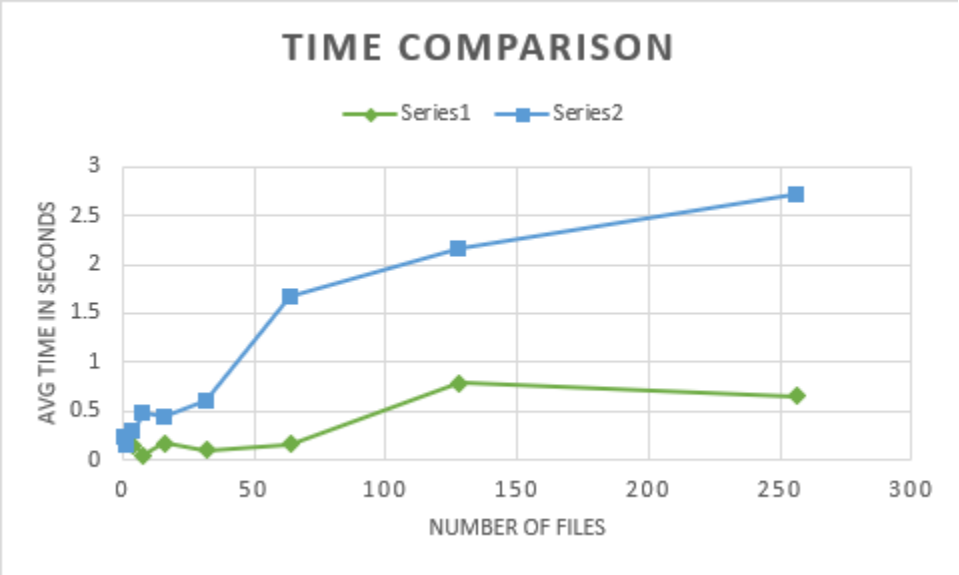
David Tian (dt474) & Rahul Boppana (rsb172)

Time Results:

File Count	Avg Multiprocess (seconds)	Avg Multithread (seconds)
1	0.2376	0.2346
2	0.152	0.2353
4	0.2939	0.1551
8	0.478	0.043
16	0.4239	0.1705
32	0.5974	0.0966
64	1.6708	0.157
128	2.121	0.7903
256	2.7163	0.6509



*Series1 = Process
*Series2 = Thread

Discussion:

There were many factors affecting our results so we tested these files at different times of day to try to diversify our results. We made sure to try times when different other users were present to vary execution variables. Although it is impossible to ensure the accuracy of these averages or reproduce them without an isolated testing environment, there seems to be a clear trend of threading being faster than multiprocessing for this particular type of program.

- Comparison between runtimes – This is probably not a fair comparison method because of the methodology of the two programs. The multiprocess sorter sorts all of the individual files with mergesort while the thread one sorts a much larger file containing all of the data at the end. Ironically however, sorting that one large file takes much

longer and a thread program that sorts individual files would probably be much faster. Otherwise forking and threading are also inherently different. Forking creates its own instance for a new process and has memory associated with it. All of the current data also needs to be copied while threading shares data and accesses the same data. These factors all help explain the results. It is also important to note that using real world time to compare these two programs is not at all practical. Real time is constantly affected by too many factors that give extremely sporadic results.

- Making one faster/slower – Considering the overhead required for processing rather than threading, it is very difficult to make process based programs quicker. One way to compensate for this would be to actually change the algorithm for both programs. But this goes against the project description. Having threading code that completed significantly more operations might match it time wise with process code that simply collected file data and printed rather than sorting each file or something of the like.

- Mergesort as algorithm of choice – For our implementation, the use of mergesort does not make too much of a difference as it sorts individual files for the processing project but then sorts a cumulative file for the threading project. In general, mergesort is a very fast sorting algorithm and sorting n files to combine is something mergesort is designed to do. In the case where mergesort has threading built in, given a file it could theoretically sort the parts of the file simultaneously and then merge them as all the threads have access to common data. Each “recursive call” could spawn a thread and be processed separately. The splitting aspect of mergesort would then be much quicker.