**Ready, Set, Sort! Project Analysis**

By: Sam Coday and Ben Wise

We began this endeavor by attempting to implement two different sorting algorithms, quicksort and insertion sort, with the knowledge that these two algorithms would eventually end up being very useful for a best case sorting algorithm. It might be useful to note that in the supporting code for our sorting algorithm, we have made the design decision to implement two parallel arrays, one of char \* pointers, which point to all of the words and another parallel array of ints which describe the size of the null delimited char string that contains each word. This decision was made primarily so that we could avoid multiple calls to the strlen function in our sorting code. Additionally, it is useful to note that we prepared several input files for our sorting algorithm of different lengths, so that we could test our code on various different size datasets and verify that our algorithms are running with the correct order. Our two favorite data sets were A Modest Proposal by Jonathan Swift (a relatively small data file of 39 kB) and a concatenated file (aptly named AllFiles.txt) consisting of The Bible, The Complete Works of Shakespeare, Alice In Wonderland, The Adventures of Huckleberry Finn, Siddhartha, A Modest Proposal, and War and Peace (a behemoth at nearly 15 MB.)

Our first attempt, was a simple insertion sort, which worked very well on small data sets, but we saw very quickly that as the data set grew from our small data set, to even any of the constituents of the larger data set, the time increased from a few tenths of a second to several minutes and what would have likely been a good part of an hour for the larger data set. We then developed a quicksort algorithm which at first had a few problems, and was exhibiting execution times rather than the times which it should show, because of an improper implementation of quicksort. Instead on pivoting based on a rule of three median, we were traversing to find the actual median. Once this was fixed, we used median of three to choose a pivot that would allow us to more quickly sort.

Logically, the next step was to attempt to marry the two for iterations of quicksort that have a small number of elements in each array. Modifying the original quicksort to turn it into this combination, called introsort, was a simple as introducing an “if” statement that caused the quicksort to recurse using our insertion sort method instead of the quicksort method. In order to properly optimize this approach, we wrote a public interface method to our sort class that allowed us to change the parameter which controls the size of the array at which our introsort switches from quicksort to insertion sort, on the fly from main(). On our laptops, the optimal number was approximately 18 and had we stuck with this approach, we would likely have further explored optimizing this number, especially on the Lyle Genuse servers on which some of the testing might be performed.

Instead of going this route, we decided to pursue a parallelized approach to optimizing our searching algorithm. After researching a few different multi-threading options, we decided to use OpenMP pragma statements, due to their ease-of-use and their (relative) simplicity in understanding both how they work and how to resolve any errors which might pop up, because of the parallelization. In order to do so, we used our working version of quicksort and used a driving function to call quicksort inside of a loop which was preceded by the OMP pragma parallel directives, instructing the compiler to parallelize this portion of the execution of the program. In this way we were able to quicksort on multiple cores much smaller datasets, taking advantage of the faster time of execution for smaller data sets. The problem with this approach is reintegrating the various subsets of our array into one large sorted array. In order to do so we implemented the merge routine from a merge sort, which would introduce an additional time cost, but at the advantage of much, much faster execution of the quicksort, for example, we got a better than 2.1 times speedup (comparing just the quicksorts) between the parallelized and non-parallelized quicksorts we implemented. Finally, we attempted to implement our introsort in a parallel fashion, however, we did not, curiously, see any speedup because of this change. In fact, the parallelized introsort gave us consistently worse times than the parallelized quicksort. We hypothesize that this is due to the fact that the parallelized introsort checked at the recursion point of each iteration of the quicksort as to whether a certain array size limit had been achieved, which seems to have taken so much additional time as to negate any positive effects of the introsort.

In the end, our fastest working approach was **BOGOBOGOSORT BECAUSE WE HAVE THE BEST LUCK OF ALL TIME AND ALL OUR ARRAYS ARE ALREADY SORTED**.

References:

Set Theory: <http://xkcd.com/982/>

Ineffective Sorts: <http://xkcd.com/1185/>