Analysis of Timing Differences between Multi-processed and Multi-Threaded CSV Sorters

Bennett Greenberg, Belal Said 11/25/2018

Figure 1 shows a plot of number of files vs. total time of execution for the multithreaded sorter vs. the multi-processed sorter (with all files in one parent directory).

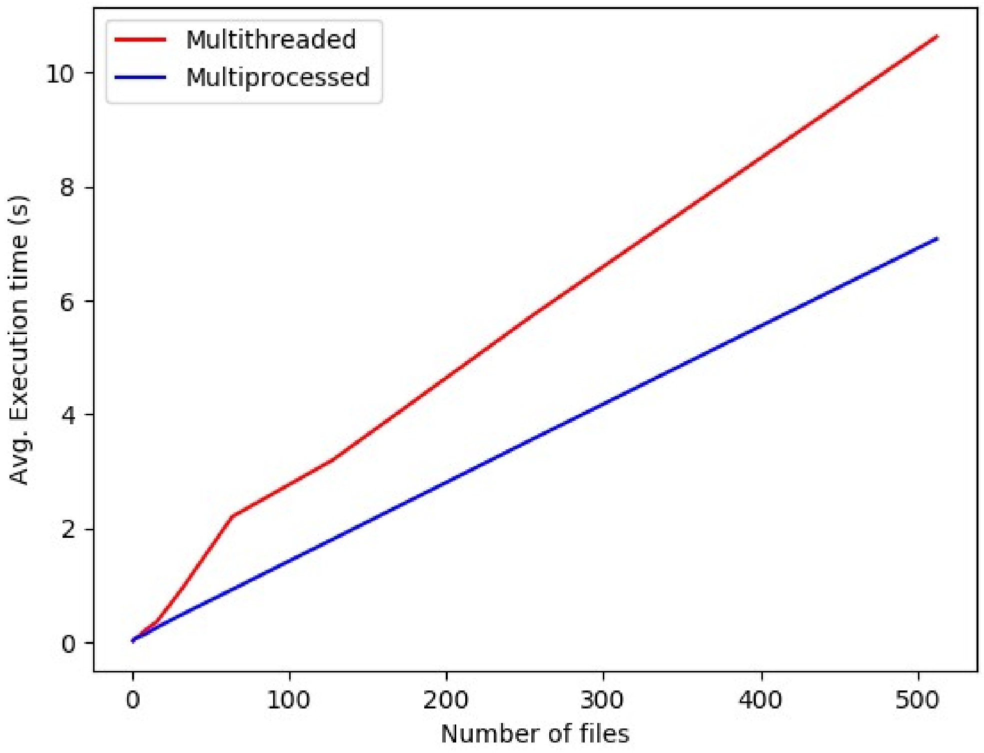

Figure 1: Execution time as a function of the number of files in a single directory for multi-processed sorting vs. multithreaded sorting

For both multiprocessing and multithreading, the execution time scales approximately linearly with the number of files, but multithreading takes some extra time with a small number of files, which causes it to be less efficient than multiprocessing for almost any number of files.

A similar plot for multiple directories with only a single file in each is shown in figure 2.

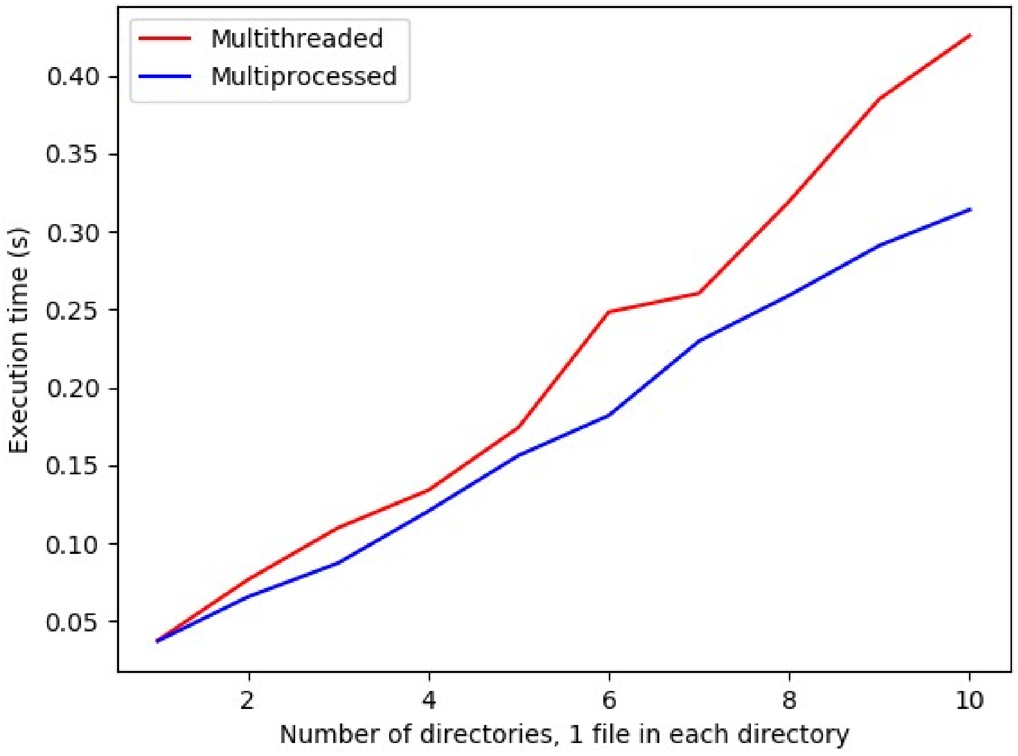

Figure 2: Execution time as a function of number of directories, each directory containing one identical .csv file

