Sorting Competition Documentation

Jake Osselaer

Quincy Schurr

To start the sorting aspect of our sorting competition, we decided to try a bubble sort, just so we could see what the slowest baseline would be for running the program. We implemented the function but never got any results during the timing process because the program took so long to process. We then decided to work on a quicksort algorithm since it is known as one of the fastest sorting algorithms. We successfully implemented a basic quicksort for the task that took a char\*\* array of words and sorted it by length using strlen and then sorted the lengths by ASCII character using strcmp. This method ran incredibly slow, much to our surprise because we thought it was supposed to be a quick method. Our code was running a 550 kb file in about 13 seconds. After re-evaluating the code we decided to implement a few different changes that we hoped would make our code faster. Most of the changes didn’t really make a difference, such as passing by reference vs passing by value, but we were able to simplify a block of code and in doing so made our sorting algorithm run faster. In the earliest version of our quicksort we switched elements of a char\*\* array by creating a temp array and setting that equal to an element in the original array and in then storing it back in another element, but we decided to do the same thing by calling swap. The time it took to run the quicksort algorithm decreased by about an average of 3 seconds. When we first implemented the quicksort algorithm we were using the first element of the list as the pivot element and that worked alright, but we knew we should try to find the median to make our pivot choice an element that was more closely aligned with the middle of the data set. We implemented a successful median of 3 and a median of 5 pivot and looking at the time results after we implemented a median pivot, we could see a decrease in the time. The reason for this is because the program didn’t need to call the function recursively as many times until it reached the base case because we were starting as close to the median of the data set as possible. We realized that another implementation was slowing down our quicksort and that it was the way we were calling the strcmp and strlen functions. When we tested different sized files we saw that our time was not increasing logarithmically as we had hoped, but exponentially. The reasoning for this is that for very large files, the strcmp and strlen functions were getting called a lot since they were included in function calls that were getting called recursively. Being called recursively didn’t seem like a bad idea at first, but then we found out that every time we had to do a strcmp or strlen it took a while to compare the pivot element to another element in the array by iterating through the length of the word to return a result. This really slowed down our sorting, but we couldn’t see a way to get around using strlen and strcmp where we did in our program. We then decided to try merge sort. We successfully implemented this sorting algorithm and it worked a lot faster than our quick sort algorithm. We were able to run our 550 kb file in about .00002 seconds. Our merge sort was still growing at an exponential rate though when we increased our test file size. We discovered that when we called the merge function we were creating a temp array of data type char\*\* and declaring it to be the size of the number of words every time merge was called and that was increasing the time it took to sort. When we tested a 12 mb file it took more than three minutes to run. To fix this we declared the temp array in the prepareData function so that the creation of the temp array would not be added to our sorting time. We really only needed to create the temp array once, and so going back and changing this allowed us to cut down on the time it took to allocate memory to a char\*\* array of such a big size. Due to this change, the time it took to run our sort function was closely aligned the big O(logn) which is what we were hoping for. Another method we tried involved the use of multi-threading. We attempted to add in multi-threading to cut down on the time it took to run the sort function, but we couldn’t correctly implement multithreading and kept getting compile errors, so we reverted back to using a merge sort for our sorting algorithm.