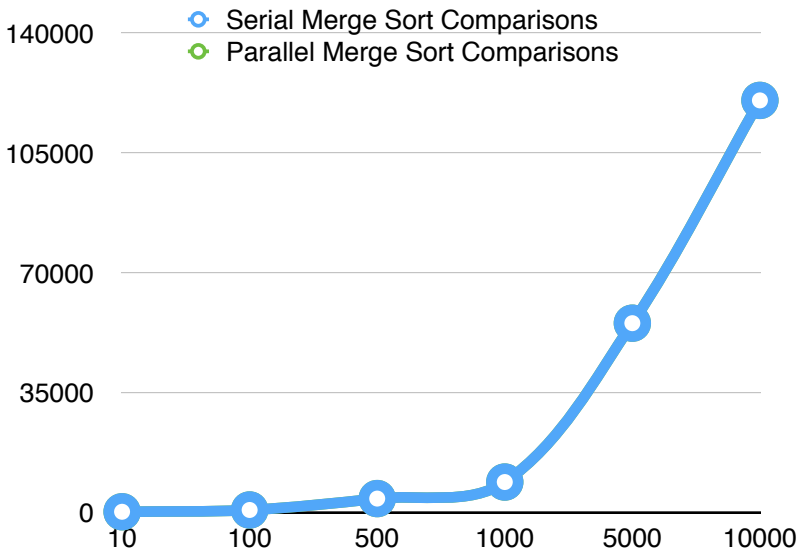
Results:

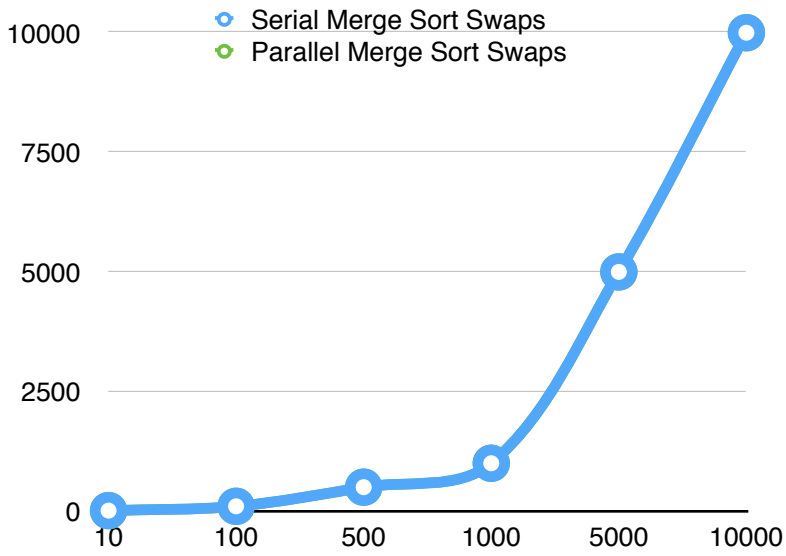


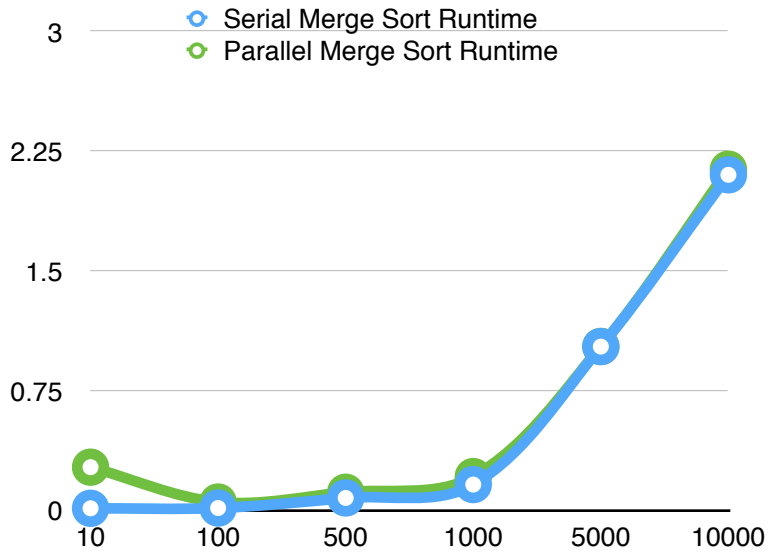
Oddly enough, the results I am receiving show that my parallel implementation is always taking much longer. I assume this has something to do with the amount of threads OpenMP is receiving. After tweaking the number of threads allocated to 4, I am seeing better results. The more this is increased, the better my results; however, the graph above is based on four threads. The point here that I learned is that without a large number of threads, using parallel processing on relatively small lengths of data will result in a slower program then the equivalent in serial. This was a good lesson to learn.

In order to perform experiment 3, I wrote a sequence file (experiment 3.txt) to compare and contrast the comparison count, swap count, and runtime of different lengths of data using both merge sort and parallel merge sort. This is a good opportunity to compare and contrast sorting algorithms that are done serially and in parallel.

Results:







While the swap and comparisons were the same for serial and parallel, I can start to see a trend where the runtime is becoming less different between serial and parallel. As per usual, a serial method outperforms parallel methods when the data set is of a low size. As the length of the data set increases, the parallel begins to take over as the dominent algorithm. Great to know!

These finding, while they may be screwed by programming mistakes, were very helpful in visualizing the advantages and disadvantages of using serial vs parallel. In addition, they gave me a great idea on when to use each scenario (wether the data set was small or large).