Michael Parrilla
John Sweeney
Denny Sabu

Assignment 1 Report

Overview:

The first step in our system is to accept arguments from the command line and pass them to our map function. After the map function checks all inputs, and ensures there are no invalid inputs, it opens the input file, breaks the file up into tokens, sets each token to lower case, and removes the specified delimiters(.,:;?!-“) and stores each token in a vector. Then a two-dimensional vector is initialized to hold each token and its counter and from this two-dimensional vector a number of sub-vectors, specified by input from the command line, are produced. From here the sub-vectors are given to threads or processes, again specified by the command line, and within these blocks the tokens are counted either by the word-counter function or number-counter function which is also determined by command line arguments. As mapping is preformed, across either threads or processes, vectors of pairs are sorted and then written to shared memory. When each thread or process finishes they wait for the others to finish before moving on to the reduce phase. Once the reduce function is called each pair in shared memory is written to a vector of pairs. These pairs are then split based on the number of reduces given by the command line and are sent to either processes or threads. In their respective threads or processes the like pairs are combined, each combination increases the counter in the pair, this is done until each pair is unique, then the pairs are sorted again. Once all threads or processes have finished the separate vectors of pairs are put together and reduced again, the final vector is then written to a file whose name is specified by the command line.

Mapper():

The Mapper takes the entire command line as an argument. With this information, we open up a buffer to read in the entire file. As it reads in the words, it uses strtok() function to break up words that the C++ parser might have kept as one word. (For explain hyphenated words are treated at one word, and the specifications for this project use them as a delimiter). As the tokens are made, they are stored in a vector of strings. From here, the single vector is made into a vector of vectors to split up the work between the threads/processes. From here, the program splits depending on the command-line argument. For processes, each process forks and then and then use the appropriate vector from the vector of vectors. This is done in WordCounter() which creates the key value pairs and then writes the pairs to shared memory. This concludes the processes portion of mapper.

For threads, we arrive at the same place where we have made a vector of vectors. From here, a helper struct is loaded up with a pointer to the appropriate sub-vector and and a mutex that it can use to lock and synchronize. From here, each thread process the data given to it and makes key-value pairs.

Reducer():

From the data created above, we are left with a vector of pairs containing the key-value pairs. We start similar to mapper and create a vector of vectors of pairs to split up the work between either threads or processes, depending on input. From here, if processes are called, each process is given its share of the work and processed to reduce and sort within each process. After each process terminates, one more final reduce is done to make sure that no split up values are not reduced.

For threads, the process is mostly the same. Each thread is given a portion of the work, which it combines the words internally and sorts. After this happens, they are all once again combined and reduced on one more time.

Then, the output is written out to the specified file and the program terminates.

Evaluation:

Both threads and processes operate very quickly due to the divide and conquer paradigm. Since both break up the work between many threads or many processes, the incredible processing power of the iLabs are allowed to shine. With multiple processors, the computer is able to take advantage of the multiprocessing that map-reduce leverages. While both were extremely quick, threads seems to be able to run a little quicker due to the fact that they are more lightweight than creating a whole new processes. However, due to the parallel processing, the difference is negligible and both produce output in a very short period of time.

One of the difficulties we faced for this project involved using shared memory. It was a challenge to figure out how to get data back from each individual process from the map to reduce phases. One way we solved this was to increment the address of our shared memory, this made sure each piece of data written to shared memory was some what indexed, and also made it such that the data on shared memory would not be over-written by other concurrent processes.