In order to carry out these experiments, the provided program, gen-fitness.cc, was modified to receive a command line argument representing the number of lines to print. Then a series of script files was used to generate large amounts of data in a short time.

To compare merge sort and insertion sort, 200 data sets ranging in size from 100 to 20,000 elements were run, then the running time, number of comparisons and swaps was recorded. The running times of parallel and serial average were tested under the same conditions. However, after completing the second round of tests, it became clear that larger data sets were needed to truly compare the parallel and serial algorithms. This is due to the overhead associated with parallelization, which resulted in poor running times for relatively small data sets. For this reason, parallel and serial merge sort were tested using a much wider range of data sets consisting of 1,000 to 100,000 elements. This was necessary to see the differences in how the algorithms perform.

**Insertion Sort and Merge Sort:**

Do you observe a cross-over point? That is, can you recommend when you should use one algorithm over the other?

Initially, for data sizes less than 200, insertion sort runs faster than merge sort. For example, when the data size is 200, insertion sort fishines .01 ms sooner, which is a 5% improvement on merge sort with the same data size. However, merge sort quickly overtakes insertion sort and when the data size is increased to 300, merge sort runs in .033 ms, while insertion sort runs in .059 ms. As the size of data increases, so does the rate at which merge sort outperforms insertion sort. Figure 1 demonstrates the discrepancy visually.

When number of swaps and comparisons are compared between the two algorithms, we see that merge sort performs significantly fewer operations even at data sizes for which the running time of insertion sort is better (see Figure 2 and Figure 3). This means that merge sort is not running slower at smaller data sizes due to the numbers of swaps and comparisons being made, but for some other reason. Insertion sort is an in place sorting algorithm while merge sort is not, so the extra overhead of copying the data many times for smaller values of n could be the cause of slower running times for small data sets. Further analysis is needed to confirm this hypothesis.

In conclusion, it makes sense to use insertion sort for data sizes less than 200 to achieve the best performance. Furthermore, if the goal is to obtain the fastest performing serial algorithm for all data sizes, the optimal approach might not be merge sort, or insertion sort, but rather a combination of both (Cormen, Leiserson, Rivest, Stein, 2009). That is, the best approach might be a version of merge sort, whose base case has been modified to call insertion sort if the data set is small enough.

**Serial and Parallel Average:**

Do you observe a cross-over point? That is, is there a size at which the parallel sum always improves the time for the computation of the average?

For data sizes less than 400, serial sum runs consistently faster than the parallel version. For data sizes between 400 and 700, both versions run equally fast on average. When the size of data is greater than 700, parallel average runs slightly faster. Experimentation with larger problem sizes is needed for a better comparison of running times between the serial and parallel algorithms.

Overall, the performance of parallel average was less consistent than that of the serial version. A similar effect was seen in Assignment 1 when implementing various parallel algorithms. For example, parallel average took .015 ms to compute a sum of 670 elements, but then took .427 ms to compute the sum of 680 elements. Figure 4 clearly demonstrates these anomalies, which can occur at any data size.

In conclusion, parallel merge sort is the better algorithm for use on sets of data with a size greater than 700. However, if consistency in time of computation is required, then serial merge sort might be the only option.

**Serial and Parallel Merge Sort**

Do you observe a cross-over point? That is, is there a size at which the parallel algorithm overtakes

the serial algorithm?

For data sizes between 1500 and 3000, the performance of the two algorithms is comparable. For data sizes greater than 3000, the parallel version consistently runs faster. Furthermore, the rate at which it out performs the serial version remains very consistent for all the data sizes tested as seen in Figure 5.   
 Figure 6 compares the comparisons and swaps made by the two algorithms, and as expected, they perform the same number of swaps and comparisons for a given data set size. Each algorithm must perform the same work, but the parallel version performs some of this work concurrently to achieve a better running time.  
 In conclusion, for data sizes greater than 3000, parallel merge sort outperforms the serial version. Furthermore, if we analyze the growth rate of the merge sort algorithm, we see that the merge component contributes O(n), while the divide and conquer component contributes O(logn). Since this algorithm only parallelizes the divide and conquer component, it is fair to conclude that parallelizing the merge procedure could further improve performance, since it makes up a majority of the running time as the data set grows larger.

Figure 1- Compares the running time of insertion sort and merge sort.

Figure 2- Compares the number of swaps made by insertion sort and merge sort.

Figure 3- Compares the number of comparisons made by insertion sort and merge sort.

Figure 4- Compares the running time of insertion sort and merge sort.

Figure 5- Compares the running time of parallel and serial merge sort.

Figure 6- Compares the number of swaps and comparisons made by parallel and serial merge sort.

References

Cormen, T. (2009). *Introduction to algorithms* (3rd ed.). Cambridge, Mass.: MIT Press.