Gregory Katz gdk26

Chris LoPresti cpl62

Aryeh Pechet aap220

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 phases:

### Phase 1: Input

The input phase 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 request is 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. As we store each word, the program updates a word count; which will later be the total number of words read. Upon reaching the end of the file, the program has a full linked list containing 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

Before the mapper will start to create threads or processes, it first assigns a value to a few key global variables. We fist determine how many words will be allotted to each thread or process fork. We then do the same thing for the future thread or process calls for reducing. The number of words allotted to each map or process is determined using the number of maps provided by the user and the total number of words. Calculating the number of words per reducer ahead of time for thread or process will allow us to jump right into the reducer phase. We also determine if the amount requested map and reduce threads or processes makes sense. Number of maps will never exceed number of words even if the user specifies such, as we will ensure to give you at most a number matching the total word count.

Now, using the total count of words read, a new int array is created of size ‘total number of words’ in POSIX shared memory. This array will hold the count of the word at its parallel index in the word array; we will refer to this array as the count array throughout the remainder of this report. 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. 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, the mapper phase has completed. The mapper will now call sort, which begins the ‘shuffle’ phase. The shuffle phase is just one call to our sort function which will merge sort our list of words. Because the shuffle phase is just merge sort, we will not create its own section in the report. Once done, the mapper calls reduce.

### 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 section on the 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.

When all threads or processes are joined, 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 an alternate approach to mapping and reducing.

The driver will still call the mapper and configure global variables, however, once done with the global variables the mapper will take a different route. It 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 sorted as they are inserted into 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, and we were unable to store pointers in shared memory. The method, requiring numerous system calls, encountered long runtime and demonstrated inefficiency. We also tried to store each individual node in shared memory but we were unable to have more than a set number of shared memory units open. The decision to move to array structures greatly increased the speed and efficiency of the program from minutes to seconds, and we could store a giant array in shared memory without any issues. Tests of csv files containing greater than 600,000 words for sort or word count 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.

Due to our use of the BST for extra credit, we have noticed a substantial increase of performance time. We must read in n number of words, then do n insertions, and numerous comparisons before finally inserting. We also can only insert one word at a time to prevent corrupted pointers or seg faults. Each thread can only insert a word to the combined BST if it has the mutex lock locked down. We then must traverse all the way to the bottom of the tree, print a leaf and work our way back up, printing and freeing. The extra credit takes 600k words about 2.5 minutes to complete, which is much slower. Though it is slower, it is much different than our map reduce procedure, which is what the instructions asked for.

## 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 could 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 could do this no problem, however this BST approach does take a couple minutes rather than seconds when doing the shared memory index array approach.