1. The analysis between the run times of the forks and the run times of the threads are not a fair comparison because the forked processes only have to sort their own csv, they do not have to merge all of the files into one global array and sort again. It is likely that merging all the files into one large, sorted csv using only forks() would make its time much larger than it currently is (in the thousandths of milliseconds). Threads should be quicker if they were doing the same process that was required of forks.
2. The discrepancy between the times comes from the overhead of sending all the already sorted files into a large array and sorting that large array again. If the same action were required of us in the fork() project, sending the arrays up to the main process would require a lot of overhead and bring its time up significantly. Also, our implementation of the multithreaded sorter contains a lot of locks, which are system calls. Locking and unlocking the locks adds time to the sort.
3. It might be possible to make the multithreaded sorter (the slower sorter) faster by making the critical sections smaller or just making it sort the csvs without having to merge them all into one large array. Another way to make the multithreaded sorter faster is to use threads in the merge sort itself, which we did not implement. We only implemented the concurrent sorting through multiple threads.
4. I think mergesort is one of the correct sort for a multithreaded sorting program because the threads would actually run concurrently on this type of sort. When the merge sort splits the list into partitions, multiple threads could be working concurrently to sort and merge those partitions rather than waiting for one partition to be sorted and then sorting the other partition.
    a. As an example, when a list of 5 elements is merged sorted in this way, some pivot is chosen and list is partitioned into two parts. Two threads are created from the current thread to recursively sort those two partitions while the current thread “joins” on both of those threads. Then each of the other threads does the same thing until it reaches the base case of just one element.
    b. Another sort that could be used for a multithreaded sorting program is quick sort, which partitions a list into separate parts based on a chosen “pivot” and sorts the left and right side of the pivot. If quicksort were implemented in a multithreaded fashion, a thread would choose a pivot, partition the list into two parts, create two threads to sort the left and right lists, then join on those two threads.
    c. The only wrong answer for a multithreaded sorting program is a program that would not take advantage on concurrent threads running, for example insertion sort or bubble sort.
        i. Multiple threads would not help insertion sort because this sort needs to run sequentially and remember the last state of the sorted list, which would require each thread to wait on the next thread (which is not concurrent use of threads). The same logic applies to bubble sort because the next state of the list is dependent on the last state of the list.