Gregory Katz

Chris LoPresti

Aryeh Pechet

MapReduce Framework

## Architectural Design

The program takes command-line input

*./mapred [-wordcount,-sort][-procs, -threads] -[num\_maps] -[num\_reduces][-infile][-outfile]*

Using this input, the program executes through 3 main stages:

### Phase 1: Input

The input stage takes the necessary steps to validate and parse information in preparation for the next phases of the program. First, the phase verifies that all information passed through the function’s command-line input is valid and accessible. Output files are created if nonexistent.

If user requests seem valid, the program begins reading information from the input file. The program reads character by character, ensuring each character is valid as it constructs a word. Upon reaching a delimiter, the program stores the character stream as a word in a node chained in a linked list. Per each word, the program stores a wordcount with the total number of words read. Upon reaching the end of the list, the program has a full linked list with each word read in the program. Finally, this linked list is converted into an array. This design choice was made for faster processing and better concurrency.

### Phase 2: Map

Using the total number of words that we have read, the program shifts to the mapping phase. Using the total count of words read, a new array is created of size count in POSIX shared memory. This count array will hold the count of the word at its parallel index in the word array. Based on the type of implementation from the user, threads or processes, referred to generically as “maps”, are created or forked off with an allotted section of the word array. The number of words allotted to each map is determined by math using the number of maps provided by the user. Number of maps will never exceed number of words even if the user specifies such, but multiple words may be allotted to a single map. Each map then updates its allotted array values in the count array to 1, signifying the distinct presence of each word. Duplicates occurrences will be condensed in the future phase. Once all threads have joined or the main process has completed a sort of its word array, the mapper phase has completed.

### Phase 3: Reduce

Based on the previous phases, reducer now creates threads or processes, referred to generically as “reduces”. Similarly to map, each reduce manipulates a subset of the indices within the two arrays. The subsets do not overlap to ensure concurrency. Each reduce will iterate through their array of words from their smallest to largest index. Since the word array is sorted at this point, duplicate occurrences of words will be grouped together in subsequent order. If there are words in the word array that are the same, the reduce accesses its count array counterpart in shared memory. Within the count array, each parallel position where the target word is found in the word array is added (or reduced) to the leftmost position of the count array. Any parallel position to the right will have its count set to 0. The reducer knows to ignore these 0 placeholders from now on and to only consider the reduced entry. When a different word is encountered in the iteration, the process resets for the new word.

Upon completion of this stage of the phase, when all threads are joined or the main process completes manipulating the data, a final reduce is called on the entire array. Entries that are zeroed out will be skipped automatically. This is accomplished by skipping the number of iterations equal to the reduced value of the leftmost word in the group. Final reduce is activated to ensure that overlapping words shared between two reducers are joined together if they exist.

At this point, the contracts of the program are nearly complete. The MapReduce program finally reference both arrays and the user’s command-line arguments to determine what to print. At an index I, both the word[i] and its associated count[i] are printed so long as count[i]>0 and user specifies -wordcount. Otherwise, if -sort is chose, word[i] is printed count[i] number of times. Upon successful printing, the memory is cleared and the program terminates successfully.

### Extra Credit

The Extra Credit implementation calls the same input phase with a different approach to mapping and reducing.

The map phase will instead call a function to dish out threads which will grab n number of words from the word array and copy them into a joint BST. Words are inserted in specific positions according to their value compared to other words in the tree. At a specific node x in the tree, if a node y is being inserted, node y will either be inserted to the left of node x (if y<x) or to the right of node x (if y>x). If the value of x=y, node x’s count will just be incremented. If the node y reaches the end of the tree, in which no word with equal value exists throughout the entire tree, it is inserted with a value of 1. This new implementation does not work without the use of mutex locks ensuring concurrency.

Once this phase is completed, the BST is printed with the same criteria as the map reduce thread (same process for sort or wordcount)

## Performance Evaluation

Originally, this project was implemented solely with the intention of manipulating nodes in a linked list structure within shared memory or on the heap. This process proved extremely taxing. The method, requiring numerous system calls, encountered long runtime and demonstrated inefficiency. The decision to move to array structures greatly increased the speed and efficiency of the program from minutes to seconds. Random access was a huge plus upon implementation compared to scanning through node pointers. Tests of csv files containing greater than 600,000 words for word sort takes about 20 seconds.

Mergesort runs at nlogn on the word array for both processes and threads. It was the best option to ensure MapReduce was not bottlenecked by sorting.

## Technical Difficulties

We had to use our shared memory index count array approach as we were limited with shmget. We could not store pointers in shared memory, or even really 500k words. However we were able to store 500k+ integers with ease so it made more sense to have an array of counts where the indexes corresponded to the indexes of the words.

Another issue we had was for our BST extra credit approach. We had to implement a BST using while loops rather than recursion, as we were finding we were limited to 255 recursive calls, which hit us with an overflow error. When switching to a while loop we were able to do this no problem, however this BST approach does take a couple minutes rather then seconds when doing the shared memory index array approach.