This plot shows that the relationship between number of directories and execution time is similar as that for number of files; it is approximately linear with multiprocessing having a slight edge with respect to time. The noise in the middle of the plot (spikey parts) can likely be attributed to the fact that there were a small number of directories tested, and a small number of tests (3) performed (and than averaged).

In figure 3, the ideas of figures 1 and 2 are combined and tested jointly. Each sorting program was run with an increasing number of directories, each directory containing an increasing number of files (2^N files in the Nth directory, with N going from 0 to 10).

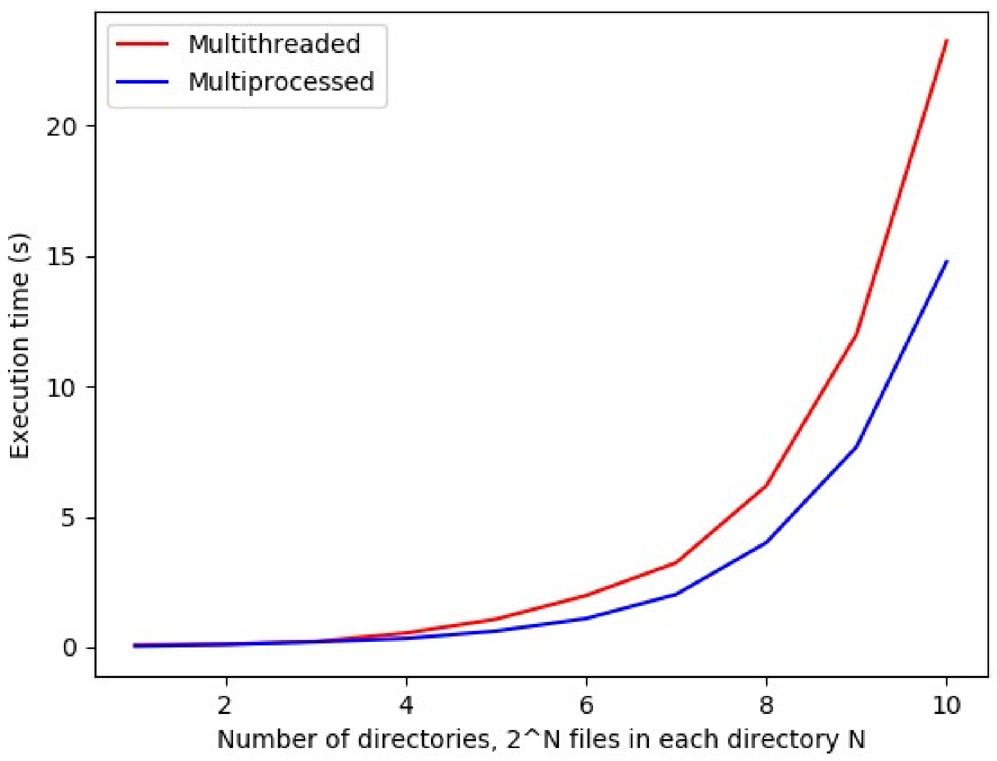

Figure 3: Execution time as a function of number of directories, each with an exponentially increasing number of files

As expected, with an exponentially increasing number of files, the execution time of both programs also increases exponentially. And the multiprocessed program out-performs the multithreaded one slightly, with the discrepancy increasing as the number of directories (and files) increases.

The edge for multiprocessing over multithreading can likely be attributed to the fact that the ilab machines at Rutgers (where these tests were all run) have a lot of cores, but not many threads, so multi-process programming, which happens on different cores, is easier and faster than multithread programming, which happens all on the same core but different threads. The comparison between times for multithreading and multiprocessing is not completely fair, however, because we did make some optimizations in this project (not related to threading or processing) which were meant to make the program run faster. So in actuality, multiprocessing is likely more efficient by a wider margin than the data presented here would indicate.

It would be possible to make the slower multithreading faster, if there were a lot of data in each file and we wanted to break each individual files into various threads/processes to sort also. If this were the case, then there would be much more communication between threads/processes needed. Due to this increase in the amount of communication, multithreading would likely be the faster choice because communicating between process can get very time-expensive the more processes there are.

Mergesort is the right choice for multithreaded sorting because it divides the data into discrete chunks, so each thread can sort its own chunk and then merge everything together at the end. We didn't implement multithreading within sorting the individual files because the amount of data we were working with was relatively small compared to the amount of time it takes to generate more threads, but if we were to scale up the project to one with much more data, mergesort would definitely still be the best choice.