Assignment 1, Map and Reduce

Members: Kevin Slachta, Tushar Sharma, Vivek Rajan

Architectural Description:
Our program read input through the command line through arguments. We utilized booleans to determine whether the user is requesting to sort a list or do a word count as well as whether to use processes or threads. Next we took in the file from the user and begin setting up the environment for the mapper process. This involved reading the information into a linked list in order to allow our mapper to evenly divide our data and send it to maps to solve. Once the data was merged back after the maps,our program would mergesort the entire dataset before sending it to reduce. While merging up after reduce, it combines on the edges to remove any matching words in different reduces.

In order to do the word count we used the same linked list but in this case we generated a key/value pair where key is the unique word and value is the number of times it is seen. Map then puts its list of key/value pairs into shared memory for reduces threads/processes to use. For the sort portion we used a similar architecture and to sort we implemented a merge sort to organize the information. Our reduce function that takes the information from each of the threads/ processes in shared memory then combines it into one structured solution. In word count it takes the sum of the same word.

Efficiency:
Through our analysis of thread and processor approach we discovered that a multi-threaded is slightly faster than a multi-processor approach. Our team came to the conclusion that this was due to the fact that when a new process is created the heap must be copied over. It is seen however that this could be a flaw in threads in that one must be careful when using shared memory. Threads that read/write to the same address could lead to inconsistency in results and thus must be handled carefully

Problems:
Our group came across a few difficulties when running creating our algorithm. The biggest problem we came across was a proper way to use posix shared memory. Our code would sometimes repeatedly access the same point in the list, and other times just jump to the end of the

list. While it was only small errors in the code, it caused huge changes in result without a segmentation fault. With a segmentation fault, we could at least figure out the exact point of failure, but with our code still running, we had to comb through looking for exactly where the issue in memory was. The errors we had were sometimes different between threads and forks. Next, we faced an issue with waiting on all our threads to finish. Our parent thread would sometimes finish without all the child threads finishing and our results would be incomplete. Our biggest difficulty with the assignment was a proper way to debug our errors. Because the threads/processors are running simultaneously and independently of one another it became hard to identify when some error occured. We solved this problem by only working with 2 threads/processes first to verify the correctness of our algorithm and then we were able to scale it to handle more.